CHAPEL BY EXAMPLE

IMAGE PROCESSING

Revised for the April 2018 Chapel 1.17 Release

will not work with 1.18 or later

Table of Contents

INTRODUCTION

A few weeks ago (in April 2015) we found a post on the web that, in the middle and almost as an aside, praised several lesser-known languages that were trying to tackle parallel computing. Always alert to random strong recommendations from strangers signaling something interesting, we looked at each. One of these languages was Chapel, developed by a team at Cray based on their experience with high-performance computing. We read some of the presentations on the Chapel home page and saw several things that were intriguing. A scan of the language specification reinforced the feeling, and we decided to dive deeper.

The language's goal is to cleanly express any type of parallelism in a program and map it to machine configurations ranging from multi-core to multi-system. It wants to avoid low-level constructs, such as threads or the MPI or OpenMP architectures, for all users without dictating how to achieve parallelism, as previous languages had done. To quote their literature, "Chapel was designed with concepts that support data parallelism, cooperative task parallelism, and synchronization-based concurrent programming." The language focuses on high-level abstractions so that they are independent of the detailed execution, while still offering the opportunity for low-level control. You can manually partition data into chunks and distribute it over the machines, or you can treat the data as one piece and let the language do the work for you. All this is wrapped in features borrowed from modern programming languages to make a comfortable programming environment.

It sounds good. Of course here we focus mainly on image processing on multi-core CPUs and GPUs, not on big hardware. The issues faced there are outside our experience. The white papers, tutorials, and documentation, however, hinted that there might be a good overlap between this problem domain and some of the basic features of the language – array support, domains, built-in parallel loops. To learn the language, we decided to implement several image processing programs in it and see how they developed. Part of our motivation is the love of learning, part is answering the practical question "is this something we can use?" As we developed these programs, chosen to increase in complexity from per-pixel operations to local neighborhoods to iterative algorithms, we would document the language and how it applies to this problem domain. This is the result.

The outcome isn't certain. We might find there's not a good fit. We aren't trying to sell the language, to convince you Chapel is the next great thing – and why aren't you using it already? Let the language designers and Cray (or whoever has control of the project) do that. We believe that the strong, interesting features of any language will become clear as you read about them. They'll prick your interest, get a nod of approval. If this happens often enough, then you too might decide to investigate further, and then hopefully this document will help you get started. We'll keep our opinions to ourselves until we get to the end.

This isn't a start-at-the-beginning kind of guide. We assume you know how to program and have some experience with C (as the first chapter will show). We will cover the basics of the language, from types to data structures to operators to statements, but our focus is more on the aspects that differ from our expectations than on basic programming techniques, and the code snippets will either focus on the layout of the commands or how their behavior differs. We'll generally build up the program for each chapter in pieces, adding more features and using more of the language as we go along. We design on Linux boxes and the development flow for compiling and running executables is based on that. The tool chain that you need to build the Chapel compiler – specifically gcc and make – is all you'll need for these programs. You will need the PNG library and header file; if not provided with your distribution, look at http://www.libpng.org.

The Chapel home page is at https://chapel-lang.org. You'll find documentation there – tutorials, presentations, and the language spec – as well as a download for the compiler source. One you have that, we can start. The examples and text are up-to-date for the latest (April 2018) release.

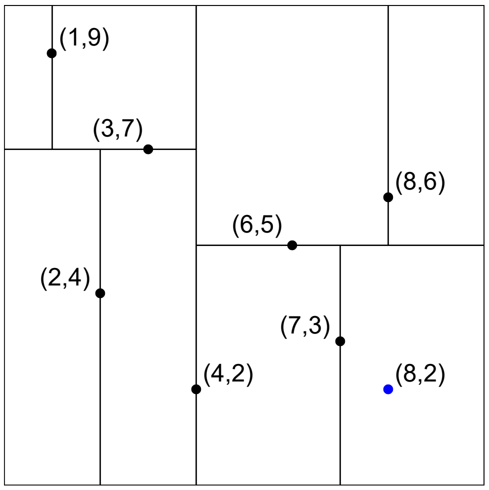
The home page for this project is at http://www.primordand.com/chapel_by_ex.html. You can find downloads for all program files and on-line and PDF copies of this text there.

The Chapel 1.18 release made deep changes to classes, including adding lifetime specifications and replacing constructors by initializers. This breaks the method we used to access image data stored in C and used by external libraries. The example programs with the text will not work with this version or later.

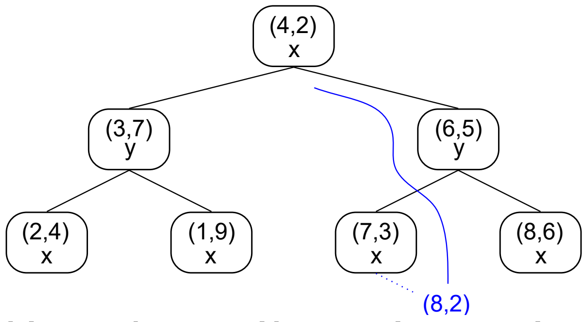
AN EXAMPLE

"Hey, we found a programming language that looks interesting so here's a long description of how to use it" might not be the most compelling motivation for spending time learning about Chapel. Here's an example that shows many of the core ideas behind the language in action.

A kd-tree is an efficient way to search a k-dimensional space. It is essentially a binary tree which cycles through the coordinates level by level, so that the root node splits the space along x, the second level by y, the third again by x (for a 2-dimensional space) or z, the fourth by y (2D) or x (3D) or the fourth coordinate, and so on. The goal of the example is to divide the space with a set of ordered points, building a balanced tree. We'll use a flat array with 1-based indexing to store it, so the left child of the parent `p` is at `2*p` and the right at `2*p + 1`. The algorithm will run in parallel as it recursively builds the tree layer by layer.



Plane division for sample kd-tree



kd-tree with point and base coordinate in node

```
param ndim = 3;
config const treedepth = 5;

var Ltree : domain(rank=1) = { 1..(2**treedepth)-1 };
var tree : [Ltree] ndim * int;
var data : [Ltree] ndim * int;

/* cbase is the coordinate 1..ndim which is the primary sort key for
   this level.  p is the node in the tree to assign in this pass. */
proc assemble_tree(ref points : [] ndim * int, ref tree : [] ndim * int,
                   cbase : int = 1, p : int = 1) {
   const cnext = if (cbase == ndim) then 1 else (cbase + 1);
   const medpos = partition_median(points, cbase);
   tree(p) = points(medpos);
   cobegin {
     assemble_tree(points[..medpos-1], tree, cnext, 2*p);
     assemble_tree(points[medpos+1..], tree, cnext, 2*p+1);
   }
}
```

```
    assemble_tree(data, tree);
```

Let's look at this piece by piece.

```
    param ndim = 3;
    config const treedepth = 5;
```

These two lines define two constants which are type-inferred to be integers. The `param` keyword means `ndim` is a compile-time constant. A `const` is a constant set at runtime and `config` means the compiler will automatically create a command-line option `--treedepth` that allows you to change the value each time you run the program. You do not have to parse the command line to get this behavior.

```
    var Ltree : domain(rank=1)
```

Arrays are a primary data structure in Chapel. They are backed by domains which define the indices. Domains can be a series of values, as we'll use here, or discrete, like a set of hash keys or sparse indices. The `var` means we can change the value of the domain; any array that uses it will automatically re-size. The colon is followed by a type declaration which says that `Ltree` is a domain. The rank of the domain is the number of dimensions, so here we're saying `Ltree` is a linear set of indices.

```
                                   = { 1..(2**treedepth)-1 };
```

Domains are built on top of ranges of values which define the continuous set of indices. You can think of ranges as the bounds of a `for` loop, and they can have an increment between values as well as a direction. The .. notation indicates this is a range from 1 to the number of nodes in the tree (inclusive), which is one less than the power of two of its depth (ie. a tree of depth 3 has 7 nodes, 4 has 15). The ranges are placed inside braces when setting the domain. The assignment is done when `Ltree` is declared so this is an initialization of the variable but all variables have default initial values if not provided.

```
    var tree : [Ltree] ndim * int;
    var data : [Ltree] ndim * int;
```

This defines two arrays. `[Ltree]` indicates they are arrays over the `Ltree` domain. The elements of the array will have type `ndim * int` which represents an `ndim`-element tuple of integers. With `ndim=3` our points will look like (x, y, z). The `data` array will hold the unsorted points, `tree` the sorted. A kd-tree cannot be modified easily once built, so the points will need to be copied into `data` before calling `assemble_tree()`.

```
    proc assemble_tree() { .. }
```

This is a function declaration where

```
                              ref points : [] ndim * int,
```

is the first argument, an array of `ndim`-dimensional point tuples. We don't have to specify the domain in the declaration and can get it later with `points.domain`. `ref` is an argument intent and says that the array is passed by reference. This means that we will be able to change the contents inside the procedure and the value in the caller will also change. Other intents include `in` for pass-by-value (where the value in the caller does not change) and `out` which copies the value assigned within the procedure to the argument in the caller.

```
                                                    ref tree : [] ndim * int,
```

is also an array of `ndim`-dimensional tuples. Because we haven't specified a size between the brackets for either array, they may be different, although Chapel has a way to require that they be the same.

```
                              cbase : int = 1,
```

defines an integer argument. The default intent for primitive types is `const in` which means the argument's value cannot be changed within the procedure (but `in` alone would allow this). `cbase` is assigned a default value of 1 in case it is omitted when calling the procedure. You can also specify arguments by name when calling.

```
                                                  p : int = 1) {
```

`p` is the root node of the tree and should have the value of 1 on the first call. We also provide a default so we can omit the argument and get the correct behavior. Such arguments must come at the end of the list if you want to mix them with arguments without defaults.

`assemble_tree()` has no return type, which would be placed between the closing parenthesis and opening brace.

```
      const cnext = if (cbase == ndim) then 1 else (cbase + 1);
```

This determines the coordinate used for sorting the next pass. The `if..then..else` construction is Chapel's ternary expression and we use it to cycle the coordinates through the dimensions of the space. The `const` intent means we cannot change the variable within the function after initialization. The type of `cbase`, `int`, comes from the expression.

```
      const medpos = partition_median(points, cbase);
```

Here we call another function, `partition_median()`, to find the median point of the points array. It works like the partition function used in QuickSort: the elements of the points array are re-ordered so that those smaller than the median lie to the left of it, those larger to the right, and the position of the median is returned and set to the constant `medpos`. Note we haven't provided an implementation of this function in this example. A simple but very inefficient approach would be to sort the points array using the `cbase`'th element of the tuple, and then to return the index of the array's middle.

```
     tree(p) = points(medpos);
```

This copies the median point into the tree. Chapel uses parentheses for array indexing.

```
     cobegin { .. }
```

Chapel has four statements for parallel execution. `cobegin` says to run each of the statements in its block in a separate task. There is also `begin` which runs statements asynchronously, `forall` which can split a range into many subranges and run each in a task, and `coforall` which must execute each iteration of the loop in a separate task. You can think of the first two as being oriented to do tasks (statements) in parallel, while the other two process data (indices) simultaneously. With the `cobegin` we will run the two recursions in parallel; we don't need any other code else to get this behavior.

```
      assemble_tree(points[..medpos-1], tree, cnext, 2*p);
      assemble_tree(points[medpos+1..], tree, cnext, 2*p+1);
```

These two recursions operate on the points that are smaller and bigger than the median. The smaller points go in the left child at `2*p`, and we take the first half-array of the points by slicing with the open-ended range

`..medpos-1`. This slice takes the intersection of the array's range with one that has no beginning. It ends just before the median. The result is a new range that starts with the first point and goes to `medpos-1`, inclusive. On the other side we need to build the tree at the right child `2*p+1` over the subrange `medpos+1..`, where the new range will start after the median and go to the last point. Because the children don't overlap we don't have to worry about changing the array in independent tasks.

```
    assemble_tree(data, tree);
```

The procedure call starts the process. As we said, we can leave out the `cbase` and `p` arguments because they have defaults. Chapel has well-defined rules for resolving a mixture of positional, keyword, and default arguments.

The core of this algorithm – partitioning and recurring on separate subranges of the input – is the same used for QuickSort. You'll find a full implementation of a kd-tree using a different approach in the RANSAC chapter.

A bullet-list of additional language features that aren't in the example includes

- Language support for defining parallel tasks which the runtime will automatically create and run for the hardware available, as well as offering control over how the mapping to hardware will be done.
- Synchronized and atomic variables.
- Generic types and data structures and polymorphic functions, where the specific versions are generated at compile time and not by dynamic routing.
- Object-oriented data structures, with classes and records that combine data and functions and inheritance.
- Modules to further partition programs into separate libraries.

OVERVIEW

The programs we'll be writing were chosen to build in complexity through the tutorial.

1. First we need to read images. We'll use the PNG library, which requires binding to some C code to handle the interface. We'll need to cover types, variable and function declarations, and two structural types, classes and records. We'll see how Chapel organizes programs into modules.

2. Next we'll do color conversion to transform our RGB input into another color space, the CIE L*A*B*, where the L plane is a good greyscale version of the image. This is a pixel-by-pixel operation and will introduce us to arrays, domains, and ranges. We'll cover the standard language – expressions and statements. Then we'll extend this to other color spaces and introduce Chapel's support for first-class functions.

3. For the third example we'll write a Gabor filter, which is a large convolution run over an image to detect oriented edges. We'll learn about subdomains and subranges to iterate over the the convolution kernel, and the syntactic support Chapel offers for defining them. We'll see reductions that combine multiple results into one.

4. The fourth example introduces us to Chapel's support for parallelism, either at a task (statement) level or by dividing a data set and working independently on each part. For the first we'll make a bank of Gabor filters to evaluate at the same time, and for the second we'll split the RGB space to verify the bounds of the LAB color correction. The language features we'll study are the statements that generate parallel tasks, and synchronization variables.

5. Our next example is k-means clustering which we'll use to quantize (reduce) the number of colors in an image. This uses an iterative refining of how pixels are assigned to clusters, which can, and will, be done in parallel. We'll talk about Chapel's support for atomic variables.

6. Leading up to our final program we'll need to implement a FAST corner detector to mark features in images we want to compare. The detector looks around a small ring to see if the center is the tip of a light or dark corner. We'll write a custom loop iterator to step around the ring, and we'll study Chapel's generic data structures and procedures in order to store a list of the corners found.

7. Finally, we'll use the corner detector to align two images, compensating for rotation, scaling, and translation. This technique uses a random sample of points to determine the alignment, running many trials in parallel to find the best correspondence. There won't be any new language features. We ourselves haven't done this before and the goal of the exercise is to see what developing new image processing code is like in Chapel.

INSTALL

Once you have downloaded the distribution from http://chapel.cray.com and untarred it, you'll find a README with instructions for compiling.

There is a simple version that does not use many third-party libraries (which are provided with the distribution). To compile it you need to source a script in your shell and then run make. We run bash, so the commands run from the top-level directory CHPL_HOME are

```
source util/quickstart/setchplenv.bash
make
make check
```

The last step runs built-in tests to make sure everything is correct.

The full version uses a different script.

```
source util/setchplenv.bash
make
make check
```

This version of the script sets several environment variables to add in extra features. (Well, technically it does not set several variables, and their default values cause the extra features to be compiled in.) You'll find a full list in CHPL_HOME/doc/rst/usingchapel/chplenv.rst (HTML versions also exist). The most important are:

variable	purpose	quickstart	full
CHPL_TASKS	threading library to use	fifo (pthread)	qthreads
CHPL_HWLOC	package to detect hardware setup	none	hwloc
CHPL_COMM	inter-processor communication package	none - local CPU only	
CHPL_GMP	GNU Multi-Precision library	no	integrated
CHPL_REGEXP	regular expression library	no	integrated
CHPL_LLVM	use LLVM compiler, can embed C directly	no	integrated

You can set these manually before compiling if you want to force the behavior. Be aware that LLVM support takes a long time to compile and you might want to pass -j # to make to do it in parallel.

Before using Chapel you will need to source the setchplenv script from CHPL_HOME once after login. You can then run the compiler anywhere. We set the following variables instead in our .bashrc

```
export CHPL_HOME=<where you unpacked the distribution>
export CHPL_HOST_PLATFORM=`"$CHPL_HOME"/util/chplenv/chpl_platformpy`
export PATH="$CHPL_HOME"/bin/"$CHPL_HOST_PLATFORM":"$CHPL_HOME"/util:$PATH
export MANPATH="$CHPL_HOME"/man:$MANPATH
```

You will also need to set the other CHPL_* variables you forced before building the compiler.

Inside the distribution you'll find these important directories and files:

doc/rst/language/chapelLanguageSpec.pdf - complete description of the language

doc/rst/technotes - descriptions of features being added to language (Work In Progress)
doc/html/index.html - browser version of the documentation, including modules
examples/primers - code examples for language features
examples/programs - complete, simple programs
examples/spec - code snippets from the spec
examples/benchmarks - various performance benchmarks
modules/standard - standard libraries , including IO, math functions, timers, and error handling
modules/internal - parts of the language that are written in Chapel
modules/packages - other libraries
third-party - support programs from outside the Chapel team used to build the compiler and runtime

There's a style definition for emacs and vim in highlight/[emacs|vim].

Compiling and Running a Chapel Program

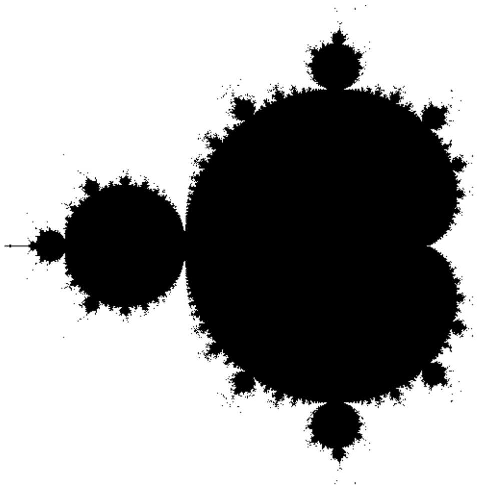
The distribution contains many example code snippets and programs. Let's pick one to see the basic compile and run cycle. Copy mandelbrot.chpl from CHPL_HOME/examples/benchmarks/shootout to a project directory. Then do

```
chpl -o mandelbrot mandelbrot.chpl
```

to compile. To generate an n x n image type

```
./mandelbrot --n=16000 > img.out
```

This creates a 16000x16000 image in PBM format with the result. The `--n` is a run-time constant that Chapel allows to be set on the command line.


Mandelbrot set

Typing `-h` or `--help` the program name prints a table with all the command-line options that are available.

Typical for a compiler, there are pages of options for the chpl command line. See the man page. We will need only a few, though, which we'll cover as we go along.

IMAGE INTERFACE (png)

Introduction

Image processing means working with images, which means we need a way to read them into our programs and write out the results. Our first task is to figure out how to interface with C libraries, namely libpng, which can read and write images in the PNG format.

The Chapel compiler translates source files into C, which are then compiled normally. This means interoperability is good. You can explicitly declare structures or functions or variables in Chapel and the compiler will link them appropriately. If the compiler is built with LLVM support then Chapel can parse header files or embedded code blocks and automatically set up the linkage. We'll assume this isn't the case, and our job will be getting the type mappings correct.

We'll begin by writing and testing a simple C program that uses libpng, and then duplicate it in Chapel using a small set of functions. We'll follow this by tying the C data structure to Chapel to access the image directly, and we'll conclude this exercise by looking at how Chapel organizes code into modules and programs.

The directory for this chapter is `png`. Programs can be built by doing `make name`, where the name is without the file suffix (.c or .chpl). We'll also give the full compiler command in the text if you want to do it directly. Executables are put in the `bin` sub-directory. `build` holds auto-generated dependency lists and intermediate object files. The sample image for this exercise is bear.png.

The files and image for this chapter are found here (as zip file). A PDF copy of this text is here.

C Library (img_png_v1.c, test_png.c)

img_png is a C file that provides functions to read and write PNG images and a data structure that holds the image size and separate arrays for the R, G, and B planes. This data structure, called `rgbimage`, is dynamically allocated and must be manually freed. Functions take a pointer to an `rgbimage` pointer if they create or destroy an image, or just an `rbgimage` pointer if they work on it. Make sure the initial value for the pointer is `NULL` or else the allocation routine will assume it already holds an image and will try to free it, which will raise a segfault.

```
rgbimage *img = NULL;
read_PNG(filename, &img);
free_rgbimage(&img);
```

img_png_v1 (the _v1 indicates we'll be modifying the code later, in this case to interface better with Chapel) contains three groups of functions. The first provides PNG support: `PNG_isa()` tests if a file contains a PNG image; `PNG_read()` creates a new image with the picture from disk; and `PNG_write()` writes a picture to a file. The second group manages rgbimages: `alloc_rgbimage()` creates a black picture (ie. R, G, and B all zero) of a given size; and `free_rgbimage()` releases the memory. Not calling `free_rgbimage()` is a memory leak. The third group accesses pixels without the caller knowing about the `rgbimage` data structure: `read_rgb()` gets the R, G, and B values at a point; and `write_rgb()` changes them. This group is to get us started, until we figure out how to directly access C arrays from Chapel.

The code should be straightforward. The PNG functions follow the guidelines in the library's documentation, including `setjmp()` and `longjmp()` for error handling. The memory functions use `calloc()` and `free()`, and the access functions are array operations. The color planes are stored row-by-row, packed into a 1D array, where a point (x,y) in an image with ncol columns is located at index xy

```
    xy = y * ncol + x;
```

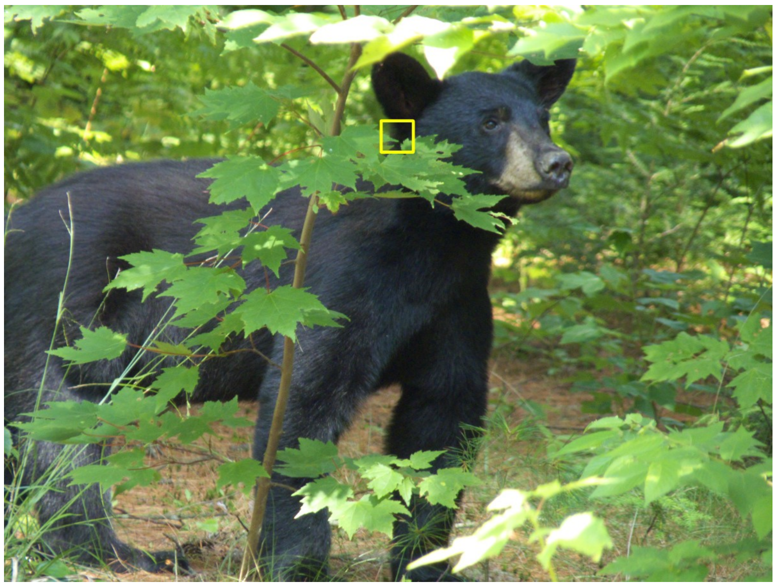
test_png.c is a simple test program for these functions. It reads an image, prints out the R, G, and B values at a pixel, then changes them to 1, 2, and 3 respectively before writing the result to a new file. Compile it with

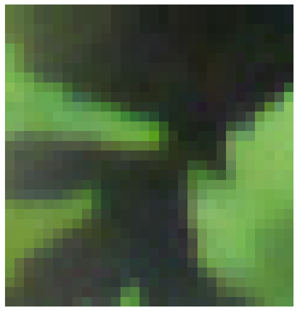
```
    gcc -c -o build/img_png_v1.o img_png_v1.c
    gcc -o bin/test_png test_png.c build/img_png_v1.o -lpng
or  make test_png
```

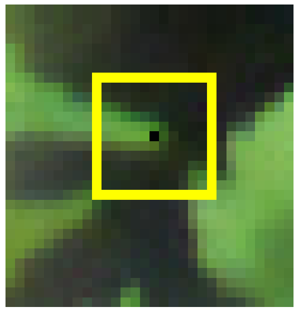
which compiles first img_png_v1.c, putting the object file in the build sub-directory and then test_png.c, putting the executable in bin. Run it with

```
   bin/test_png bear.png tstimg.png 615 212
```

The point (615, 212) is the tip of the leaf below the bear's right ear. It has R 83 G 156 B 25, which you can confirm in the program's output. If you open tstimg.png you can verify that the pixel has changed to R 1 G 2 B 3.


Original image bear.png


detail of leaf tip


changed pixel on leaf

C interface from Chapel (rw_png_v*.chpl)

The goal of this chapter is to duplicate the test_img.c program, but this time in Chapel. That is, we'll show how we can read an image, access and manipulate it, and save the result. The program logic is trivial; the difficulty will be interfacing to the C world. The concepts we'll need to understand are variable definitions and the Chapel basic types, procedure declarations to mirror the functions in the C library, and what Chapel requires to link to external code, both in the source file and when compiling.

Primitive Types

Chapel has a small list of primitive types: int and uint (unsigned); real, complex and imag(inary); bool(ean); and void. Types can be given a size in bits. Currently supported are

	8	16	32	64	128	bits
int	x	x	x	D		
uint	x	x	x	D		
real			?	?D		
imag			?	?D		
complex				?	?D	
bool	x	x	x	x		(default varies)

? means potentially supported depending on the hardware, D is the default bit depth, and x an otherwise allowed bit depth. Sizes are written in parentheses after the type

```
    int(32)
```

Use 0x[0-9A-Fa-f]+ to represent a hexadecimal number, 0o[0-7] for an octal, and 0b[01] for a binary. The x, o, and b can be capitalized, but anybody who uses 0O is evil and deserves the everlasting pain they will suffer.

An imaginary number is followed by i. The format of complex numbers is 'a + bi' or (a, b), a tuple. The real part of a complex variable `c` is gotten with `c.re`, the imaginary with `c.im`. You can use these components on the left side of an assignment.

```
    c = -1.0 + 1.0i;
    writeln("real part " + c.re + "   img part " + c.im);
    > real part -1.0   img part 1.0

    c.re = 2.0;
    writeln("real part " + c.re + "   img part " + c.im);
    > real part 2.0   img part 1.0
```

The boolean values are `true` and `false`. In conditional expressions for `if`, `while`, and `do-while` an integer is treated as `false` if 0 and `true` if non-zero, and a class as `false` if `nil`, `true` if not.

Strings are a class and not a primitive type. They're implemented as an array of characters and a variable with the length. They only support ASCII and may have restrictions on copies on remote machines.

Void is only used as an empty argument list or return type.

Variable Declarations

Chapel has three ways to declare a variable: either a type is provided and there is a default initialization value, or an initialization value is provided and the type is inferred from it, or both are given. So

```
    var a : int(32);            /* default to 0 */
    var a = 1234;              /* int(64) */
    var a : int(32) = 1234;    /* int(32) */
```

The kind of the variable must precede the name. There are three. `var` is a normal variable, `param` a compile-time constant, and `const` a runtime constant. A parameter can be initialized with a primitive value, an enumerated type, or a simple expression using the normal math and comparison operators. Several of the functions in the Math standard module, including `abs()` and `mod()`, are also available if they are declared with a param intent. A constant can be of any type, but must be initialized and keeps that value for its lifetime.

Numbers default to 0, booleans to `false`, and strings to the empty string. Classes default to `nil`.

A variable name is any alphanumeric character, the underscore, or dollar sign. It must begin with a letter or underscore.

Multiple variables can be combined in a comma-separated list with types and initializations shared.

```
    var a, b=-1, c, d=1, e: uint(8), f: string;
```

creates five 8-bit unsigned integers (`a`-`e`) and one string. `a` and `b` are set to -1, `c` and `d` to 1, and `e` and `f` get their default, 0 and "".

There are two optional modifiers to the kind of variable. One is `extern`, and we'll get to it soon. The other is `config`. This creates a configuration variable/parameter/constant that can be initialized from the command line. For example, from the Mandelbrot benchmark program (in CHPL_HOME/examples/benchmarks/shootout/mandelbrot.chapel)

```
    config const n = 200,           // the problem size
                 maxIter = 50,       // the maximum # of iterations
                 limit = 4.0,        // the limit before quitting
                 chunkSize = 1;      // the chunk size of the dynamic iterator
```

The Chapel compiler will automatically add command line arguments to change these values. You can either pass `--<cfgvar>=<val>`, or `-s<cfgvar>=<val>` (without a space after the s). If the value is a string with spaces, you must wrap it in quotes.

Try this. The four lines of code are placed in a file and compiled, then the program is run.

```
    config param cfgparam : int = 1;
    config const cfgconst : int = 2;
    config const cfgvar : real = 2.178;
    writeln("cfgparam = ", cfgparam, "   cfgconst = ", cfgconst,
            "   cfgvar = ", cfgvar);
```

```
    chpl -o ex_config ex_config.chpl
    ./ex_config
    > cfgparam = 1   cfgconst = 2   cfgvar = 2.178

    ./ex_config --cfgvar=3
    > cfgparam = 1   cfgconst = 2   cfgvar = 3.0

    ./ex_config -scfgvar=3 --cfgconst=1
    > cfgparam = 1   cfgconst = 1   cfgvar = 3.0
```

The value on the command line must be valid for the type of the variable/constant.

```
    ./ex_config --cfgvar=3 --cfgconst=1.7
    > <command line setting of 'cfgconst'>: error: Unexpected character when
      converting from string to int(64): '.'

    ./ex_config --cfgvar=true
    > <command line setting of 'cfgvar'>: error: Unexpected character when
      converting from string to real(64): 't'
```

To set a parameter during compilation you must use the `-s` form. Parameters cannot be changed from the command line during execution.

```
    ./ex_config --cfgvar=3 --cfgconst=1 --cfgparam=2
    > <command-line arg>:3: error: Unexpected flag: "--cfgparam=2"

    chpl --cfgparam=9 -o ex_config2 ex_config.chpl
    > Unrecognized flag: '--cfgparam=9' (use '-h' for help)

    chpl -scfgparam=9 -o ex_config2 ex_config.chpl
    ./ex_config2
    > cfgparam = 9   cfgconst = 2   cfgvar = 2.178
```

`-h` or `--help` print a table listing all the options that are available.

```
    ./ex_config2 -h
> FLAGS:
> ======
>   -h, --help            : print this message
>   -a, --about           : print compilation information
>   -nl <n>               : run program using n locales
>                           (equivalent to setting the numLocales config const)
>   -q, --quiet           : run program in quiet mode
>   -v, --verbose         : run program in verbose mode
>   -b, --blockreport     : report location of blocked threads on SIGINT
>   -t, --taskreport      : report list of pending and executing tasks on SIGINT
>   --gdb                 : run program in gdb
>   -E<envVar>=<val>      : set the value of an environment variable
>
> CONFIG VAR FLAGS:
> =================
>   -s, --<cfgVar>=<val>  : set the value of a config var
>   -f<filename>         : read in a file of config var assignments
>
> CONFIG VARS:
> ============
> Built-in config vars:
>        printModuleInitOrder: bool
>       dataParTasksPerLocale: int(64)
>   dataParIgnoreRunningTasks: bool
>       dataParMinGranularity: int(64)
>                    memTrack: bool
>                    memStats: bool
>                    memLeaks: bool
>               memLeaksTable: bool
>                      memMax: uint(64)
>                memThreshold: uint(64)
>                      memLog: c_string
>                 memLeaksLog: c_string
>                  numLocales: int(64)
>
> config config vars:
>                    cfgconst: int(64)
>                      cfgvar: real(64)
```

This example is in ex_config.chpl. Compile and run it with

```
    chpl -o bin/ex_config ex_config.chpl
or  make ex_config

    bin/ex_config
```

There is one quirk of the initialization of `config` variables if we put more than one on the same line. Initialization proceeds from left to right along the list, with the variable getting the value of the previous. It seems to go the other way, but the type and default value move to the first free variable in the list and are then copied to the right. If we provide a value for only one from the command line, then all others to its right will also get that value. In this example setting x also changes y, but y does not change x. z, listed on a separate line in the source, only changes when listed on the command line.

```
    config const x, y = -1;
    config const z = -1;
    writeln("x = ", x, "   y = ", y, "   z = ", z);

    ./ex_init
    > x = -1   y = -1   z = -1

    ./ex_init --x=3
    > x = 3   y = 3   z = -1

    ./ex_init --y=3
    > x = -1   y = 3   z = -1

    ./ex_init --z=3
    > x = -1   y = -1   z = 3
```

In short, `config` variables with initialization conditions should be put on separate lines. You'll find the example in ex_init.chpl. Compile and run it with

```
    chpl -o bin/ex_init ex_init.chpl
or  make ex_init

    bin/ex_init
```

Function Prototypes

One more piece we'll need for this exercise is how to declare the C-side functions for Chapel. A Chapel function declaration looks like:

```
   proc fnname(arg1 : type1, arg2 : type2) : returntype
```

and has a number of options for defining and using it..

1. You can call procedures with the arguments named:

```
    proc tstfn1(arg1 : int, arg2 : real) {
      writeln("tstfn1:  arg1 = ", x, "   arg2 = ", y);
    }

    tstfn1(arg2=3, arg1=1);
    > tstfn1:  arg1 = 1   arg2 = 3
```

2. You can provide default values for the arguments:

```
    proc tstfn2(arg1 : real = 2.718, arg2 : string = "dummy string") {
      writeln("tstfn2:  arg1 = ", x, "   arg2 = ", y);
    }

    tstfn2();
    > tstfn2:  arg1 = 2.718   arg2 = "dummy string"

    tstfn2(3.14);
    > tstfn2:  arg1 = 3.14   arg2 = "dummy string"

    tstfn2(arg2="a new string")
    > tstfn2:  arg1 = 2.718   arg2 = "a new string"

    tstfn2(arg1=1.41, "another string");
    > tstfn2:  arg1 = 1.41   arg2 = "another string"

    tstfn2("another string", arg1=1.41);
    > tstfn2:  arg1 = 1.41   arg2 = "another string"
```

In the first output line we see the default values for both arguments. In the second we call by position, which means supplying a value for `arg1` and using the default for `arg2`. To use the default for `arg1` we need to name `arg2` in the third call. The last two uses show that it's possible to swap the order of a named and positional argument with Chapel evaluating the named argument as if it were passed at its position in the argument list. `arg1` in the last call pushes the unnamed argument to the second position, which aligns with `arg2`.

3. You can omit the parentheses if there are no arguments to the procedure, or just put the parentheses with nothing between them. You must call the procedure with or without parentheses as it was defined.

```
    proc tstfn3a { writeln("tstfn3a:  no args"); }
    proc tstfn3b() { writeln("tstfn3b:  no args"); }

    tstfn3a;        /* tstfn3a() is illegal */
    > tstfn3a:  no args
```

```
tstfn3b();      /* tstfn3b is illegal */
> tstfn3b:  no args
```

4. An argument can be qualified by an "intent" which governs how the argument is passed to/from the procedure.

- An `in` argument receives a copy of the value from the caller. Any changes within the procedure are kept local.
- An `out` argument means the value on call is ignored, but that the caller's value is changed when the procedure returns. The variable will initialize to its default value. It may be referred to within the procedure.
- An `inout` argument gets a copy of the value in the caller when the procedure is called, and the caller's value is changed on return.
- A `ref` intent means the argument is passed by reference. Any access is done at the caller's site. Unlike an `inout` argument, no copying is done during the call or return, which means this may not be safe when changes happen in parallel.
- A `const in` intent means the argument receives a copy of the value from the caller and that the argument may not be changed locally.
- A `const ref` intent is like a `ref`, but there may be no changes made locally. If one procedure is working on a `const ref` argument and another on just a `ref`, then changes made by the second may still be seen by the first if the two are run concurrently.

```
proc tstfn4(in arg1 : int, out arg2 : real, inout arg3 : int,
            ref arg4 : real, const in arg5 : int,
            const ref arg6 : real ) {
  arg1 = arg1 - 1;
  arg2 = arg1 + 1;   /* ie. the original value of arg1, the -1 +1 cancel. */
  arg3 = arg3 + arg5 - 2;
  arg4 = arg6 + 1;
  /* Either of these two would cause a compilation error.
     arg5 = arg1 - 1;
     arg6 = 2 * arg2;
   */
}

var call1 = 3;
var call2 = 1.618;
var call3 = 7;
var call4 = 29.98e-9;
var call5 = 5;
var call6 = 9.807;

tstfn4(call1, call2, call3, call4, call5, call6);
writeln("tstfn4:  call1 = ", call1, "   call2 = ", call2, "   call3 = ",
        call3);
writeln(          call4 = ", call4, "   call5 = ", call5, "   call6 = ",
        call6);
> tstfn4:  arg1 = 3   arg2 = 3.0   arg3 = 10
>          arg4 = 10.807   arg5 = 5   arg6 = 9.807
```

It would be a compilation error if you tried to pass a `const` value to an `out`, `inout`, or `ref` argument (`const ref` is OK).

```
/* These are compilation errors.
  const call7 = 3.0;
  const call8 = 3;
  tstfn4(call1, call7, call3, call4, call5, call6);
  tstfn4(call1, call2, call8, call4, call5, call6);
  tstfn4(call1, call2, call3, call7, call5, call6);
*/
```

The default intent is `const in` except for synchronization variables and arrays, where it is `ref`.

5. To return a value from the procedure, use a `return` with an expression. If you specify a return type for the procedure (as in our introductory example), all return statements must produce either a value of that type or one that can be implicitly cast to it. If you do not specify a type, then all return statements are examined. There must be an implicit cast of all of them to one type, which becomes the return type of the procedure. If there is no `return`, the type is void. (So the procedures in the examples above all return void.)

```
proc tstfn5(arg1 : int) {
  if (arg1 < 5) {
    return 5;            /* an int */
  } else if (arg1 < 10) {
    return 6.0;          /* a real */
  } else {
    return 1.0 + 1.0i;   /* a complex number, the return type */
  }
}

var ret5 = tstfn5(3);
writeln("tstfn5:  real part ", ret5.re, "   complex ", ret5.im);
> tstfn5:  real part 5.0   complex part 0.0

ret5 = tstfn5(7);
writeln("tstfn5:  real part ", ret5.re, "   complex ", ret5.im);
> tstfn5:  real part 6.0   complex part 0.0

ret5 = tstfn5(11);
writeln("tstfn5:  real part ", ret5.re, "   complex ", ret5.im);
> tstfn5:  real part 1.0   complex part 1.0
```

Return types may also get intents. There are four. The `ref` intent means the procedure returns a reference to a variable. When such a procedure call is placed on the left side of an equals sign, or used as an `(in)out` or `ref` argument in another call, then the referred variable's value changes. When it is used on the right side, the procedure returns the existing value of the variable. The `param` intent evaluates at compile time and must create one of the simple expressions acceptable for a parameter. The `type` intent means the procedure will return a type such as `uint(8)`. This can be used for generic programming. The procedure will only be evaluated during compilation. The throws intent signals the procedure can create an error/exception.

6. Procedures may be nested inside others, with the inner only visible within the outer's scope. Nested functions may refer to variables in the outer procedure. Names in the inner procedure will shadow the outer.

```
proc tstfn6(arg1 : int) {
  const val1 = 3.14;
  proc tstfn6b(arg1 : real) {
    writeln("tstfn6:  arg1 = ", arg1, "   val1 = ", val1);
  }
  tstfn6b(arg1 + val1);
}

tstfn6(3);
> tstfn6:  arg1 = 6.14   val1 = 3.14
```

7. You can precede the proc with the keyword inline. The procedure will be inlined at every call site.

8. Procedures, including many operators, may be overloaded; the best match to the arguments in the caller will be used. Operators keep their precedence and arity, the number of arguments they take. The language specification contains a table with the list of operators that may be overloaded.

9. You can have a variable number of arguments, indicated with three dots. This, combined with generic procedures and type arguments, allows a level of meta-programming which we'll look at shortly

You cannot combine variables with the same type in the declaration.

```
proc tstfn(arg1, arg2 : int) { }
```

is not allowed, you have to put the type after each argument.

The code samples for this section are in ex_fn.chpl. Compile and run them with

```
    chpl -o bin/ex_fn.chpl
or  make ex_fn

    bin/ex_fn
```

External Linkage (rw_png_v1.chpl)

Now the pieces are in place. The general outline of this first program will be:

- define command-line arguments (file names, pixel coordinates)
- define internal variables
- define the C interface
- read the PNG image, get the pixel value and print it, change it, and write the result back

To access things C-side, we need to declare them in the Chapel program and provide the C header and object files when we compile. The declarations follow the variable and function formats we've already described. We only need to put the `extern` keyword before them.

The five C prototypes we need are:

```
    int PNG_read(const char *inname, rgbimage **img);
    int PNG_write(const char *outname, rgbimage *img);
    int read_rgb(rgbimage *img, int x, int y, uchar *r, uchar *g, uchar *b);
    int write_rgb(rgbimage *img, int x, int y, uchar r, uchar g, uchar b);
    void free_rgbimage(rgbimage **);
```

(where `uchar` is a typedef for `unsigned char`). Let's start with `write_rgb()` as it's the easiest.

If the C program uses the standard types that encode the bit size in their name (`int32_t`) then it's safe to use the corresponding Chapel type `int(32)` directly. Otherwise the recommendation is to use the types `c_<typename>` (`c_int`, `c_uint`, `c_char`, `c_uchar`) because there might be a size mismatch. These types are aliases for the appropriate Chapel type and can be used freely in their place.

So, ignoring the pointer for a moment, the `write_png()` prototype becomes

```
    extern proc write_rgb( ???, x : c_int, y : c_int,
                           r : c_uchar, g : c_uchar, b : c_uchar) : c_int;
```

How about the image pointer? First we need to handle the type. If all we're doing is passing around a pointer between C functions, without needing to access the data from Chapel, then we can define an empty type alias that tells Chapel to only handle the pointer. The data structure behind the pointer is "opaque", and can only be assigned to other opaque variables of the same type or used as an argument to a C function. That's all we need for now, so the command for the type alias is

```
    extern type rgbimage;
```

For the pointer, there are a couple ways to approach it. There is a type `c_ptr(<type>)`, which can also be constructed from a Chapel variable by calling `c_ptrTo(<variable>)`, where it gets the type from the variable. Or we can use a `ref` intent on the argument, which effectively means a pointer will be used. Since we don't have a variable with the `rgbimage`, only a pointer to a blob of that type, we'll use the `c_ptr()` option. The prototype becomes

```
    extern proc write_rgb(img : c_ptr(rgbimage), x : c_int, y : c_int,
                           r : c_uchar, g : c_uchar, b : c_uchar) : c_int;
```

How to declare the `PNG_write()` function is now clear. Chapel defines the `c_string` type as an alias for `const char *`, which is the first argument, and the same type of pointer as we just talked about for the second. string and c_string do not have the same internal representation and cannot be substituted. We must convert the Chapel value to a null-terminated array when we call the function. The prototype is

```
    extern proc PNG_write(fname : c_string, img : c_ptr(rgbimage)) : c_int;
```

For `read_rgb()`, we know how to represent the first three arguments. What about the pointers to `uchar` for the RGB triple, where we are returning the values to the variable? This does sound like we want the `ref` intent for the pointer. First we'll need three variables to hold the values, and then declare the corresponding arguments as refs.

```
    var rpix, gpix, bpix : c_uchar;
    extern proc read_rgb(img : c_ptr(rgbimage), x : c_int, y : c_int,
                         ref r : c_uchar, ref g : c_uchar, ref b : c_uchar);
```

The final two prototypes require a double pointer to receive the pointer to the memory with the allocated image. This must be set to `NULL` initially or the PNG functions will assume there's an image already there and will try to free it. The equivalent to this in Chapel is to declare a pointer to the type as a variable and to pass that variable as a `ref` argument, ie. a pointer to the variable which is a pointer. When we declare the variable we want to initialize it to `nil`. Only classes can be assigned `nil`, which is also their default value; this is a placeholder until the class is actually instantiated. In this next initialization, then, we're foreshadowing that `c_ptr` is implemented as a class, as we'll soon see.

```
    var rgb : c_ptr(rgbimage) = nil;
    extern proc PNG_read(fname : c_string, ref img : c_ptr(rgbimage)) : c_int;
    extern proc free_rgbimage(ref img : c_ptr(rgbimage)) : void;
```

This defines the five prototypes for the C interface, and four local variables that we need for arguments modified by the functions.

Following the command line for test_png, we will want four options:

```
   config const inname : string;
   config const outname : string;
   config const x, y : c_int;
```

`config` means we can supply the value on the command line (and must, because this version of the program will crash if we don't provide them). We don't change the value so `const` is appropriate. We can use the `c_int` type here because it is an alias for `int(32)` on our machines, but must use Chapel's strings. To convert to `c_string` we use the `c_str()` method function, as in C++.

The last thing that's left is the logic: calling the five functions and printing the RGB values we've retrieved.

```
    PNG_read(inname.c_str(), rgb);
    read_rgb(rgb, x,y, rpix, gpix, bpix);
    writef("\nAt %4i,%4i   R %3u  G %3u  B %3u\n", x,y, rpix,gpix,bpix);
    write_rgb(rgb, x,y, 1, 2, 3);
    PNG_write(outname.c_str(), rgb);
    free_rgbimage(rgb);
```

The `writef()` statement is the equivalent of C's `printf()` where i is the format code for an `int` and u a `uint`. Note that we're ignoring the return value of the functions for now. If something goes wrong, it will go horribly wrong. Try running the program without any command line options and you'll get a segmentation fault because we ignore the error code that `PNG_read()` will return (`inname` is an empty string, so it can't read that file) and when `read_rgb()` runs it crashes because the `rgb` variable is nil.

To compile the program we need in addition to the program (rw_png_v1.chpl) the header file for the C code (img_png.h), the object file (build/img_png.o), and a link to the PNG library. Chapel parses the header file to set up bindings to the C data and functions and the linker will add in our object file and the library. To compile and run the program use

```
    chpl -o bin/rw_png_v1 rw_png_v1.chpl img_png_v1.h build/img_png_v1.o -lpng
or  make rw_png_v1
```

```
    bin/rw_png_v1 --inname=bear.png --outname=tstimg.png --x=615 --y=212
    > At  615, 212   R  83  G 156  B  25
```

You can inspect tstimg.png, or compare it to the test_png output to see that it's the same.

Aside: Embedding C Requirements (rw_png_v1b.chpl)

In the October 2015 release Chapel added support for embedding C file requirements inside the Chapel source file instead of on the compiler command line. The declaration in the source file is a comma-separated list of strings following the `require` keyword that specify header, C source, and object files or libraries. Header files are included, source files are compiled as Chapel normally would, and object files and libraries are linked. For this program we can write

```
    require "img_png_v1.h", "build/img_png_v1.o", "-lpng";
```

which we used on the compiler command above. Compilation is then just

```
    chpl -o bin/rw_png_v1b rw_png_v1b.chpl
or  make rw_png_v1b
```

Whether you find it better to track dependencies in the source or make file is a question of style. We'll keep listing them in the compilation command because we need to list the dependencies for the make target anyway.

Structural Types (rw_png_v2.chpl)

The `rgbimage` type in _v1 was opaque. Chapel has no idea what's in it and only passes a pointer around. Can we gain access to the fields of the structure? To do that we'll need a structural type.

Chapel has three structural types: class, record, union. Classes and records are very similar, except classes are handled by reference and records by value. We'll see the difference clearly when we modify img_png_v1.c and rw_png_v1.chpl so we can directly access the members of the `rgbimage` struct. All three allow variables, types, and procedures to be declared inside. Variables within these are called 'members' or 'fields' and procedures 'methods'. Using the terminology from object-oriented programming is deliberate, as classes and records also support inheritance. These fields are accessed by dot notation, or `instance.method()`. (The language spec describes a fourth type, tuples, as structural, but they have a different, simplified, layout, without methods or named members. We'll use tuples in the next exercise.)

The examples for this section are in ex_struct.chpl. Compile and run it with

```
    chpl -o bin/ex_struct ex_struct.chpl
or  make ex_struct.chpl

    bin/ex_struct
```

Unions are the simplest of the three. They are declared with the `union` keyword, possibly preceded by `extern`, followed by a name and then any declarations within braces.

```
   union numbers {
     var svar : int(16);
     var ivar : int(32);
     var lvar : int(64);
     var rvar : real;
```

```
    var cvar : complex;
  }
```

Unions can only contain a value in one field at a time. Writing to another field will unset the first. As an example, let's assume we have two variables of this union type:

```
  var exunion, cpunion : numbers;
  exunion.ivar = 1234;
  writef("assigned union int %i\n", exunion.ivar);
  > assigned union int 1234
```

If we assign a union variable to another, then the active field's value is copied and that field is set active.

```
  cpunion = exunion;
  writef("copy has int field %i\n", cpunion.ivar);
  > copy has int field 1234
```

This copy has been made by value; there is no link between `cpunion` and `exunion` at this point. Changing the active field in `exunion` will not affect `cpunion`.

```
  exunion.cvar = 1.0 + 2.0i;
  writef("changed union to complex %z\n", exunion.cvar);
  writef("copied union still int   %i\n", cpunion.ivar);
  > changed union to complex 1 + 2i
  > copied union still int   1234
```

It is a runtime error to access a field that has not been set.

```
  writef("union now has no int     %i\n", exunion.ivar);
  > ex_struct.chpl:36: error: halt reached - illegal union access
```

When a union is initialized no field is set. The variable declarations inside should not be given initialization conditions. Although the compiler will not complain, you will not be able to access an initialized field (the runtime will give an illegal union access error), as if the initialization does not set it active. You will still need to assign something to one field before accessing it.

Records and classes are declared the same way, except for the keyword and the possibility of specifying a superclass for a class. Records and classes differ in a few ways, the main ones being how they are initialized and assigned. If we think of a class as a pointer to a structure, and a record as a structure, then we understand many of the differences. A pointer can take the `NULL` value so variables of a class type can take `nil` while records cannot. A class must be manually constructed and destroyed while storage for records is known and allocated when the variable is declared and is reclaimed when it leaves scope. Class variables are assigned by copying a pointer and so the left hand side refers to the right after the assignment, while records are copied by value and are independent afterward.

Let's look at some examples to see these points.

A record implementation of the rgbimage struct in img_png.h might look like

```
    record rgbimage_rcd {
      var ncol : c_int;
      var nrow : c_int;
```

```
  var npix : c_int;
  var r : c_ptr(c_uchar) = nil;
  var g : c_ptr(c_uchar) = nil;
  var b : c_ptr(c_uchar) = nil;
}
```

We're using the C types (see the previous section) to lay the groundwork for the new version of the rw_png program we'll be making. For each element of the structure, Chapel automatically makes a single method with the name of the member. It acts as both a getter and setter. Do you remember from the procedure discussion that you can omit the parentheses after a procedure if it takes no argument, and that any call must also omit them? That's essentially what's happening here. The method is given a `ref` return intent so that if it's placed on the left side of an assignment statement the member is modified, and on the right it returns the value. In other words, this record implictly includes

```
proc ncol : c_int ref { return ncol };
```

With one procedure we have both the getter and the setter.

When we declare a variable as an instance of this record, it is initialized member by member following the normal rules, ie. if an initialization expression has not been provided, then the type's default value is used. We can also assign a record to a variable or use it in an expression by using the keyword `new` before the record name, which allows us to access its default constructor. The compiler automatically generates the constructor, where the arguments are provided in the same order as the fields and given their names.

```
var exrcd = new rgbimage_rcd(ncol=300, nrow=400);        /* default npix=0 */
var cprcd : rgbimage_rcd;                      /* default ncol=0, nrow=0, npix=0 */
```

To set a field, simply assign it a value.

```
cprcd.npix = 100;
writef("exrcd: %4i x %4i (= %6i npix)   nil r? %s\n",
       exrcd.ncol,exrcd.nrow, exrcd.npix, (nil == exrcd.r));
writef("cprcd: %4i x %4i (= %6i npix)   nil r? %s\n",
       cprcd.ncol,cprcd.nrow, cprcd.npix, (nil == cprcd.r));
> exrcd:  300 x  400 (=      0 pix)   nil r? true
> cprcd:    0 x    0 (=    100 pix)   nil r? true
```

The compiler will generate default =, ==, and != methods. Each is applied member by member with the usual behavior. Because records copy by value, the assignee is kept separate from the assigner.

```
writef("ex == cp? %s\n", (exrcd == cprcd));
> ex == cp? false

cprcd = exrcd;
writeln("copy exrcd to cprcd");
writef("exrcd: %4i x %4i (= %6i npix)   nil r? %s\n",
       exrcd.ncol,exrcd.nrow, exrcd.npix, (nil == exrcd.r));
writef("cprcd: %4i x %4i (= %6i npix)   nil r? %s   == exrcd? %s\n",
       cprcd.ncol,cprcd.nrow, cprcd.npix, (nil == cprcd.r),
       (exrcd == cprcd));
> copy exrcd to cprcd
> exrcd:  300 x  400 (=     0 pix)   nil r? true
> cprcd:  300 x  400 (=     0 pix)   nil r? true   == exrcd? true

exrcd.ncol = 600;
exrcd.nrow = 800;
writeln("set exrcd ncol, nrow");
writef("exrcd: %4i x %4i  (= %6i npix)   nil r? %s\n",
       exrcd.ncol,exrcd.nrow, exrcd.npix, (nil == exrcd.r));
writef("cprcd: %4i x %4i  (= %6i npix)   nil r? %s   == exrcd? %s\n",
       cprcd.ncol,cprcd.nrow, cprcd.npix, (nil == cprcd.r),
       (exrcd == cprcd));
> set exrcd ncol, nrow
> exrcd:  600 x  800 (=     0 pix)   nil r? true
> cprcd:  300 x  400 (=     0 pix)   nil r? true   == exrcd? false
```

The fields in `cprcd` are overwritten during the assignment but are then independent of changes to `exrcd`.

Fields are only accessible through an instance of a record or class. Identifiers declared `param` will not be visible.

Our class example will run the same with two differences. First, we can't simply declare a variable as a class and have it appear; we must make an instance with the `new` keyword. The comments about constructors generated by the compiler or the program still apply.

```
class rgbimage_cls {
  var ncol : c_int;
  var nrow : c_int;
  var npix : c_int;
  var r : c_ptr(c_uchar) = nil;
  var g : c_ptr(c_uchar) = nil;
  var b : c_ptr(c_uchar) = nil;
}

var excls : rgbimg_cls;
writef("excls nil post-declaration? %s\n", (nil == excls));
> excls nil post-declaration? true

excls = new rgbimg_cls(200, npix=1000, ncol=5);
writef("new excls:     %4i x %4i  (= %6i npix)   nil r? %s\n",
       excls.ncol,excls.nrow, excls.npix, (nil == excls.r));
> new excls:        5 x  200 (=   1000 pix)   nil r? true

excls.ncol = 10;
excls.npix = 2000;
writef("excls changed: %4i x %4i  (= %6i npix)   nil r? %s\n",
       excls.ncol,excls.nrow, excls.npix, (nil == excls.r));
> excls changed:   10 x  200 (=   2000 pix)   nil r? true
```

The example variable defaults to `nil` until we create a new instance of the class. Accessing and changing member values happens the same way. We can of course instantiate the class when we declare the variable, but watch what happens after we create the copy.

```
var cpcls = excls;
writef("make cpcls:    %4i x %4i (= %6i npix)   nil r? %s   == excls? %s\n",
       cpcls.ncol,cpcls.nrow, cpcls.npix, (nil == cpcls.r),
       (cpcls == excls));
> make cpcls:      10 x  200 (=   2000 pix)   nil r? true   == excls? true

excls.ncol = 20;
excls.npix = 4000;
writef("excls changed: %4i x %4i (= %6i npix)   nil r? %s\n",
       excls.ncol,excls.nrow, excls.npix, (nil == excls.r));
writef("cpcls now:     %4i x %4i (= %6i npix)   nil r? %s   == excls? %s\n",
       cpcls.ncol,cpcls.nrow, cpcls.npix, (nil == cpcls.r),
       (cpcls == excls));
> excls changed:   10 x  400 (=   4000 pix)   nil r? true
> cpcls now:       10 x  400 (=   4000 pix)   nil r? true   == excls? true
```

The change to `excls' ncol` and `npix` appears in `cpcls` because both variables point to the same underlying memory.

(By the way, the reason for always testing if the `r` field is `nil` is because the Chapel write routines don't understand how to print a `c_ptr`. The compilation will fail.)

You can define constructors and destructors for both records and classes. Constructors have the name of the structure and take any arguments that you want, which are supplied when instantiating with `new`. Destructors are named `deinit()` and take no arguments. They're called before Chapel releases the instance's memory.

```
class image_cls {
  var ncol, nrow, npix : int;

  proc image_cls(numcol : int, numrow : int) {
    ncol = numcol;
    nrow = numrow;
    npix = numcol * numrow;
  }

  proc deinit() {
    writeln("freeing allocated image");
  }
}

var eximg = new image_cls(10, 20);
writef("image size %4i x %4i (= %6i npix)\n",
       eximg.ncol,eximg.nrow, eximg.npix);
> image size   10 x   10 (=    200 npix)
```

Be aware that there is one critical difference between records and classes. Records are automatically created and destroyed, classes are handled manually. You can use `new` with a record to access a constructor; if you provide a custom constructor the default version will not be available. Without using `new` a record's fields are initialized to their default per their type. When a record variable leaves scope its destructor, default or custom, executes – you cannot do this yourself. On the other hand, a class variable is `nil` when declared and you must use `new` to create an instance. You must use delete to call the destructor, and if you do not you will leak memory.

```
    class test_cls {
      var tmp : int;

      proc ~test_cls() {
        writeln("  called class destructor, tmp was ", tmp);
      }
    }
    record test_rcd {
      var tmp : int;

      proc ~test_rcd() {
        writeln("  called record destructor, tmp was ", tmp);
      }
    }

    proc test_destruct1() {
      var c = new test_cls(3);
      var r = new test_rcd(2);

      writeln("in test_destruct1:");
      /* This leaks c. */
    }

    proc test_destruct2() {
      var c = new test_cls(2);
      var r = new test_rcd(3);

      writeln("in test_destruct2:");
      delete c;
    }

    test_destruct1();
    > in test_destruct1:
    >   called record destructor, tmp was 2
    test_destruct2();
    > in test_destruct2:
    >   called class destructor, tmp was 2
    >   called record destructor, tmp was 3
```

The examples for records and classes can be compiled and run with

```
    chpl -o bin/ex_struct ex_struct.chpl
or  make ex_struct
```

```
    bin/ex_struct
```

Just as procedure arguments and return values have intents, methods have an optional qualifier. A `type` intent makes a static method.

```
    class num {
      var v : int;

      proc type dump(msg : string) {
        writeln(msg);
      }
    }

    num.dump("calling without an instance")
    > calling without an instance
```

Such a method cannot be called on an instance of the class.

```
    var n = new num(3);
    n.dump("or via an instance");
    > ex_method.chpl:18 error: unresolved call 'num.dump("or via an
      instance")'
    > ex_method.chpl: 5: note: candidates are: num.type dump(msg:string)
```

A ref intent means the argument, an implicit this or the instance itself, is passed-by-reference and can be changed. In this example we replace the existing instance with a completely new one. The %t format specifier prints an object-dependent description. Note that you can declare a method outside its class by fully qualifying its name.

```
    proc ref num.changeInst() {
      this = new num(5);
    }

    var oldn = n;
    writef("original instance %t (same as old? %s)\n", n, (n == oldn));
    > original instance {v = 3} (same as old? true)

    n.changeInst();
    writef("new instance %t (same as old? %s)\n", n, (n == oldn));
    > new instance {v = 5} (same as old? false)
```

There is a third method intent, param, that means it can only be applied to a compile-time parameter. You can define such methods on custom classes as we have done here, or the built-in types.

Compile and run the examples for method intents with

```
    chpl -o bin/ex_method ex_method.chpl
or  make ex_method

    bin/ex_method
```

So how do we apply this to our example? We have a problem that `rgbimage` in img_png.h was essentially defined as a record, but we always use a pointer to it. We also allocate memory and have the problem that Chapel does not necessarily use the C standard library (it uses jemalloc in distributed systems) and cannot manage the arrays. Instead of a record we really want to work with a class, where we control the object's life cycle. This requires changing the C-side structure to

```
typedef struct __rgbimage {
  int ncol, nrow;
  int npix;
  uchar *r, *b, *g;
} _rgbimage, *rgbimage;
```

This pattern:

```
typedef struct __D {
} _D *D;
```

is required by the Chapel compiler when it parses the header file. In other words, the structure name (`D`) must be a pointer to a structure whose type alias is the same except for a leading underscore (`_D`) and whose name has two leading underscores (`__D`). The edit can be found in img_png_v2.h.

Only the class members that match the C structure elements are linked. The Chapel class can leave some out, in which case they aren't accessible. The class can also add members, which have no effect on the C elements. This also is true for methods.

To access the members in Chapel we delete the empty type definition for `rgbimage` ('`extern type rgbimage;`') and declare an `extern` class:

```
extern class rgbimage {
  var ncol : c_int;
  var nrow : c_int;
  var npix : c_int;
  var r : c_ptr(c_uchar);
  var g : c_ptr(c_uchar);
  var b : c_ptr(c_uchar);
}
```

We'll use a variable to this class, but not instantiate it because the memory allocation is done C-side in `PNG_read()`. The variable defaults to nil when declared, which is the value we need for the first C allocation. When done we must call `free_rgbimage()` with the variable as the argument to release the memory and again set the Chapel value to nil. We do not use `new` and `delete` with this class. Effectively the variable is a pointer to data that Chapel knows now knows how to access but does not control.

Now, how to access the image arrays without going through `read_rgb()` or `write_rgb()`? If you look in CHPL_HOME/modules/standard/SysBasic.chpl, you'll find the definitions for the C equivalent types. `c_ptr()` is in CHPL_HOME/modules/internal/CPtr.chpl. It turns out this is a class with three member functions:

```
class c_ptr {
  /* The type that this pointer points to */
```

```
      type eltTye;
      /* Retrieve the i'th element (zero based) from a pointer to an array.
         Does the equivalent of ptr[i] in C.
      */
      inline proc this(i: integral) ref {
        return __primitive("array_get", this, i);
      }
      /* Get element pointed to directly by this pointer.  If the pointer
         refers to an array, this will return ptr[0].
      */
      inline proc deref() ref {
        return __primitive("array_get", this, 0);
      }
      /* Print this pointer */
      inline proc writeThis(ch) {
        (this:c_void_ptr).writeThis(ch);
      }
    }
```

Classes support a method `this` in addition to the member `this` (which appears in the call to `__primitive()`) which allows us to treat the instance as a procedure. In other words, we can write the instance variable followed by arguments inside parentheses, just like a procedure call. Here the method has a `ref` return intent, so it acts as both a getter and a setter for the an array element using the internal Chapel function `array_get`. If `rgb` is an instance of our `rgbimage` class, then

```
    rgb.r(xy)
```

will return the pixel value at index xy and

```
    rgb.r(xy) = newval;
```

will change the value.

We can now replace our call to `read_rgb()`

```
    read_rgb(rgb, x, y, rpix, gpix, bpix);
```

with

```
    const xy = (y * rgb.ncol) + x;
    rpix = rgb.r(xy);
    gpix = rgb.g(xy);
    bpix = rgb.b(xy);
```

xy here is just the conversion we've seen from a 2D point to a 1D array index.

Similarly, the call to `write_rgb()`

```
    write_rgb(rgb, x, y, 1, 2, 3);
```

becomes

```
    const xy = (y * rgb.ncol) + x;
    rgb.r(xy) = 1;
    rgb.g(xy) = 2;
    rgb.b(xy) = 3;
```

The new version of the program can be found in rw_png_v2.chpl. Compile and run it with

```
    chpl -o bin/rw_png_v2 rw_png_v2.chpl img_png_v2.h build/img_png_v2.o -lpng
or  make rw_png_v2

    bin/rw_png_v2 --inname=bear.png --outname=tstimg.png --x=615 --y=212
```

The output will be the same as img_png_v1 and test_png except you will also see the size of the image, showing that we can directly access the image data.

Modules (rw_png_v3.chpl)

If we forget to pass arguments on the command line, or cause an error by giving the wrong input file name for example, the program will crash. We should at least fail gracefully and clean up after ourselves. (Our policy here is to free any memory we've allocated, even after an error, which helps with memory leak checks). Improving this will introduce us to Chapel's module system.

Modules are a simple namespace. They can declared implicitly or explicitly. An explicit declaration contains the keyword `module` followed by a name and a set of statements wrapped in braces:

```
    module example {
    }
```

Any symbols that are defined within the braces are scoped to the module. An implicit declaration uses the name of the file, less the '.chpl' extension, as the module name. So the two programs we've looked at so far have actually defined two modules, rw_png_v1 and rw_png_v2, with all variables and procedures local to them. Note that if the file name does not form a legal identifier (alphanumeric plus underscore and dollar sign) then the module cannot be referenced.

There may be multiple modules in a file, and they may nest. In this case the inner modules see all symbols in the outer, but the inner symbols must be qualified with the module name when used in the outer scope.

```
    module outerexample {
      var outervar : int(64);
      /* Must access innvervar with innerexample.innervar. */
      module innerexample {
        var innervar : int(64);
        /* Can access outervar without qualifying the name. */
      }
    }
```

If there are symbols at the top file level outside a module, and a module is also defined, then that module is nested inside the intrinsic module.

Statements at the top level of a module are executed as the module is created. This is what's happening in our example programs. Chapel processes the implicit module for each file and executes the statements along the way. Because these statements aren't part of a procedure, some, like `return`, aren't legal. The compiler will raise an error.

To use a module, give the keyword `use` and the name.

```
    use example;
```

If only an inner module is needed, provide its fully qualified name.

```
    use outerexample.innerexample;    /* outerexample is not accessible. */
```

You do not need to qualify a symbol in that module, ex.

```
    innervar = 10;
```

But you can always provide a fully qualified name to access anything.

```
    outerexample.outervar = 11;
```

Symbols within modules, either variables or procedures, may be hidden from access outside the module by prefacing them with `private`. If `public` or no keyword is used, then they will be visible. If modules are nested then the inner can see the private symbols in the outer, but not the other way around. Note that you cannot use a fully qualified name to gain access. Modules themselves may also be declared public or private by putting the qualifier before the `module` keyword.

You'll see the private keyword in use later when we pull common code into library files, which will start with ip_. An example for this section is ip_color_v1.chpl.

You do not have to list files with modules in the compile command. They'll be found and pulled in automatically.

Chapel provides a large set of modules. In CHPL_HOME/modules/internal you'll find those used to implement the language, and in CHPL_HOME/modules/standard the standard libraries. CHPL_HOME/modules/packages are libraries that haven't yet matured. Each source file in the library has its documentation embedded in it, which is extracted into CHPL_HOME/doc/[rst|html]/modules.

Any Chapel program automatically uses four of the standard modules: Assert, IO, Math, and Types. You do not have to import these yourself. We will need Help for this example.

Our first task is to sanity-check the config constants. Since 0 is a valid pixel address (x and y are 0-based), we need to change the default initial value to something less than 0. We then test that the user has supplied a non-negative value on the command line. We will also check that there are input and output file names; the default initial value of an empty string will work fine. We will use the C function `PNG_isa()` to verify that the input file is a PNG file. Once the file has been read, we can then check that x and y are not too big. Because the values are 0-based, this means they must be less than the number of columns or rows.

The `if` statement has two forms. We can follow the conditional test with the keyword `then` and a single command, possibly followed by `else` and a single command. Or we can use braces for multiple commands. If we use the keyword form and nested `if`'s, the `else` will bind to the last `if`.

```
    const a = 1;
    const b = 2;
    const c = 1;
```

```
if (a == b) then
  writeln("a matches b");
writeln("going to next test");
if (a == c) then
  writeln("a matches c");
else
  writeln("a, c, differ");
writeln("tests done");
> going to next test
> a matches c
> tests done
```

```
if (a == b) {
  writeln("a matches b");
  writeln("going to next test");
}
if (a == c) {
  writeln("a matches c");
} else {
  writeln("a, c differ");
  writeln("tests done");
}
> a matches c
```

```
if (a == b) then
  writeln("a matches b");
else if (a != c) then
  writeln("a, c, differ");
  else
    writeln("a matches c");
> a matches c
```

The example is in ex_if.chpl. Compile and run it with

```
    chpl -o bin/ex_if
or  make ex_if

    bin/ex_if
```

What do we do if we have a bad value? The Help module provides one method, `printUsage()`, that creates the usual output for the `-h` argument to the executable. It prints a table with the standard command line arguments and configuration variables. It does not exit, however. We'll create our own usage procedure that prints an error message, then the normal help output, and then exits.

```
config const inname : string = "";
config const outname : string = "";
config const x : c_int = -1;
config const y : c_int = -1;

proc usage(msg : string) {
  writeln("\nERROR");
  writeln("  ", msg);
  printUsage();
  exit(1);
}

if (x < 0) then
  usage("missing --x or value < 0");
if (y < 0) then
  usage("missing --y or value < 0");
if ("" == inname) then
  usage("missing --inname");
if (!PNG_isa(inname)) then
  usage("input file not a PNG picture");
if ("" == outname) then
  usage("missing --outname");

PNG_read(inname, rgb);

if (rgb.ncol <= x) then
  usage("--x (0-based) >= image width " + rgb.ncol);
if (rgb.nrow <= y) then
  usage("--y (0-based) >= image height " + rgb.nrow);
```

Note that we're using the integer interpretation of a boolean, ie. zero is `false` and everything else is `true`, for the test on `PNG_isa()`. The + operator applied to string concatenates them. In the last two tests the `ncol`/`nrow` integer is converted to a string representation and combined with the error messages.

Chapel uses a `try...catch` mechanism. Errors are a class hierarchy rooted in Error (in the ChplError internal module) with specific error types in the SysError standard module. To raise an error use the `throw new <Errror>;` statement. The procedure must also have the `throws` intent. In the caller wrap the function call in `try { } catch { }`, where you can follow the `catch` with a type filter. For example, the IO module has two `check()` procedures, one that returns the error in an argument. (This is historical, for compatibility with code written before error handling was added to the language.)

```
    /* Throw an error if a file is invalid */
    proc file.check() throws {
      if is_c_nil(_file_internal) then
        throw SystemError.fromSyserr(EBADF,
                                    "Operation attempted on an invalid file"
    );

    /* Return a syserr through out error if a file is invalid */
    proc file.check(out error:syserr) {
      var err : syserr = ENOERR;
      try {
        check();
      } catch e : SystemError {
        err = e.err;
      } catch {
        err = EINVAL;
      }
      error = err;
    }
```

You can use `try!` instead of `try`. This causes Chapel to halt the program immediately. There is no final clause; Chapel instead provides a `defer` block to indicate which statements must execute when the block leaves scope. Use this for cleanup.

This way of handling errors doesn't really help us, though, with handling C codes. We could always wrap the C function and translate an error code into an Error, but we'll follow the C style and test the return value from the PNG functions and manually clean-up and halt if there's a problem. Since we won't use the C interface much, this is sufficient. For example,

```
  var retval : int;
  retval = PNG_write(outname, rgb);
  if (retval < 0) {
    free_rgbimage(rgb);
    halt("program stopped after error");
  }
```

The new version of the program can be found in rw_png_v3.chpl. Compile it with

```
    chpl -o bin/rw_png_v3 rw_png_v3.chpl img_png_v2.h build/img_png_v2.o -lpng
or  make rw_png_v3

    bin/rw_png_v3 --inname=bear.png --outname=tstimg.png --x=615 --y=212
```

Try different values for the arguments to see if they are handled correctly. To test the error handling, you can run the program once and turn off write permission to the output file before running it a second time, which will cause `PNG_write()` to fail.

Aside: Variadic Procedures (rw_png_v3b.chpl)

Although Chapel doesn't have macros, it does have generic procedures and compile-time parameters that let us cleanly handle errors. Consider the following,

```
  proc end_onerr(retval : int, inst ...?narg) : bool {
    /* Note we skip the argument if we don't know how to clean it up. */
    if (0 <= retval) then return false;
    for param i in 1..narg {
      /* Note we skip the argument if we don't know how to clean it up. */
      if (inst(i).type == rgbimage) then free_rgbimage(inst(i));
      else if isClass(inst(i)) then delete inst(i);
    }
    return true;
  }
```

This uses a lot of the language we haven't seen yet. The `...?narg` notation denotes a variadic argument list of mixed type (so this is a generic procedure). `narg` is the number of arguments in the list and is a parameter. The

`?` is a query that assigns a value to `narg` when the compiler knows how many arguments there are from the caller. `for param` is a parameter loop that is unrolled at compile time, with the body repeated for each value in the range `1..narg`. `inst(i)` accesses the `i`'th argument. `inst(i).type` is the type of the argument, which we test against the class name rgbimage. `isClass()` is a procedure in the standard module Types.chpl that returns true if the argument is an instance of a class. If the `i`'th argument is an instance, we delete it. Other types could be handled by adding clauses.

All this is done at compile time. If you wanted to limit the list to arguments of just one type, put the type name before the ellipsis.

```
    proc joinbyline(x : string...?)
```

You can also force the number of arguments by changing the '?' to a number. This might not seem useful – why not just list that many arguments? – but consider that the number could be generated from a parameter and therefore set at compile time.

Since the procedure cleans up and exits all we have to do is call it with the error code and data not managed by Chapel.

```
    retval = subroutine_returning_error_value();
    end_onerr(retval, objects_to_delete);
```

rw_png_v3b.chpl shows this procedure in use after the `PNG_read()` and `PNG_write()` calls. For simplicity we'll stay with the approach in _v3 for the rest of this exercise, but we'll be using `end_onerr()` in the other chapters. Compile the program with

```
    chpl -o bin/rw_png_v3b rw_png_v3b.chpl img_png_v2.h build/img_png_v2.o -lpng
or  make rw_png_v3b

    bin/rw_png_v3b --inname=bear.png --outname=tstimg.png --x=615 --y=212
```

Aside: Main / Program Organization (rw_png_v4.chpl)

One final small question to answer. Passing named options on the command line adds a good bit of typing. Is there a way to parse the command line? In C we get the options as strings passed to main:

```
    int main(int argc, char **argv) { .. }
```

but our programs don't have a `main()` function.

Well, they do, but it's an empty procedure automatically generated by the compiler.

```
    proc main() { }
```

Chapel starts a program by running all top-level statements in a module. Initialization begins in the module that defines `main()`, then those it uses (but only once, so there cannot be multiple includes) in the order given in the program. The example CHPL_HOME/examples/spec/Modules/init-order.chpl shows how this works:

```
    module M1 {
      use M2.M3;
      use M2;
      writeln("In M1's initializer");
```

```
  proc main() {
    writeln("In main");
  }
}
module M2 {
  use M4;
  writeln("In M2's initializer");
  module M3 {
    writeln("In M3's initializer");
  }
}
module M4 {
  writeln("In M4's initializer");
}
> In M4's initializer
> In M2's initializer
> In M3's initializer
> In M1's initializer
> In main
```

M1 contains the main function, and begins with the '`use M2.M3;`' statement. To initialize M3, we need to have initialized M2, which requires setting up M4. Therefore, M4 is first, then M2 finishes its initialization before M3. M2 has already been set up when we reach the '`use M2;`' statement in M1, so we skip it and finish M1. Finally the program starts executing main.

A program with multiple main procedures may fail to compile, depending on the implementation. The compiler option `--main-module <module>` lets you select one.

`main()` can have two return types, `void` or `int`. If your program does not define `main()`, the auto-generated version will return a `void` and the runtime will return 0 to the operating system. (In C this means the program ran without error; any non-zero integer is interpreted as an error code.) Any `return` statements at the top level must not have a value. If you do provide `main()`, you can leave off the return type and Chapel will infer it from the `return` statements. If an integer is returned at any point, then one must be returned everywhere, including at the end of the procedure. You cannot fall off the end without a `return` in this case.

Chapel does also support a main with one argument:

```
proc main(args: [] string) {
  for a in args {
    /* do work here */
  }
}
```

This uses two language features that we'll talk about in the next exercise. `[]` indicates the variable is an array, and `for a in args` loops over all elements of `args`, assigning each in turn to the variable `a`. Configuration variables and built-in options that have already been processed are not passed to `main()`. You will not see them in the list. The exception is `-h` and `--help`, which are passed, with the goal being to allow the program to provide a different help message or to supplement the standard text produced by the Help module. As with C, the first argument will be the program name.

In rw_png_v4.chpl you'll find an example of using `main()` to simulate the argument naming rules for procedures on the command line. That is, with these edits command line options can either be given by their flag (`--inname=<file>`) or positionally. The positional arguments work from left to right, skipping those that have been provided. So without any named options the program will take four values on the command line,

```
    rw_png_v4 <inname> <outname> <x> <y>
```

(the same order as for test_png). If `--x` were given, then three would be needed:

```
    rw_png_v4 --x=<x> <inname> <outname> <y>
```

The changes needed to implement this are to make the four variables `var` instead of `const` (so we can change them), wrapping all the code outside the variable and procedure declarations in `main()`, and adding this loop:

```
    for a in args[1..] {
      if (("-h" == a) || ("--help" == a)) {
        printUsage();
        exit(0);
      }

      if ("" == inname) then inname = a;
      else if ("" == outname) then outname = a;
      else if (x < 0) then x = a : c_int;
      else if (y < 0) then y = a : c_int;
      else usage("too many arguments on command line");
    }
```

There's a couple features in this code that we haven't seen yet. `args[1..]` represents a slice – a subrange of the array. It starts at index 1 and goes on until the array runs out of elements. We need to skip the first element, the program name. The second thing is the presence of a type when `x` or `y` are set, as in `x = a : c_int`. The type here represents a cast, and uses built-in converters from strings to the numeric types. We don't need an equivalent to C's `sscanf()`. So for each argument in the array we test which configuration variable has not yet been set in the order arguments should appear on the command line. The variable will have already been set if it was provided by a command line option.

As usual, compile and run this with

```
    chpl -o bin/rw_png_v4 rw_png_v4.chpl img_png_v2.h build/img_png_v2.o -lpng
or  make rw_png_v4

    bin/rw_png_v4 --inname=bear.png --outname=tstimg.png --x=615 --y=212
```

This is not a recommendation for using `main()` to parse the command line! The purpose of this chapter was to talk about the overall program organization and the start-up sequence. We will not use this style again, but will always employ command-line options. If typing the options gets tiresome you can instead put them in a file, as we'll see in the Gabor Filter chapter.

One other note that hasn't fit elsewhere. Chapel supports both /* and */ and // style comments. /* */ nest, so

```
    /* this is
```

```
/* a wrapped comment */
wrapper */
```

is legal.

Wrap-Up

We've used the need to use a C library to read and write images to cover Chapel's variables and procedures, the basic data structures, and program organization. We've learned:

- There are a few primitive data types: `int`, `uint`, `real`, `imag`, `complex`, `bool`, and `void`. These are qualified by bit-size. Strings are a class that hold an array of ASCII characters and a length. They must be manually translated to C's null-terminated arrays.

- There are three variable types: `param` (compile-time constant), `const` (run-time constant), and `var` (normal variable). Variables may be qualified as `extern` (are imported from C) or `config` (can be provided on the command line).

- Function arguments can be named and can have default values. You can mix positional with named arguments in the function call.

- Function arguments have one of six intents: `in` (input), `out` (value in caller will be changed but function does not see original value), `inout` (function does see original value, and modifies in caller's scope), `ref` (pass by reference rather than value, modifies value in caller), `const in` (input, but cannot be modified within function; this is the default for the primitive types), and `const ref` (pass by reference, but no modifications allowed; this is the default for structural types).

- Functions may have four intents on the return: `ref` (function returns a variable that can be modified, ie. the function can be placed on the left side of an assignment), `type` (function returns a data type), and `param` (function returns an expression suitable for a parameter), `throws` (function may raise an error).

- Functions and some operators may be overloaded. The compiler will chose the appropriate version based on the number and type of arguments in the call.

- Functions may be declared inline.

- Functions may have a variable number of arguments, of potentially different types. Chapel allows reflection on the types at both compile and run time.

- There are three structural types: classes, records, and unions. They can have members, or fields, and methods. They support an inheritance hierarchy (not covered in this chapter).

- Unions may have only one member active at a time. They are copied by value.

- Records and classes are largely the same; the major difference is that records are passed by value (ie. a copy is made) and classes are handled by reference (a copy still refers back to the original).

- Classes and records can have constructors that are called with `new` and destructors for clean-up.

- Class or record methods may have three intents: `type` declares a static procedure accessible through the structure and not an instance, a `ref` method modifies the instance itself, and `param` requires that the instance is a compile-time parameter.

- Records are created and destroyed automatically, although you can use `new` to call a custom constructor. Classes are handled manually. You must call `delete` on an instance to free its memory.

- Importing a pointer-to-struct from C requires a special naming layout, and links to a Chapel class. The class may contain fewer or more members. Only those that match will link to the C structure.

- Chapel programs are split into modules. There is an implicit module per source file, and sub-modules may be declared with the module keyword. The `use` statement allows one module to refer to another's variables and procedures, although fully qualifying the symbol by pre-pending the module name is always possible. You do not have to list files with modules on the command line.

- There may be one `main()` function defined, or the compiler will add one implicitly. Modules are set up in the order of their use, starting with the module with `main()`.

- Chapel uses both single-line // and multi-line /* */ comments. /* */ nest.

Now that we can read in images, it's time to actually some parallel image processing. We'll start with a color conversion program.

Exercises

1. We're writing a greyscale image as a color image, setting all three color planes to the same value. PNG does have a PNG_COLOR_TYPE_GRAY that takes only a single byte instead of three, saving space. Modify `PNG_write()` to take an extra argument whether to save all three color planes or just one (specifying which plane to save). (You'll find our modification in img_png_v3 and rw_png_v5. We'll be using this version for the color converter. If you define an `enum` in C to specify the color planes, then its members can be imported one by one into Chapel by declaring them `extern const`.)

2. Add a cropping function in C that takes one `rgbimage` and the corners of a cropping rectangle, or one corner and a width and height, and creates a new image with just the pixels inside. Create a Chapel program that reads an image, crops it, and writes the smaller image back out.

Files

A tarball with the programs and image for this chapter is available here. A zip file is here.

A PDF copy of this text is here.

COLOR CONVERSION (color)

Introduction

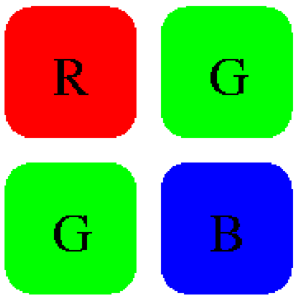
For our first image processing exercise we want something simple that loops through an image pixel by pixel. We'll chose color conversion as our problem, which is a translation from the RGB color space of PNG to another. It will introduce us to the Chapel concepts of domains and ranges, which is where we'll start to see the power of the language. We'll also conclude our survey of the general language, covering expressions, statements, enumerations, and tuples. Along the way we'll build a library (module) for color conversion, since we'll be wanting to use this in the future. As with the previous exercise, we've gone overboard: the original plan was to extract just a greyscale version of a color image. That is done in two versions of the first program, lab_v1.chpl and lab_v2.chpl. We then went ahead and extended it to other color spaces, storing the conversion as real numbers and building a general translation procedure to scale it to an 8-bit 1- or 3-color channel PNG. That is the color_convert.chpl program.

The directory for this chapter is `color`. You'll be using the C routines for reading and writing PNG images in img_png_v3.h and img_png_v3.c (from the first exercise of the PNG chapter). Programs can be built with `make name`, where the name is the Chapel source file without the .chpl suffix. Executables are put in the `bin` sub-directory. The `build` sub-directory holds object files and auto-generated dependencies for the C library. The sample image for this exercise is bobcat.jpg.

The programs and image for this chapter are here (as zip file). A PDF copy of this text is here.

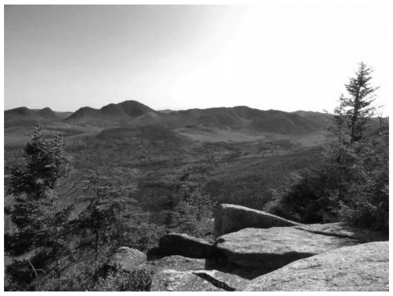
Aside: Color Spaces

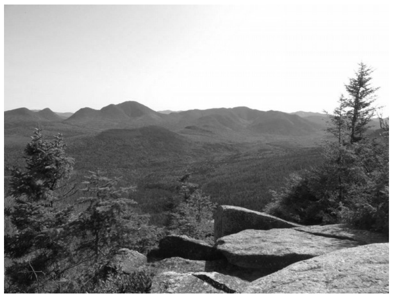
PNG color images contain three color planes corresponding to red, green, and blue, with values between 0 and 255 (inclusive). This follows the data coming from the image sensor, where the pixels are covered by three color filters. Each block of 2x2 pixels has a R G / G B pattern with green filters on a diagonal. This is called a Bayer pattern or matrix. Because every pixel has only one filter, not three, the camera will interpolate the two missing colors from the surrounding pixels in a process called de-mosaicing. By combining the three values we can cover many colors (but not all). For example, yellow is R=255, G=255 and cyan is G=255, B=255. The disadvantages of the RGB color model are that it does not match how we perceive color, which is a non-linear process, it depends on the physical characteristics of the display which are non-linear and change with technology, and it does not accurately reproduce color because there is no standard white reference.

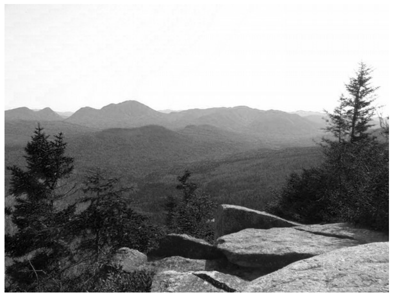


Bayer RGGB matrix

Image to divide into color planes (not provided)


R(ed) plane

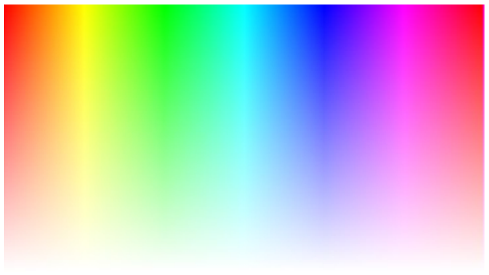

G(reen) plane

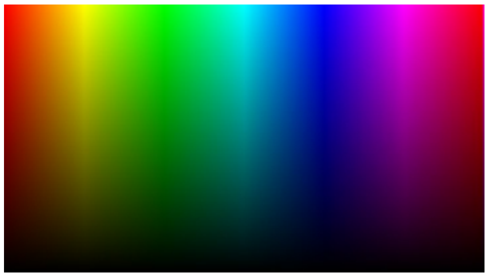

B(lue) plane

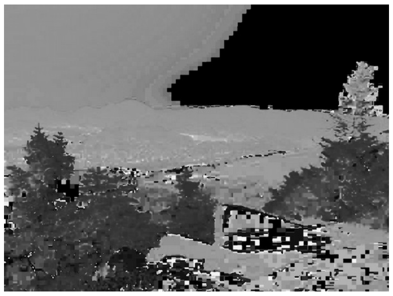
HSV, or Hue Saturation Value, may be familiar to you by tools used to pick colors. It re-maps the RGB cube into a cylinder. As the angle changes, the hue goes from red (at 0 degrees) to yellow (60), green (120), cyan (180), blue (240), magenta (300), and back to red (360). The saturation is the "strongness" of the color, and at low values the color approaches a grey value. The value is related to the brightness of the color. There are several ways to define it, for example the largest R, G, or B component as a fraction of its maximum scale (255). Small values mean the color is dark. The disadvantages of the HSV scale are those of the RGB, since it's just a transformation of that space: brightness is on a linear scale but not perceived at such, perceived brightness depends on the hue, and the values are not well-controlled since the underlying RGB model has no standard definition. The conversion from RGB to HSV we use is:

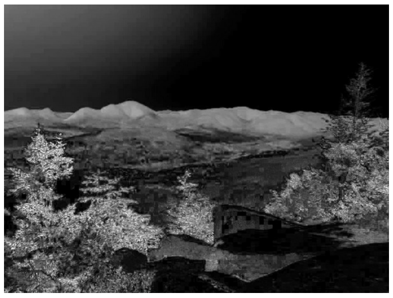
```
H = 60 * (G - B) / (max - min)            if R is the largest value
    120 + (60 * (B - R) / (max - min))    if G is the largest
    240 + (60 * (R - G) / (max - min))    if B is the largest
S = 1 - (min /max)
V = max / 255
```

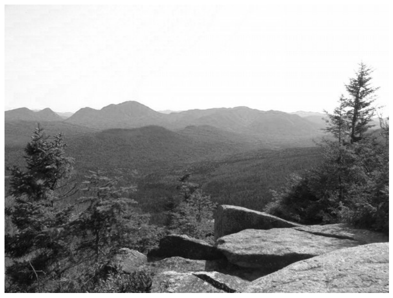
where min is the smallest of the R, G, and B values and max the largest. H takes values between 0.0 and 360.0 (you may need to wrap the calculated value into this range), S between 0.0 and 1.0, and V from 0.0 to 1.0.


H-S plane, V=1.0, H horizontal axis


H-V plane, S=1.0, H horizontal axis

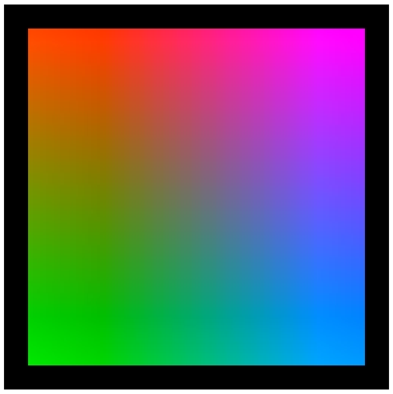

H(ue) plane

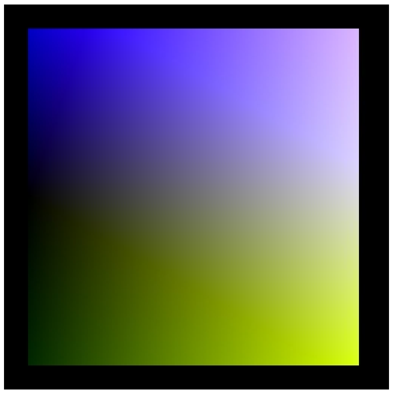

S(aturation) plane

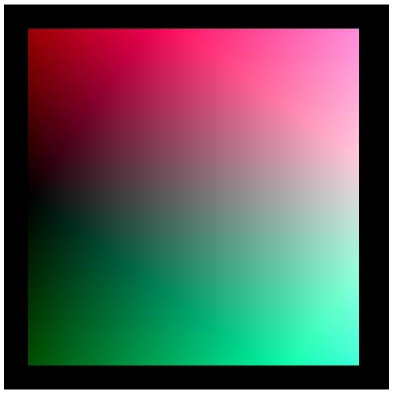

V(alue) plane

YUV or YCrCb is a broadcast standard. The Y channel carries luminance, and is modeled after how we perceive brightness; green dominates this value. The Cr and Cb channels are the red and blue components with luminance subtracted out and scaled as required. In a broadcast system Y corresponds to a black-and-white television signal, with the red and blue difference channels added for color. This is the model used by JPEG images and MPEG video. The YCrCb calculation is supposed to use a non-linear compression, called gamma correction, of the underlying R, G, and B data. The (linear) conversion we use is:

```
    y =  16 + (0.299 * R) + (0.587 * G) + (0.114 * B)     range 16 t/m 235
    u = 128 - (0.168 * R) - (0.331 * G) + (0.500 * B)     range 16 t/m 239
    v = 128 + (0.500 * R) - (0.419 * G) - (0.081 * B)     range 16 t/m 239
```


U-V plane, Y=128, U hor axis


Y-U plane, V=128, Y hor axis


Y-V plane, U=128, Y hor axis

Y plane (luminance/greyscale)

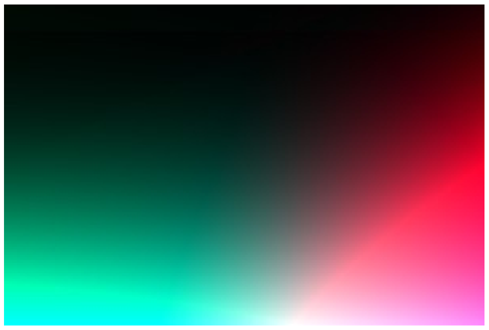
U (Cr) plane

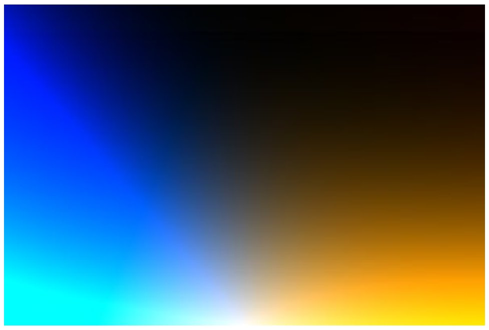
V (Cb) plane

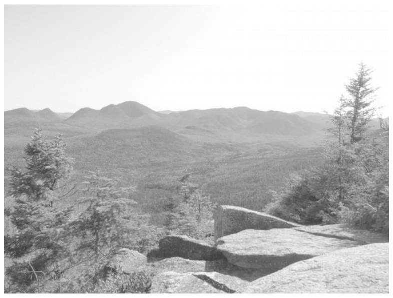
The LAB standard (aka CIE L*A*B*) is a device-independent color space designed to be perceptually uniform, with differences between colors independent of the colors themselves. As with YUV, L is the lightness of a color and A and B are color difference channels. Negative A corresponds to green and positive A to red; negative B corresponds to blue and positive B to yellow. If A and B are 0 then we have a greyscale value. The color space is non-linearly compressed, and can represent more colors than the RGB space. There are LAB points that do not map back to valid RGB values. This space is more difficult to calculate, and uses the CIE XYZ space (which we don't otherwise use) as an intermediate step:

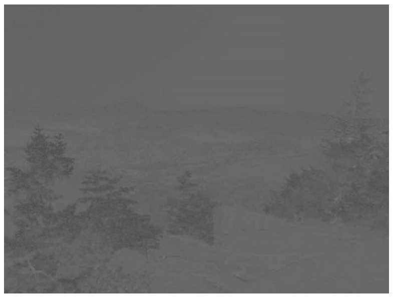
```
    X = ((0.49  * R) + (0.31  * G) + (0.2  * B)) / (255 * 0.17697)
    Y = ((0.177 * R) + (0.812 * G) + (0.011 *B)) / (255 * 0.17697)
    Z =              ((0.01  * G) + (0.99 * B)) / (255 * 0.17697)
    L = 116 * map(Y * 5.6507) - 16
    A = 500 * (map(X * 5.6507) - map(Y * 5.6507))
    B = 200 * (map(Y * 5.6507) - map(Z * 5.6507))
    map(t) = t ** (1/3) if t < 0.2069 or (7.787 * t) + 0.1379 else
```

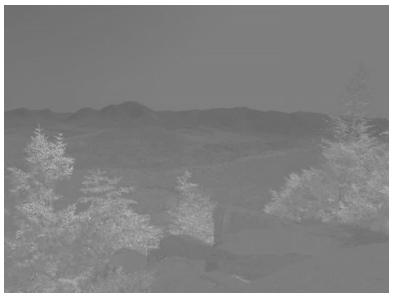
(This mapping is a non-linear compression with a knee point at 6/29.) L takes values between 0.0 and 100.0, A between -500.0 and 500.0, and B from -200.0 to 200.0. In practice the ranges for A and B are much smaller. The scaling factor of 5.6507 for X, Y, and Z is actually related to the choice of a white point, the color at which L=100.0 and A=B=0.0. We chose the value so that (255, 255, 255) is treated as white, but remember that the RGB space is uncontrolled and this may not represent true white in the scene.


L-A plane, B=0, A horizontal axis


L-B plane, A=0, B horizontal axis


L plane


A plane


B plane

Language Description

To avoid scattering our discussions about the language amongst the programs, we'll start by talking about the basic language concepts separately. We'll then discuss arrays, domains, and ranges with the programs as examples.

Expressions and Operators

Operators (listed in order of precedence within each group)

```
  arithmetic      ** * / % + (unary) - (unary) + (binary) - (binary)
  comparison      <= >= < > == !=
  logical         ! & ^ | && ||
  bit             ~ << >> & ^ |
  structure       . new
  parallel        reduce scan dmapped sync single atomic
  other           () [] : .. in if/then/else forall/do for/do
  range           .. by # align
```

Description and Notes

– Unary minus (-) or negation inverts the sign of both components of a complex number.

```
writeln("-(1.0 - 2.0i) = ", -(1.0 - 2.0i));
> -(1.0 - 2.0i) = -1.0 + 2.0i
```

– Arithmetic operations on the largest integer types may be restricted. Combinations of `uint(64)` and `int(64)` raise an error. `uint(64)` cannot be negated. Division drops fractions giving the integer closest to zero.

– Addition (+) and subtraction (–) on a mixture of real and imaginary arguments produce a complex number. Multiplication (*) and division (/) convert between the two as appropriate (ex. imaginary * imaginary = real).

```
writeln("1.0i * -2.0i = ", 1.0i * -2.0i);
> 1.0i * -2.0i = 2.0

writeln("1.0i * 2.0 = ", 1.0i * 2.0);
> 1.0i * 2.0 = 2.0i

writeln("3.0i / 2.0i = ", 3.0i / 2.0i);
> 3.0i / 2.0i = 1.5

writeln("3.0i / 2.0 = ", 3.0i / 2.0);
> 3.0i / 2.0 = 1.5i
```

– Addition (+) where one operand is a string converts the other to a string and concatenates them.

```
writeln("\"concat\" + true = ", "concat" + true);
> "concat" + true = concattrue
```

– Modulus (%) is only defined for integers. The result has the same sign as the dividend.

```
writeln("9 % 5 = ", 9 % 5);
> 9 % 5 = 4

writeln("(-9) % 5 = ", (-9) % 5);
> (-9) % 5 = -4

writeln("9 % (-5) = ", 9 % (-5));
> 9 % (-5) = 4
```

– Integers are treated as a collection of bits when applying the bit operators. That is, the interpretation of the sign is ignored. Combining an `int` with an `uint` gives an `uint`.

```
/* ineg == -86, ipos == 85, u == 204 */
```

```
const ineg : int(8) = 0b10101010 : int(8);
const ipos : int(8) = 0b01010101 : int(8);
const u : uint(8) =   0b11001100 : uint(8);

writef("ineg | u = %08bu (%u)\n", ineg | u, ineg | u);
> ineg | u = 11101110 (238)

writef("ipos | u = %08bu (%u)\n", ipos | u, ipos | u);
> ipos | u = 11011101 (221)

writef("ineg | ipos = %08bu (%u)\n", ineg | ipos, ineg | ipos);
> ineg | ipos = 11111111 (255)

writef("ineg & u = %08bu (%u)\n", ineg & u, ineg & u);
> ineg & u = 10001000 (136)

writef("ipos & u = %08bu (%u)\n", ipos & u, ipos & u);
> ipos & u = 01000100 (68)

writef("ineg & ipos = %08bi (%i)\n", ineg & ipos, ineg & ipos);
> ineg & ipos = 00000000 (0)

writef("ineg ^ u = %08bu (%u)\n", ineg ^ u, ineg ^ u);
> ineg ^ u = 01100110 (102)

writef("ipos ^ u = %08bu (%u)\n", ipos ^ u, ipos ^ u);
> ipos ^ u = 10011001 (153)

writef("ineg ^ ipos = %08bi (%i)\n", ineg ^ ipos, ineg ^ ipos);
> ineg ^ ipos = 11111111 (255)

writef("~ineg = %08bi (%i)\n", ~ineg, ~ineg);
> ~ineg = 01010101 (85)

writef("~ipos = %08bi (%i)\n", ~ipos, ~ipos);
> ~ipos = 10101010 (170)

writef("~u = %08bu (%u)\n", ~u, ~u);
> ~u = 00110011 (51)
```

– Left shift (<<) inserts zeroes in the low-order bits. Right shift (>>) inserts 1's for negative integers and 0's for all other cases.

```
writef("ineg << 2 = %08bu (%u)\n", ineg << 2, ineg);
> ineg << 2 = 10101000 (170)

writef("ipos << 2 = %08bu (%u)\n", ipos << 2, ipos);
> ipos << 2 = 01010100 (85)

writef("u << 2 = %08bu (%u)\n", u << 2, u);
> u << 2 = 00110000 (204)

writef("ineg >> 2 = %08bu (%u)\n", ineg >> 2, ineg);
> ineg >> 2 = 11101010 (170)

writef("ipos >> 2 = %08bu (%u)\n", ipos >> 2, ipos);
```

```
> ipos >> 2 = 00010101 (85)

writef("u >> 2 = %08bu (%u)\n", u >> 2, u);
> u >> 2 = 00110011 (204)
```

– And (`&`), or (`|`), and xor (`^`) can be interpreted as logical operators when they operate on integers, using the normal "0 is false, anything else is true" interpretation.

– And (`&&`) and or (`||`) short-circuit: the second argument is not evaluated if the first is respectively false or true. For example,

```
(denom != 0) && ((num / denom) < 10)
```

will never generate a divide-by-0 exception because the first test means the && must be false if the denominator is 0 and the second term will never be evaluated.

– The relational operators applied to strings cause a character-by-character comparison, where the ordering depends on the encoding. String equality means both operands match exactly.

– Equality (`==`) and inequality (`!=`) for classes checks that the operands do (not) reference the same storage location. For records, the value of each field is compared.

– The by operator (`by`) controls the step-size across a range or domain. The alignment operator (`align`) sets the offset, and the count operator (`#`) adjusts the range to generate a given number of values.

– `if..then..else` as an expression is equivalent to the ternary `?:` in C. The `else` is generally required, except inside a `for` or `forall` loop expression. Only the expression in the `then` or `else` clause that is true is evaluated. For example,

```
a = if (dx == 0) then (pi / 2.0) else atan(dy / dx);
```

– `for` used as an expression performs the loop and returns the results at each step in an array. For example,

```
odd = for i in 1..100 do if (0 != (i % 2)) then i;
```

Here we see an `if..then` expression without an `else`. `odd` will contain the odd numbers from 1 to 100 in an array.

– `new` creates an instance of a class. The dot operator (`.`) accesses a member or method of a class, record, or union.

– Arrays are indexed using parentheses, not brackets, but are declared with brackets. Parentheses are also used to group terms in expressions, hold arguments to functions, and to form tuples. Braces are used to define domains and group statements. Brackets are used to define arrays and for array types, and in `forall` expressions. They may also be used to hold arguments to functions. Yes, this works:

```
proc tstfn(i : int) { writeln("tstfn called with " + i) };
tstfn[2];
> tstfn called with 2
```

The cast operator (`:`) forcibly converts between data types, raising an error if it is not possible. Chapel will implicitly convert mixed type expressions if it can do so without losing information. In other words, an `uint(s)` can cast to an `uint(t)` if t >= s, and to an `int(t)` if t>s. Additionally, Chapel allows conversions from `int(64)` and `uint(64)` to `real(64)` and `complex(128)` even though the destination types cannot hold all the bits. An explicit cast can lose information, for example the extra bits from a larger `int` will be dropped when going to a smaller, and a real may lose precision.

```
const spos : int(16) = 0xcafe : int(16);
writef("0x%xu : int(8) = 0x%xu\n", spos, spos : int(8));
> 0xcafe : int(8) = 0xfe
```

Casting from numbers to booleans follows the "0 is false, all else is true" rule. We have seen in the example with processing the arguments to `main()` that a string can be cast to another type. If the parse fails, then the program will crash with an error. Note that the integer base prefixes 0x, 0o, 0b will not be read.

```
const icast = "1234" : int;
writef("\"1234\" : int        = 0x%xi\n", icast);
> "1234" : int        = 0x4d2

const ucast = "1234" : uint(8);
writef("\"1234\" : uint(8)    = 0x%xu\n", ucast);
> "1234" : uint(8)    = 0xd2

const imcast = "6.626e-34i" : imag;
writef("\"6.626e-34i\" : imag  = %m\n", imcast);
> "6.626e-34i" : imag  = 6.626e-34i

const bcast = "false" : bool;
writef("\"false\" : bool      = %s\n", bcast);
> "false" : bool      = false
```

But not all casts are allowed. The language specification spends a chapter talking about the rules.

The code snippets demonstrating the operators can be found in the file ex_op.chpl. Compile and run them with

```
make ex_op

bin/ex_op
```

Statements

Statements may be grouped into blocks by putting them inside braces. Semicolons terminate statements but are not needed after braces.

Assignments

Assignment statements set the value of the left hand side to the right, or copy a reference if appropriate. Several operators may be combined with the assignment to form a compound assignment.

```
a <op>= b       means  a = a <op> b
```

For example, if `a` = 0b11011; `a` ^= 0b01101 means `a` will end up with the value 0b10110. The compound assignments are +=, -=, *=, /=, **=, &=, |=, ^=, &&=, ||=, <<=, >>=.

Swap

The swap statement transfers the values between two variables.

```
    a <=> b;
```

is equivalent to

```
    const t = b; b = a; a = t;
```

If .. then .. else

We've already seen the `if..then..else` statement in the rw_png_v4.chpl example in the previous section. Use the keyword `then` only if it's followed by a single statement, otherwise a block is needed. `Else`, if desired, must be written and followed by a single statement or a block. If the conditional expression can be evaluated at compile time, then the test will be deleted and the appropriate result will be kept. Each block, or even single statement, following an `if` generates its own scope so that variables can be locally declared.

```
    if (a < b) then writeln("a smaller"); else writeln("b smaller");
    if (max < min) {
      writeln("max smaller than min, swapping");
      min <=> max;
    } else {
      var midpt = (min + max) / 2;
      writeln("midpoint of ", min, " and ", max," is ", midpt);
    }
```

Select

The `select` statement is used to chose between alternatives. It takes an expression that is compared to each values in each `when`, where there may be many comma-separated values/expressions in the list. If there is a match, then the statements(s) afterwards are executed. As with `if..then`, you can place the keyword `do` before a single statement, or use braces to surround a block. There can be one `otherwise` clause that captures any value not matching a `when`. Each block, or single statement, after a `when` is in its own scope.

```
    select imgname {
      when "fisher" do writeln("haven't got a picture of a fisher yet");
      when "bobcat" {
        var rgb : rgbimage;
        PNG_read("bobcat.png", rgb);
        PNG_write("copy_bobcat.png", png);
        free_rgbimage(rgb);
      }
      otherwise writeln("don't know about an image named ", imgname);
    }
```

While .. do, do .. while

The while statement can have either a `while..do` or `do..while` form. The first evaluates its condition before entering its loop, so that if the condition is initially false the `do` statements will never execute. The second form evaluates its condition after the first pass and guarantees that the `do` statements will execute at least once. In the first form the do keyword is only needed if it is followed by a single statement, otherwise it can be replaced by a block. The do is required in the second form. The block, or even single statement, will have its own scope, and in the do..while form this scope extends to the test condition. This means that variables (or constants) declared in the body of the loop can be used in the test.

```
i = -1;
while ((0 <= i) && (i < 5)) { writef("  %i", i) ; i += 1; }
```

does not execute, but

```
i = -1;
do {
  const j = i + 1;
  writef("  %i", i);
  i += 1;
} while ((0 <= j) && (j < 5))
```

produces

```
>   -1  0  1  2  3  4
```

Here j is a constant that is created locally for each pass of the loop.

for

The for statement is a general loop over any data type, including classes, that provides an iterator called these. For built-in types this includes ranges, domains, arrays, and iterators, none of which we've covered yet. The loop variable is declared by the statement and is valid for the block. If the iterator provides multiple values, then you can provide multiple variables that will automatically match. For example, the iterator for a 2D domain will produce a tuple with the x and y coordinates, and we can put a tuple with two variables that will bind to these coordinates. If there is no need for variables then they can be omitted. The format of a for statement is

```
for [param] <loop vars> in <iterator> do <single statement>
for [param] <loop vars> in <iterator> { <block> }
for <iterable> do <single statement>
for <iterable> { <block> }
```

The loop will execute serially once for each value produced by the iterator, and the block, or even single statement, will have its own scope. If the param keyword is provided, then the iterator must be a range formed by parameters, ie. known at compile-time. The loop is then unrolled and replaced in the compiled code by the block evaluated for each possible value of the range. There is no loop in this case.

break, continue, label

The flow of execution within the while and for loops may be changed with the break and continue statements. A continue causes the flow to immediately skip to the end of the block. A break causes the flow to exit the loop. They are not permitted in parallel loops. You can also apply a label to a loop and give the label's name to a continue or break to exit an inner loop.

```
label outer
for y in 1..5 {
  for x in 1..5 {
    if (x == y) {
      writef("\n");
      continue outer;
    } else {
      writef("  (%i,%i)", x,y);
    }
  }
}
writef("\n");
>
>   (1,2)
>   (1,3)   (2,3)
>   (1,4)   (2,4)   (3,4)
>   (1,5)   (2,5)   (3,5)   (4,5)
>
```

use, require

The `use` statement imports a module's symbols, making them available without qualification (you don't need to prefix the symbol with the module name and '.'). Multiple modules may be listed in one `use` statement, separated by commas. It may be placed within a block, and only the block will see the import (essentially the module is placed between the parent's and the block's scope). The `require` statement lists C header, source, and object files and libraries to compile with the Chapel program.

I/O

The I/O statement <~> sends a value on the right side to an output (Writer) on the left, or puts a value from an input (Reader) on the right into a variable on the left.

The code snippets for this section are found in ex_stmt.chpl. Compile and run them with

```
make ex_stmt

bin/ex_stmt
```

Enumerations

An enumeration is a list of identifiers backed by integer values. Unlike C, they are not a synonym for an integer but a separate type. To use an `enum`, you must type its name, a dot, and then the member's name. As an example, consider a list of the color spaces we'll support:

```
enum clrspace {
  HSV, LAB, YUV, RGB
};

config const space : clrspace;
if (clrspace.YUV == space) {
  rgb_to_yuv(r, g, b, c1, c2, c3);
} else if (clrspace.HSV == space) {
  rgb_to_hsv(r, g, b, c1, c2, c3);
} else if (clrspace.LAB == space) {
  rgb_to_lab(r, g, b, c1, c2, c3);
} else {
  writeln("unsupported color space " + space);
}
```

Although the final else isn't really necessary, our practice here is to include it in case we should ever change the contents of the `enum` and forget to add a new value. You could also write this with a select

```
  select space {
    when clrspace.YUV do rgb_to_yuv(r, g, b, c1, c2, c3);
    when clrspace.HSV do rgb_to_hsv(r, g, b, c1, c2, c3);
    when clrspace.LAB do rgb_to_lab(r, g, b, c1, c2, c3);
    otherwise writeln("unsupported color space " + space);
  }
```

The `otherwise` is needed here too because the `select` does not check that all values are covered.

The first identifier in the `enum` is assigned 1, with the value increasing for each member thereafter. You can also assign a number to one and the value will begin to increment from the next. The default value of an `enum` variable is the first member. You can force two identifiers to have the same value, in which case they will be considered equal.

```
  /* HSV=1, LAB=2, YUV=1, RGB=2 */
  enum clrspace_v2 {
    HSV, LAB, YUV=1, RGB
  }

  if (clrspace_v2.HSV == clrspace_v2.YUV) then writeln("HSV, YUV identical");
  > HSV, YUV identical

  if (clrspace_v2.LAB == clrspace_v2.RGB) then writeln("LAB, RGB identical");
  > LAB, RGB identical

  if (clrspace_v2.HSV == clrspace_v2.LAB) then writeln("HSV, LAB identical");
  >

  proc print_space(space : clrspace_v2) {
    writeln("picked color space " + space);
  }

  print_space(clrspace_v2.HSV);
  > picked color space HSV

  print_space(clrspace_v2.YUV);
  > picked color space HSV
```

You can make a configuration variable an `enum`. In this case pass the member's name on the command line, not its value.

```
  config const space : clrspace_v2;
```

```
writef("From the command line you ");
print_space(space);
```

```
./ex_enum
> From the command line you picked color space HSV

./ex_enum --space=LAB
> From the command line you picked color space LAB

./ex_enum --space=YUV
> From the command line you picked color space HSV

./ex_enum --space=2
> <command line setting of 'space'>: error: halt reached - illegal
> conversion of string "2" to clrspace_2
```

Here again we see the aliasing of the `enum` names from the underlying values.

You can use the `enum` as a constant directly, if Chapel can figure out the type to give the value. If it can't, cast it appropriately.

```
enum clrmasks {
  RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff
}

writef("enum vals   R 0x%06xu  G 0x%06xu  B 0x%06xu\n",
       clrmasks.RED, clrmasks.GREEN, clrmasks,BLUE);
> enum vals   R 0xff0000  G 0x00ff00  B 0x0000ff

writef("decompose 0x123456    R 0x%06xu  G 0x%06xu  B 0x%06xu\n",
   0x123456 & clrmasks.RED : uint), 0x123456 & (clrmasks.GREEN : uint),
   0x123456 & (clrmasks.BLUE : uint));
> decompose 0x123456    R 0x120000  G 0x003400  B 0x000056
```

Without the explicit cast this will not compile because the type needed to pass to the `&` function isn't clear. The `%xu` format string, by the way, is for a hexadecimal `uint`. Similarly you can go the other way, casting an int to an enum constant.

```
writef("convert int to enum   ", 0xff0000 : clrmasks);
> convert int to   RED
```

Compilation will fail if the integer is not a value in the `enum`.

The code snippets for enums are in ex_enum.chpl. Compile and run them with

```
make ex_enum

bin/ex_enum
```

Tuples

A tuple is a collection of unnamed elements, not necessarily all the same type. To construct a tuple put a list of types between parentheses. If the types are all the same you can use the `<number>*<type>` notation, which creates that many elements of the given type.

```
    var coord3D_1 : (int, int, int);
    var coord3D_2 : 3*int;
```

Both define two triples where all three members are integers. If you want to create a tuple with just one element, you must use the `*` notation, as Chapel's grammar resolves `(<type>)` as an expression with a type inside parentheses.

```
    var coord1D_1 : 1*int;
```

If you assign a tuple to a new variable, the type of the elements on the right hand side determine the type of the tuple on the left.

```
    var coord3D_3 = (10, 20, 30);
```

This version is also a triple of three integers. Again there's confusion between a single number between parentheses representing an expression or a tuple. To create such a tuple, leave a trailing comma.

```
    var coord1D_2 = (100,);
```

You can retrieve the n'th value in the tuple, starting at 1, by putting the index in parentheses after the name.

```
    writeln("y = ", coord3D_3(2));
    > y = 20
```

You can iterate directly over the elements with a `for` loop.

```
    for x in coord3D_3 do writeln("coord = " + x);
    > coord = 10
    > coord = 20
    > coord = 30
```

Alternately, the method `size` is defined.

```
    for i in 1..coord3D_3.size do writeln("coord " + i + " = " + coord3D_3(i));
    > coord 1 = 10
    > coord 2 = 20
    > coord 3 = 30
```

You cannot loop like this, with either approach, if the tuple has multiple types.

Destructuring is a form of pattern matching where elements in a structure (a tuple in this case) are set position-by-position to elements in a matching structure. The tuples can be nested. If you want to skip a value, put an underscore at that position on the left side. Both sides must have the same number of elements.

```
    var x, y, z : int;
    (x, _, z) = coord3D_3;
    writeln("x = ", x, "  y = ", y, "  z = ", z);
    > x = 10  y = 0  z = 30
```

The second member of `coord3D_3` does not get assigned to anything; `y` keeps its default initialization value.

Destructuring also works if tuples are declared as function arguments by wrapping them in an extra set of parentheses.

Tuples support some operators. The unary operators +, −, ~, and ! are applied to each element, returning the results in a tuple of the same size.

```
coord3D_2 = -coord3D_3;
for x in coord3D_2 do writeln("- coord = " + x);
> - coord = -10
> - coord = -20
> - coord = -30
```

Binary operators combine the elements at their corresponding position.

```
coord3D_1 = coord3D_3 * coord3D_3;
for x in coord3D_1 do writeln("coord*coord = " + x);
> coord*coord = 100
> coord*coord = 400
> coord*coord = 900
```

The operators that work for tuples are +, −, *, /, %, &, |, ^, <<, and >>. If the tuple has a mixed type, then the operator must be valid for all the types. Comparisons can also be made. For == all element pairs must meet the condition, for != one pair must differ.

```
writeln("coord3D_3 == -coord3D_2 ? ", coord3D_3 == -coord3D_2);
> coord3D_3 == -coord3D_2 ? true

writeln("(1,1,1,0,1,1,1) != (1,1,1,1,1,1,1) ? ",
        (1,1,1,0,1,1,1) != (1,1,1,1,1,1,1));
> (1,1,1,0,1,1,1) != (1,1,1,1,1,1,1) ? true
```

For >, >=, <, and <= elements are scanned from left to right. If the pair is equal it is skipped, otherwise the relationship evaluates to true or false depending on if the condition is satisfied. If all elements are checked, then the result if true if using >= or <=, else false. In other words, common elements at the start of the tuple are ignored, and >= or <= only apply if the two are completely equal, with > and < failing if they are.

```
writeln("fail 1st index: (1, 2, 3) >  (2, 2, 2) ? ", (1, 2, 3) > (2, 2, 2));
> fail 1st index: (1, 2, 3) >  (2, 2, 2) ? false

writeln("pass 1st index: (3, 2, 1) >  (2, 2, 2) ? ", (3, 2, 1) > (2, 2, 2));
> pass 1st index: (3, 2, 1) >  (2, 2, 2) ? true

writeln("fail 2nd index: (2, 1, 3) >  (2, 2, 2) ? ", (2, 1, 3) > (2, 2, 2));
> fail 2nd index: (2, 1, 3) >  (2, 2, 2) ? false

writeln("pass 2nd index: (2, 3, 1) >  (2, 2, 2) ? ", (2, 3, 1) > (2, 2, 2));
> pass 2nd index: (2, 3, 1) >  (2, 2, 2) ? true

writeln("fail 3rd index: (2, 2, 1) >  (2, 2, 2) ? ", (2, 2, 1) > (2, 2, 2));
> fail 3rd index: (2, 2, 1) >  (2, 2, 2) ? false

writeln("pass 3rd index: (2, 2, 3) >  (2, 2, 2) ? ", (2, 2, 3) > (2, 2, 2));
> pass 3rd index: (2, 2, 3) >  (2, 2, 2) ? true
```

```
writeln("fail at end:    (2, 2, 2) >  (2, 2, 2) ? ", (2, 2, 2) > (2, 2, 2));
> fail at end:    (2, 2, 2) >  (2, 2, 2) ? false

writeln("pass at end:    (2, 2, 2) >= (2, 2, 2) ? ", (2, 2, 2) >= (2, 2, 2));
> pass at end:    (2, 2, 2) >= (2, 2, 2) ? true
```

The code for these examples is in ex_tuple.chpl. Compile and run it with

```
make ex_tuple

bin/ex_tuple
```

Arrays and Ranges (lab_v1.chpl)

The goal for this section is to convert our color input image to greyscale so that we can do further image processing. We have two options, either calculate the L plane from the LAB space or the Y plane from YUV (YCrCb). We'll pick the LAB transform, as defined above. For each pixel in the image we will need to calculate L, and we will want to scale the value to an `uint(8)` (`c_uchar`) so we can write the result as a PNG. Although we could do the conversion and scale it for display in one step, we'll break it into two. This will be closer to our workflow later, of passing intermediate results from one processing routine to another until we're ready to save the result. Therefore, we will create temporary arrays for the conversion and store all three components. We'll further develop this approach in the next version.

An array declaration looks like

```
var <name> : [<domain>] <type>
```

A domain represents the indices that are valid for the array. It provides iterators so that we can traverse the set. Domains may have as many dimensions as desired. They may use ranges of numbers, or a discrete set of values, possibly non-numeric. They may be subsets of other domains or have a continuous (dense) or sparse set of indices. They may be mapped to a specific runtime configuration involving different locales, which are computational centers with processing and storage. Domains are one of the key abstractions Chapel provides for safely working with arrays in a parallel environment.

We will be using a rectangular domain based on a range of numbers. A range is specified by its lower and upper bounds, the stride or the increment in the index between steps, and its alignment or offset from 0. The format for a range is

```
 [<start>] .. [<end>] [by <stride>] [align <offset>]
```

Either the start or the end can be omitted; in this case the range will generate a potentially infinite sequence of numbers. You can use the range to iterate over the sequence only if it has a start; a range unbounded at the start can be used, however, to select a subset of another range, or if the stride is negative and you indicate how many steps to take. An unbounded end can also be used to select a subset, or you can define the number of iteration steps that will be taken with a positive stride. Note that the end value is inclusive.

You can think of the stride and alignment defining an infinite sequence of numbers, spaced by stride and including align. The start and end points clip this sequence so that only members that fall between them (inclusively) are generated. If the start or end are omitted then the sequence is infinite and can only be used if it modifies another range or you provide a count of how many values you want.

Some examples. If we want to define a range covering all the pixels of an image

```
    0 .. (npix-1)
```

because the end is inclusive (in C we would write the loop as '`for (xy=0; xy<npix; xy++) { .. }`'). The default stride is 1 and the default alignment is 0. This generates all numbers from 0 t/m `npix-1`. If we want to define a range covering one row, `y`,

```
      (y*ncol) .. ((y+1)*ncol - 1)
```

assuming that we treat the array as `nrow` rows of `ncol` pixels each. The end of the row is just the pixel before the start of the next row `(y+1)`. If we want to define a range covering the first column

```
      0 .. (npix-1) by ncol
```

That is, we increment the index by an entire row between steps. If we want a column `x` in the middle, we can set a range in two ways. First, we can change the start of the range

```
      x .. (npix-1) by ncol
```

This generates the numbers `x`, `x+ncol, x+2*ncol, ... x+(nrow-1)*ncol` through the last row. Alternately we can specify the alignment

```
      0 .. (npix-1) by ncol align x
```

without changing the bounds that define the entire image. Qua style this is a better approach because it makes clear what the limits of the image are and how we should cover pixels within it. The first option is closer to how the loop would be written in C, however, and years of practice means it feels more natural.

The test program ex_range.chpl prints out the indices generated by a small 10x10 image. It can take two command line arguments `--x` and `--y` to print a row or column. Compile and run it with

```
    make ex_range

    bin/ex_range
    >  0   1   2   3   4   5   6   7   8   9
    > 10  11  12  13  14  15  16  17  18  19
    > 20  21  22  23  24  25  26  27  28  29
    > 30  31  32  33  34  35  36  37  38  39
    > 40  41  42  43  44  45  46  47  48  49
    > 50  51  52  53  54  55  56  57  58  59
    > 60  61  62  63  64  65  66  67  68  69
    > 70  71  72  73  74  75  76  77  78  79
    > 80  81  82  83  84  85  86  87  88  89
    > 90  91  92  93  94  95  96  97  98  99

    bin/ex_range --x=5     /* 0..lastpix by ncol align x */
    > column  5:   5  15  25  35  45  55  65  75  85  95

    bin/ex_range --y=5     /* y*ncol..(y+1)*ncol-1 */
    > row    5:  50  51  52  53  54  55  56  57  58  59
```

The program has checks that the column and row values are within the image. If you remove them, you'll notice that the column can be any number and still generate a proper, valid sequence of indices. That's because the alignment is taken modulo the number of columns – it is folded back into the sequence. Changing the alignment shifts the infinite sequence, and a shift by a multiple of the number columns intersects the image bounds at the same points. This would not happen with the first alternative: if `x` was greater than or equal to `ncol` then we would skip rows at the start, and if it was greater than `npix-1` we would generate no indices at all (if end < start we have an empty set).

If the stride is negative the start and end indices are reversed internally so that the sequence counts down. When writing out the range, however, still keep start < end.

Ranges are proper data structures and can be assigned to variables. Starting with an existing range, we can generate a second in four ways.

1. `by` and `align` can be applied, modifying the base sequence. For example, here the first range generates the odd integers to 20, and the second by takes every other one of those, or 1, 5, 9, 13, and 17.

```
 (1 .. 20 by 2) by 2
```

2. We can say how many indices we want to generate. This is the count operator and the command looks like

```
<range> # <number>
```

This adjusts the bounds of the range, but not the stride, to create `<number>` steps. If `<number>` is positive the values are taken at the start of the sequence, if negative from the end. This will generate the indices of column x in the first five rows, or x, x+ncol, x+2*ncol, x+3*ncol, x+4*ncol.

```
 (0 .. npix-1 by ncol align x) # 5
```

This for the last five rows, or x+(nrow-6)*ncol, x+(nrow-5)*ncol, ..., x+(nrow-1)*ncol.

```
(0 .. npix-1 by ncol align x) # -5
```

3. We can add or subtract a number from the start and end values. This also shifts the alignment, but not the stride. For addition the number can be placed on either side of the range, ie. `<range> + <number>` or `<number> + <range>`; for subtraction it can only go to the right, ie. `<range> - <number>`.

4. We can use a second range to take a subrange of the first. This is called slicing, and the resulting set is the intersection of the two ranges. To define a slicing, place the second range in parentheses or brackets after the first

```
<range1>(<range2>)    or    <range1>[<range2>]
```

where, as usual, the range can be either fully written out or a variable to which a range has been assigned.

For this program we only want a range to cover the entire image. The conversion to LAB is the heart of the program. Let `rgb_to_lab()` be the top-level function that processes the entire image, and `rgbpix_to_lab()` a function to handle one pixel.

```
const LAB_LMIN =    0.0;
const LAB_LMAX =  100.0;
const LAB_AMIN = -128.15;
const LAB_AMAX =  182.46;
const LAB_BMIN = -155.36;
const LAB_BMAX =  156.20;
```

```
proc rgb_to_lab(rgb : rgbimage, ref lab : rgbimage) : c_int {
  const imgbounds = 0 .. (rgb.npix-1);   /* range covering image */
  var l : [imgbounds] real;              /* L color plane */
  var l_a : [imgbounds] real;            /* A color plane */
  var l_b : [imgbounds] real;            /* B color plane */
  var retval : c_int;

  retval = alloc_rgbimage(lab, rgb.ncol, rgb.nrow);
  if (retval < 0) then return retval;

  for xy in imgbounds {
    rgbpix_to_lab(rgb.r(xy), rgb.g(xy), rgb.b(xy), l(xy), l_a(xy), l_b(xy));

    clamped = clamp(l(xy), LAB_LMIN, LAB_LMAX);
    lab.r(xy) =
      ((255.0 * (clamped-LAB_LMIN) / (LAB_LMAX-LAB_LMIN)) + 0.5) : c_uchar;
    clamped = clamp(l_a(xy), LAB_AMIN, LAB_AMAX);
    lab.g(xy) =
      ((255.0 * (clamped-LAB_AMIN) / (LAB_AMAX-LAB_AMIN)) + 0.5) : c_uchar;
    clamped = clamp(l_b(xy), LAB_BMIN, LAB_BMAX);
    lab.b(xy) =
      ((255.0 * (clamped-LAB_BMIN) / (LAB_BMAX-LAB_BMIN)) + 0.5) : c_uchar;
  }
  return 0;
}

proc rgbpix_to_lab(r : c_uchar, g : c_uchar, b  :c_uchar,
                   out l : real, out l_a : real, out l_b : real) {
  /* see source for implementation */
}

proc clamp(val : real, minval : real, maxval : real) : real {
  if (maxval < minval) then return minval;
  else if (val < minval) then return minval;
  else if (maxval < val) then return maxval;
  else return val;
}
```

Here `rgb` is the PNG image we've read and `lab` is the greyscale image we'll create. The `imgbounds` range has been used to create the arrays, as well as driving the `for` loop. `rgbpix_to_lab()` uses an `out` intent on the three components to modify the array elements in the caller directly. Internally it calls two other procedures for the XYZ conversion and non-linear mapping. Remember that the LAB space is bigger than the RGB, so we define constants `LAB_[LAB]MIN` and `LAB_[LAB]MAX` that specify how to scale each component (Example 2 asks you to explain these values). The `clamp()` function limits the value to this range, and then we scale the value and convert it to a `uint(8)`. The 0.5 term is for rounding after the cast. We end up storing L in the R plane of the lab image, A in the G plane, and B in the B(lue) plane. We can specify in the `PNG_write()` call which plane we want to save since we're using the version from Exercise 1 of the previous chapter. `CLR_R` is the luminance/greyscale information.

The `main()` routine is simple. It checks that the command line arguments are valid, reads the image, calls `rgb_to_lab()`, and writes the result.

```
config const inname : c_string;          /* input file name */
config const outname : c_string;         /* output file name */
proc main() {
  var rgb: rgbimage;                      /* image we've read */
  var grey : rgbimage;                    /* LAB transform in 8 bit grey */
  var retval : c_int;

  verify_setup();

  retval = PNG_read(inname, rgb);
  end_onerr(retval, rgb);

  retval = rgb_to_lab(rgb, grey);
  end_onerr(retval, rgb, grey);

  retval = PNG_write(outname, rgb, CLR_R);
  end_onerr(retval, rgb, grey);

  free_rgbimage(rgb);
  free_rgbimage(grey);

  return 0;
}
```

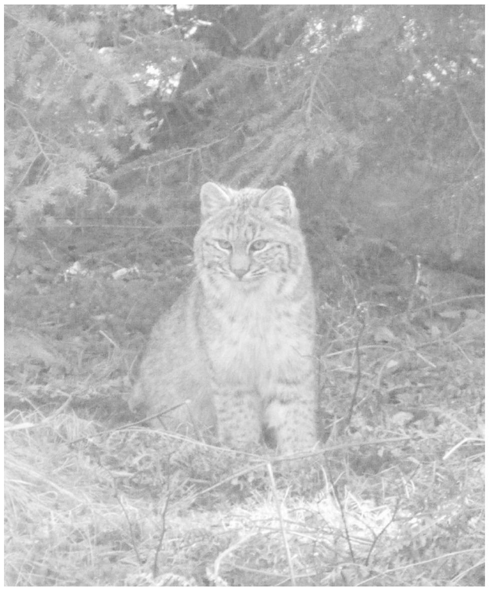
Compile the program with

```
make lab_v1
```

It takes two command line arguments. `--inname=<PNG file>` is the file to read, and `--outname=<file>` is the file to create. You can try the program with

```
bin/lab_v1 --inname=bobcat.png --outname=grey_bobcat.png
```

Original image bobcat.png


Luminance/greyscale channel

Domains (lab_v2.chpl)

For the second version of the LAB conversion, we'll re-organize the program to make the two steps of the process, conversion and 8-bit scaling, clear. We'll move the intermediate arrays we used into a separate data structure. This gives us a chance to change the way we represent the image. We use a flat array when working with C because building a 2D array, allocating row by row, is annoying. This requires that we transform (x, y) image coordinates to an xy index; it's not a huge burden, creates standard idioms in code for loops, and is efficient at addressing, but it does obscure the intent of the code if you aren't familiar with the approach. With Chapel's ranges and domains it will be easy to use the coordinates instead of the index, so we will change to a 2D domain. As we split off the 8-bit scaling, we will also add different strategies other than over a fixed range, which lab_v1 uses. This re-organization will lay the groundwork for the final program, a general colorspace converter.

A domain has a number of dimensions called a rank. In lab_v1 the domains behind our arrays had a rank of 1: they used a single index. In our Chapel images we want to work with rank 2. Indices will be represented by a tuple (x, y). Our domain is rectangular and dense, meaning we will have data at all points inside it. In Chapel's terminology we'll use a base, rectangular domain. To declare a domain, call the constructor

```
    domain(rank=<number dimensions>, idxType=int, stridable=false);
```

As the named arguments imply, you can change the type of the index generated. The stridable flag turns on support for the 'by' keyword, similar to striding in ranges. Since each index will have the same type, we can also create a domain by passing a tuple with all elements of that type

```
    domain(3*int);
```

Domains are proper data structures and may be assigned to variables.

For our images, we want a 2D domain. We'll start building a data structure to hold the arrays, using a class. We need to track the image size and will still calculate the number of pixels, although there's less need for that since we aren't using a 1D index. We've already seen how to define an array over a domain; we'll use `c1`, `c2`, and `c3` to represent generic color planes whose interpretation depends on the color space we're in. (Because we don't usually mix color spaces within a program, we won't use an enum inside the data structure to track which space is being used.) The members of the class are

```
class clrimage {
  var ncol : int;                  /* width (columns) of image */
  var nrow : int;                  /* height (rows) of image */
  var npix : int;                  /* number pixels = w * h */
  var area : domain(rank=2);       /* array bounds by x and y */
  var c1 : [area] real;            /* first color plane (L, H, Y, R) */
  var c2 : [area] real;            /* second plane (A, S, U, G) */
  var c3 : [area] real;            /* third plane (B, V, V, B) */
}
```

Each dimension of a domain is specified with a range. The domain is the product of all the ranges, that is, all combinations of their values. The indices are generated in rank order, so the range in the first rank varies the least. That is, the indices are formed by a series of nested loops, with the first rank the outermost and the last the innermost. To specify a domain, put the ranges in a list between braces.

```
area = {0..nrow-1, 0..ncol-1}
```

defines the bounds of our image, using 0-based indexing. The first rank runs over y (along a column), the second over x. Loops will match the pixel layout in memory, moving through the linear array. You cannot use strides with these ranges.

The size of an array depends on the underlying domain. When the domain changes, the array changes. The domain must be kept in a variable to do this, as we have done above. You cannot change the domain assigned to an array (although there is a method to access it), but you can change the contents of the domain if it's been given a name. As we see above, one advantage of this approach is that domains can be shared. If we change `area`, all three color planes will grow or shrink accordingly. During the re-size, array elements in the overlap between the old and new domain specifications are kept and any elements added are initialized to the default of the data type.

`area` here is a placeholder. Once we've read in the PNG, we can then set up the Chapel image using a constructor we add to the class.

```
class clrimg {
  /* members defined above */
  proc clrimg(w: int, h: int) {
    ncol = w;
    nrow = h;
    npix = w * h;
    /* This automatically re-sizes the arrays. */
    area = {0..nrow-1, 0..ncol-1};
  }
}
```

To use this, we modify the `rgb_to_lab()` procedure to take a `clrimage` instead of an `rgbimage`. We instantiate the image, and use the domain to convert each pixel. Since we're now generating the x and y coordinates from the domain, we need to calculate the xy index for the `rgbimage` inside the loop. The `c1`, `c2`, and `c3` array references take (x,y) as a tuple.

```
proc rgb_to_lab(rgb : rgbimage, ref lab : clrimage) {

  lab = new clrimage(rgb.ncol, rgb.nrow);

  for (y, x) in lab.area {
    const xy = (y * rgb.ncol) + x;
    rgbpix_to_lab(rgb.r(xy), rgb.g(xy, rgb.b(xy),
                  lab.c1(y,x), lab.c2(y,x), lab.c3(y,x));
  }
}
```

With this framework we'll be able to support different color space conversions by creating a `rgbpix_to_hsv()` or `rgbpix_to_yuv()` function.

We next need to convert the real results to 8-bit. In lab_v1 the valid data range was fixed: if L, A, or B went outside the defined limits they were clamped before scaling to 0 t/m 255. The range differs for each plane, however, and other color spaces use different strategies. We will need

1. BOUND - The limits of the data are known. The minimum is offset to 0, the maximum scaled to 255. Values outside the limits are set to 0 or 255. The minimum and maximum must be specified.

2. DATA - The limits are set to the smallest and largest values in the array. Pixels after scaling will span 0 t/m 255.

3. CENTER - The limits are set to +/- the largest absolute value in the array, putting 0 in the middle. Unless the minimum is the opposite of the maximum, the pixel values will not cover the full 8-bit range.

4. CLIP - Do no scaling. Values outside the 8-bit range are clipped to 0 or 255.

We'll use an `enum` to represent these cases.

```
enum rgbconvert {
  BOUND, DATA, CENTER, CLIP
}
```

For the general framework, we'll want to support which plane to copy to the PNG. If we pick one of `C1`, `C2`, or `C3` then we'll populate the RGB planes with the scaled value from the source plane. We'll also allow an `ALL` option which will put `C1` in `R`, C2 in `G`, and `C3` in `B`. The `enum` to specify this is

```
enum clrplane {
  C1, C2, C3, ALL
}
```

A record will store the complete conversion spec

```
record conversion {
  var plane : clrplane;          /* which plane to copy to RGB */
  var how : rgbconvert;          /* how to scale clr to rgb */
  var min : real;                /* lower limit of bound/clip range */
  var max : real;                /* upper limit of bound/clip range */
}
```

The routine that starts the conversion is called `display_color()`.

```
proc display_color(clr : clrimage, ref rgb : rgbimage,
                   spec : conversion) : c_int
```

For each plane that must be copied to RGB, it calls `display_plane()` with the source `clrimage` array and the destination rgbimage pointer.

```
proc display_plane(clr : [] real, rgb : c_ptr(u_char), ncol : int,
                   spec : conversion) : c_int
```

This prototype shows how we can pass arrays and, for the C interface, pointers. The image width is necessary, since it's not stored with the C array and we need it to calculate the xy index. The domain for the Chapel array, however, is available through the array method `clr.domain`. (We do not assume both images have the same size, but `rgb` must be the same size or larger of the two.) The return values follow the error code convention of `PNG_read()` and `PNG_write()`.

In other words, `display_color()` routes the source color plane to the output plane with `display_plane()`. Except if all planes are being mapped, it also needs to make a greyscale image by copying the result into the other two planes using a range over all the pixels in the RGB image.

```
if ((clrplane.C1 == spec.plane) || (clrplane.ALL == spec.plane)) {
  retval = display_plane(clr.c1, rgb.r, rgb.ncol, spec);
  if (retval < 0) then return retval;

  if (clrplane.C1 == spec.plane) {
    for xy in 0..(rgb.npix-1) {
      rgb.g(xy) = rgb.r(xy);
      rgb.b(xy) = rgb.r(xy);
    }
    return 0;
  }
}
/* Similarly for C2/G, C3/B. */
```

`display_plane()` uses a reduction to yield a single value from a set. It has the form

```
<operation> reduce <iterable>
```

`<operation>` is a small set of operators and functions: + and `*`; `&`, `&&`, `|`, and `||`; `min`, `max`; and the extrema with their location, `minloc` and `maxloc`. The `reduce` command is a concise way to find the smallest and largest color values for the DATA and CENTER scaling.

```
minpix = min reduce clr;
maxpix = max reduce clr;
```

If we were to write this out, it would like like

```
var minpix = max(real);
var maxpix = min(real);
for (y, x) in area {
  if (clr(y,x) < minpix) then minpix = clr(y,x);
  if (maxpix < clr(y,x)) then maxpix = clr(y,x);
}
```

In this snippet we're using the `min(<type>)` and `max(<type>)` procedures defined in modules/standard/Types.chpl that return the limit values for a given type. The rest of the procedure calculates the scaled color value, then converts it to an `uint(8)`. The conversion is the same as in lab_v1, except for the CLIP case.

Compile and run the program with

```
make lab_v2

bin/lab_v2--inname=bobcat.png --outname=grey_bobcat.png
```

If you run both lab_v1 and lab_v2 you can see that the output image is the same.

Program Organization (color_convert.chpl)

Our final program re-organizes lab_v2.chpl, pulling out the color transformations and 8-bit conversions into a separate module (file), ip_color.chpl, and adding RGB-to-HSV and RGB-to-YUV converters. The color_convert.chpl program uses this library to transform an input image into a new color space, choosing which plane to save as a greyscale image.

We don't need any new language features. ip_color.chpl contains both the color conversion library as well as the C interface. We may wish to separate these in the future, but we expect that every program that will use this library will want to do immediately convert its input into another color space, so it makes sense to leave them in the same module. The library adds four new top-level functions, `rgb_to_hsv()`, `rgb_to_yuv()`, and `rgb_to_rgb()` which convert an 8-bit RGB image into the new spaces (the last function merely copies the pixels, changing them to reals), as well as a routing function `rgb_convert()`.

color_convert also does nothing new. It adds command-line configuration constants for the color space and plane. In `main()` it transforms the color, sets up the conversion spec back to 8-bit, and generates and saves the image for the requested plane.

To build and run the color conversion

```
make color_convert

bin/color_convert --inname=<file> --outname=<file> --space=[LAB|HSV|YUV]
                  --plane=C[123]
```

where the enumeration constants are given by name and default to LAB and C1. For example,

```
bin/color_convert --inname=bobcat.png --outname=bobcat_h.png --space=HSV
```

Aside: First-Class Functions (color_convert_v1b.chpl, ip_color_v1b.chpl)

The directory CHPL_HOME/doc/rst/technotes contains documentation for Chapel features that are works in progress and not yet ready to add to the language spec. One of these is support for functions, both as a type and as something that can be assigned to variables. You'll notice that all the `rgb_to_*` procedures in ip_color are the same except for the function call to convert a pixel. We can modify `rgb_convert()` to show how first-class functions can simplify them.

Chapel places strong restrictions on the functions. They cannot be generic, they cannot belong to a class or record, and they cannot be overloaded, among others. They cannot refer to non-global variables, although arguments may be declared with an intent that allows them to be modified. More importantly for us, their argument lists cannot take non-default intents. Therefore we have to re-work all four conversion functions to return a triple:

```
proc rgbpix_to_lab(r : u_char, g : u_char, b : u_char) : 3 * real {
  var l, l_a, l_b;
  /* convert normally */
  return (l, l_a, l_b);
}
```

The function type is formed with the `func` keyword and a list of the argument types in parentheses. The final argument is the return type. The prototype of our conversion function is

```
var fn : func(u_char, u_char, u_char, 3 * real);
```

You can then assign a procedure name to this variable and then use the variable as the name when calling the procedure. In `rgb_convert()`, we'll select the proper conversion routine, then do what all the `rgb_to_*` procedures did: create an instance of the destination image and loop through all the pixels, converting each in turn. Like this

```
proc rgb_convert(rgb : rgbimage, ref clr : clrimage,
                 space : clrspace) {
  var clrfn : func(c_uchar, c_uchar, c_uchar, 3 * real);
  var xy : int;

  select space {
    when clrspace.LAB do clrfn = rgbpix_to_lab;
    when clrspace.HSV do clrfn = rgbpix_to_hsv;
    when clrspace.YUV do clrfn = rgbpix_to_yuv;
    when clrspace.RGB do clrfn = rgbpix_to_rgb;
    otherwise halt("unknown colorspace " + space);
  }

  clr = new clrimage(rgb.ncol, rgb.nrow);

  for (y, x) in clr.area {
    xy = (y * rgb.ncol) + x;
    (clr.c1(y,x), clr.c2(y,x), clr.c3(y,x)) =
        clrfn(rgb.r(xy), rgb.g(xy), rgb.b(xy));
  }
}
```

The other edits to ip_color_v1b deleted the four `rgb_to_*` variants and added rgbpix_to_rgb().

To test this we've modified color_convert_v1b.chpl to use this library. Compile it with:

```
    chpl -o bin/color_convert_v1b color_convert_v1b.chpl ip_color_v1b.chpl \
         img_png_v3.h build/img_png_v3.o -lpng
or  make color_convert_v1b
```

Run it the same way

```
    bin/color_convert_v1b --inname=bobcat.png --outname=bobcat_h.png --space=HSV
```

This is just an example of how you can pass around functions in variables. It would be just as legitimate to move the select into the loop and call the right `rgbpix_to_*` function, although you would have to duplicate the argument list for each case.

Chapel also supports unnamed functions called lambdas. You create them when you assign them to a variable. The lambda can take an argument list and is followed by a block of statements.

```
    var fn = lambda(x : int, y : int, ncol : int) { return ((y * ncol) + x); }
```

A lambda is evaluated as an expression.

Wrap-Up

In this chapter we've completed our survey of the basic language, including operators and statements and two additional data structures, enumerations and tuples. We've also seen how Chapel represents domains, the set of indices that back arrays, and the range specifications that generate them. We've learned:

– Chapel offers the normal arithmetic, logical, comparison, and bit operators. It has a ternary `if..then..else` expression and can generate a set of results from an in-line loop.

– Numbers will be implicitly cast to other data types only if no information can be lost; the exception is that `int(64)` and `uint(64)` can implicitly cast to `real(64)` and `complex(128)`. You can force a cast by putting `: <type>` after the expression, although not all casts are allowed. Simple numeric strings, without base specifiers at the front, can be cast to the numeric types.

– Assignment statements can be compound, combining an operation on the left hand side with the assignment (`a **= b` means `a = a ** b`). There is a swap assignment `<=>`.

– The control statements offer a keyword version if followed by a single statement, or support blocks of statements between braces ('`if <cond> then <stmt>`' vs. '`if <cond> { <stmts> }`'). All following statements, whether a block or single, form their own scope for local declarations.

– The loop constructs are `while..do`, `do..while`, and `for`. Loops can be broken out of or continued, and outer loops can be labeled and used as targets of the `break` or `continue`.

– A var or const declaration in the `do..while` form is available in the while test.

– Enumerations are a set of named constants. The name must be fully qualified in the code, `<enum>.<member>`. A `config enum` variable can be set on the command line with the name of the member. The values of the constants are available, although you may have to cast them to `int` if the language can't recognize them in an implicit cast.

– Tuples are anonymous collections of elements, possibly of different types. They may be assigned via destructuring, where elements on the left hand side are assigned the value in the corresponding position on the right. Tuples support a few arithmetic and bit operators.

– Ranges are specifications for generating a sequence of numbers. They may have a start and an end, an increment between steps, and a value that must be present in the sequence. You can also limit the number of values generated, if that makes a clearer specification. They are proper data structures and may be assigned to variables. Ranges may be applied to other ranges.

– Domains define an n-dimensional volume. The rank n is the number of dimensions, and each axis may be a range of numbers or a discrete set of values.

– Domains should be assigned to a variable, and that variable used when declaring an array. This allows changing the size of the domain and having the array re-size automatically.

– Support for treating procedures as first-class citizens has been added. You can assign procedures to variables and use the variable as the procedure name when making a call. Anonymous functions or lambdas are also available.

Now that we can generate greyscale images from our color, we can start doing some image processing. Since array operations are a key concern for the language, let's look at convolutions, a.k.a. matrix multiplication, next.

Exercises

1. Add the inverse conversions from the LAB, YUV, and HSV spaces back to RGB. You can find the formulas on the web. You'll find our version in ip_color_v3.chpl, used from the k-Means Clustering chapter onward.

2. Confirm the values of LAB_AMIN, _AMAX, _BMIN, and _BMAX. Because we have a limited 8-bit RGB space, we can exhaustively run all RGB combinations and find the actual limits. You'll find our approach in lab_limits.chpl in the Parallel Programming chapter.

Files

A tarball with the programs and image for this chapter is available here. A zip file is here.

A PDF copy of this text is here.

GABOR FILTER (gabor)

Introduction

Convolutions are an essential part of low-level image processing. They represent a local neighborhood calculation: at each pixel we sum a weighted contribution from those surrounding it. The weights are placed in a kernel, a size x size matrix. Think of a convolution as shifting this kernel over an image. At each point we multiply the pixels by the kernel values that lie atop (treating weights outside the kernel as 0, ie. these pixels are ignored) and sum when done. A convolution can be a low-pass filter, emphasizing broad features in the image. If the kernel elements are all the value 1/(size*size), then we compute the local average value. A Gaussian filter has a bell-shape. Small low-pass filters are often used to reduce noise in an image, large filters effectively lower the resolution. A convolution can also be a high-pass filter, emphasizing local changes. These might indicate edges, or steps in brightness. The Gabor filters are tunable and can belong to either class.

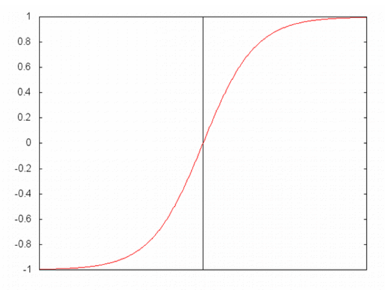
In this chapter we write a convolver with a programmable kernel. If it were fixed, then we could optimize the program by per-calculating the memory accesses and arithmetic operations, but for this approach we will need to loop over the kernel. This introduces us to Chapel's concept of subdomains and subranges, and we'll look at a few different ways to express the computation.

The directory for this chapter is `gabor`. You'll be using the C routines from the first chapter for reading and writing, and the color conversion procedures from the second for generating greyscale values. Use `make name` to build the programs, where the name is without a .chpl suffix. The `bin` sub-directory will contain executables, the `build` sub-directory intermediate files. The sample image for this exercise is francnotch.png. It has foreground objects with strong edges at different angles.

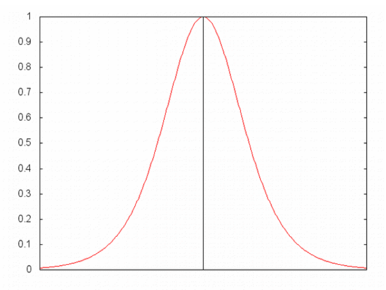
The programs developed in the chapter are in a tarball here (as zip file). A PDF copy of this text is here.

Aside: Edge Detectors

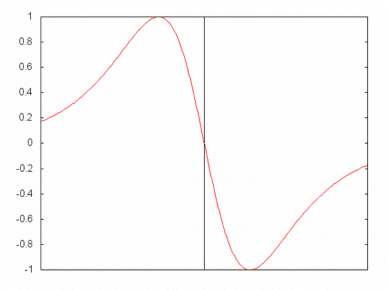
There are two types of edge detectors. Differential operators look at the gradient, or first derivative, of intensity across an image. An edge is found where the gradient is large, either positive or negative. The size of the kernel determines the breadth of edges that will be marked. Differential operators are usually balanced so that their response to a uniform field is 0. They tend to be sensitive to noise, although an optimal filter that combines the differentiation with smoothing can be constructed given a model of the noise. Zero-crossing operators, the second kind, look at the second derivative of the intensity. For a linear step they yield 0 at the edge. This point is more robust as edges get broader; a differential operator will instead see a smaller peak response. They combine a smoothing response with the differentiator to perform better in noise.



Simulated edge



Differential detector



Zero-crossing detector

Simple differential detectors are the Sobol and Prewitt kernels. These are for vertical edges or steps over x. Rotate them by 90 degrees for horizontal edges or steps over y. The values in the kernels are different approximations of the differentiation.

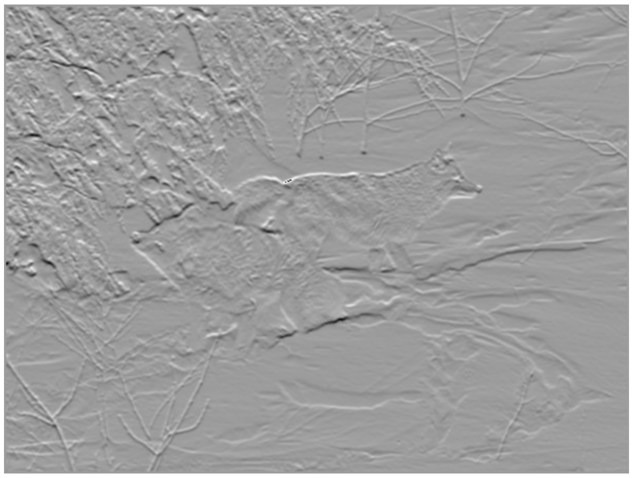
```
      Sobol        Prewitt

     -1 0 1        -1 0 1
     -2 0 2        -1 0 1
     -1 0 1        -1 0 1
```

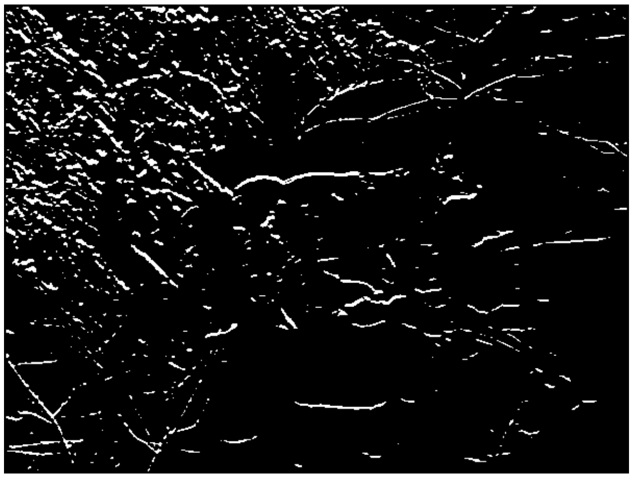
As an example, let's apply the Sobol operator to two pixel neighborhoods, one with a step and one that's uniform with noise. Remember that we aren't doing a matrix multiplication, rather a pixel-by-pixel multiplication followed by a sum. We're ignoring the outer rows and columns as the kernel doesn't full cover them. Usually the result is compared against a threshold, with pixels whose absolute sum exceeds it being marked as edges. In the example the threshold is set at 10, and only the pixels at the step are marked.

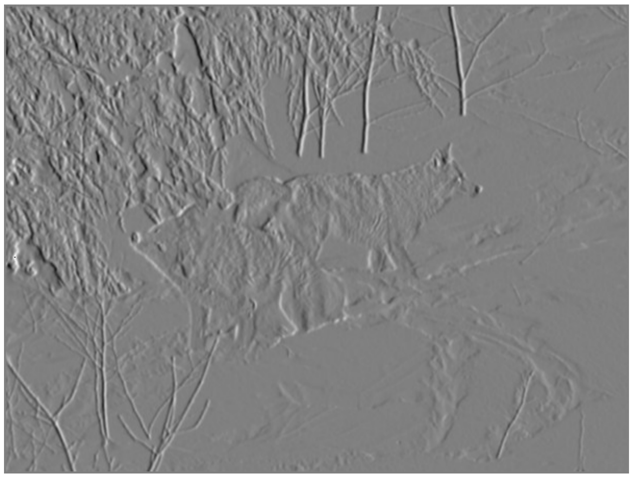
```
          data             kernel             result                threshold

 10 10 10 10  5  5  5                 .   .   .   .   .   .   .      . . . . . . .
 10 10 10 10  5  5  5                 .   0   0 -20 -20   0   .      . . . + + . .
 10 10 10 10  5  5  5      -1 0 1     .   0  -5 -20 -15   0   .      . . . + + . .
 10 10 10  5  5  5  5  *   -2 0 2  -> .   0 -15 -20  -5   0   .  ->  . . + + . . .
 10 10 10  5  5  5  5      -1 0 1     .  -5 -20 -15   0   0   .      . . + + . . .
 10 10  5  5  5  5  5                 . -15 -15  -5   0   0   .      . + + . . . .
 10  5  5  5  5  5  5                 .   .   .   .   .   .   .      . . . . . . .



 10  9 11 10 11  9  8                 .   .   .   .   .   .   .      . . . . . . .
 11 10 11 10 11 10 10                 .   2   2   0  -2  -6   .      . . . . . . .
  9 11 10 12 10 11  9      -1 0 1     .   1   2   0  -2  -2   .      . . . . . . .
 12 10 11 10 11 10 12  *   -2 0 2  -> .  -1  -2   0   2   2   .  ->  . . . . . . .
 10 11 10  8 10 11 11      -1 0 1     .   1  -6   0   6   2   .      . . . . . . .
  9 10 11 10 11 10 10                 .   1  -3   3   2  -2   .      . . . . . . .
 11 10  8 10 11  9 10                 .   .   .   .   .   .   .      . . . . . . .
```

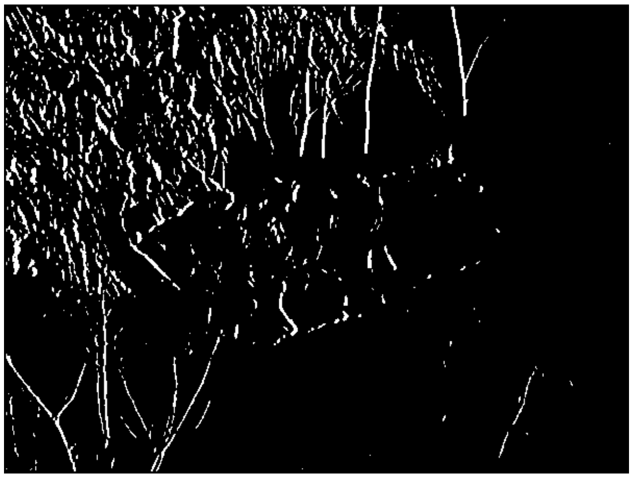
The filter applied to an image captures the images, although there are many false edges, many are missed, and some diagonal lines appear in both directions. More sophisticated strategies than a binary threshold are often used to improve the result. A Canny edge detector, for example, uses a high threshold to select only the strongest edges as seeds, and then extends them into adjacent pixels that are above a lower threshold.

Original image (not provided)


Raw Sobol filter for horizontal edges


Sobol filter after thresholding, horizontal edges


Raw Sobol filter for vertical edges


Sobol filter after thresholding, vertical edges

The Laplacian of Gaussian is a common zero-crossing detector. The Gaussian is a smoothing function.

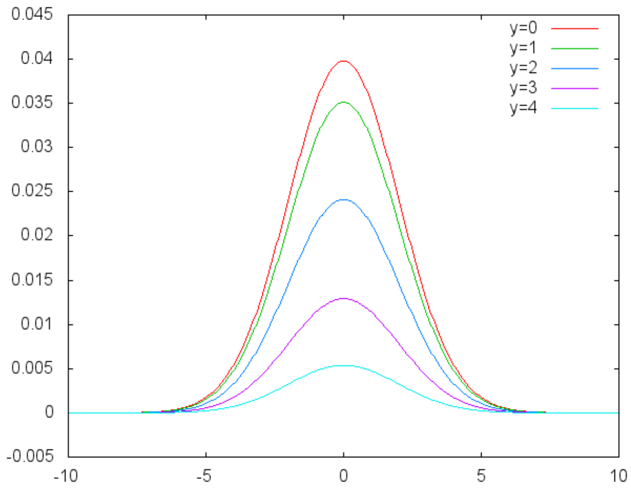
$$G(x,y) = \frac{1}{2\pi s^2} \; e^{-(x^2+y^2)/2s^2}$$

This is a normal distribution where s is the standard deviation, assumed equal for both axes, and the mean is 0. The factor 1/(2*pi*s**2) is to normalize the integral. The Laplacian is the differentiator

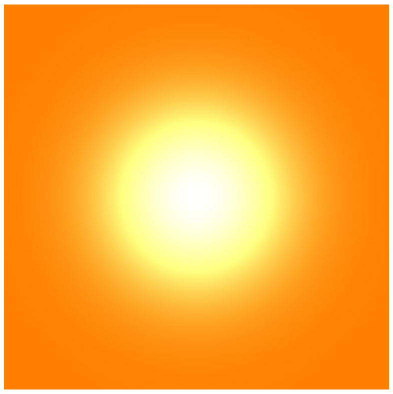
$$L = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Together they give

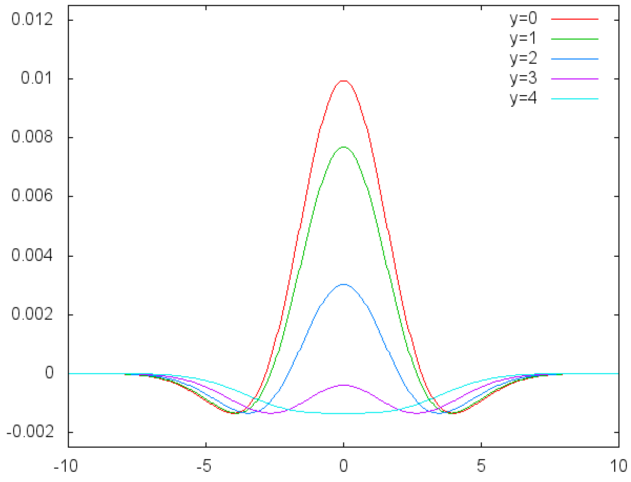
$$LoG(x,y) = \frac{1}{2\pi s^4} \; \frac{x^2+y^2-2s^2}{2s^2} \; e^{-(x^2+y^2)/2s^2}$$



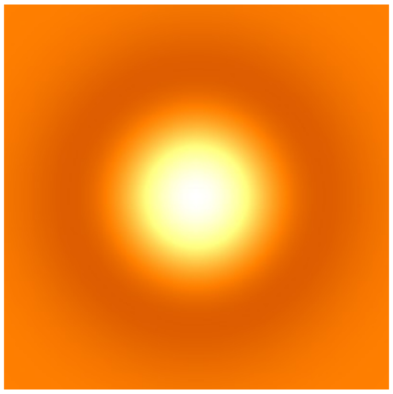
Gaussian cross-sections along x for given y



Heat map of Gaussian function



Laplacian of Gaussian cross-sections along x



Heat map of LoG function

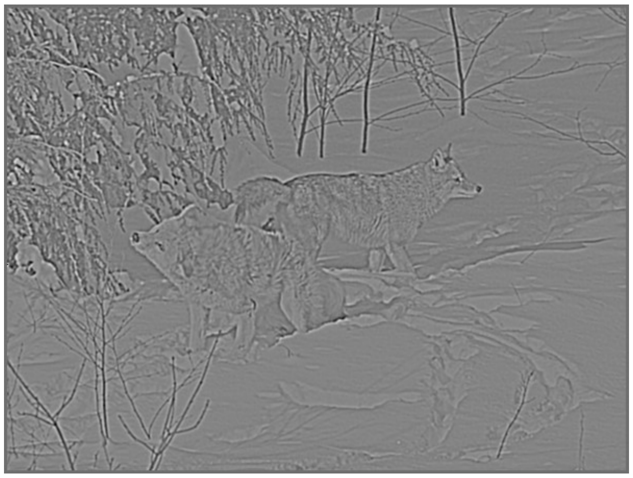
For example, for s = 2.0 and an 11x11 kernel, after normalizing the matrix values so they sum to 1.0 and scaling so that the center value is 100, our kernel becomes

```
  -2  -3  -5  -7  -9 -10  -9  -7  -5  -3  -2
  -3  -6 -10 -13 -14 -14 -14 -13 -10  -6  -3
  -5 -10 -14 -13  -8  -5  -8 -13 -14 -10  -5
  -7 -13 -13   0  20  30  20   0 -13 -13  -7
  -9 -14  -8  20  58  77  58  20  -8 -14  -9
 -10 -14  -5  30  77 100  77  30  -5 -14 -10
  -9 -14  -8  20  58  77  58  20  -8 -14  -9
  -7 -13 -13   0  20  30  20   0 -13 -13  -7
  -5 -10 -14 -13  -8  -5  -8 -13 -14 -10  -5
  -3  -6 -10 -13 -14 -14 -14 -13 -10  -6  -3
  -2  -3  -5  -7  -9 -10  -9  -7  -5  -3  -2
```

and for s = 1.5

```
  -1  -1  -1  -1  -2  -2  -2  -1  -1  -1  -1
  -1  -1  -2  -5  -7  -8  -7  -5  -2  -1  -1
  -1  -2  -6 -11 -14 -14 -14 -11  -6  -2  -1
  -1  -5 -11 -14  -5   4  -4 -14 -11  -5  -1
  -2  -7 -14  -4  35  62  35  -4 -14  -7  -2
  -2  -8 -14   4  62 100  62   4 -14  -8  -2
  -2  -7 -14  -4  35  62  35  -4 -14  -7  -2
  -1  -5 -11 -14  -5   4  -4 -14 -11  -5  -1
  -1  -2  -6 -11 -14 -14 -14 -11  -6  -2  -1
  -1  -1  -2  -5  -7  -8  -7  -5  -2  -1  -1
  -1  -1  -1  -1  -2  -2  -2  -1  -1  -1  -1
```

Notice that the kernels are symmetric, but that they do not sum to 0. This means that the output signal will have a bias. This can be removed by subtracting a local average. To use a zero crossing detector we compare each pixel after the convolution with its neighbors. If the signs in a pair differ, then there is a zero between them that can be marked as an edge.


Raw Laplacian of Gaussian filter

Laplacian of Gaussian with zero crossings marked

A non-symmetric version of this filter offers some interesting possibilities. If the central region does not form a dot then the filter will respond to elongated features. If it is not rotationally symmetric, then it responds to oriented features. The Gabor filter modifies the Gaussian. It first introduces separate scaling parameters (sigmas) for each axis. If the scales are the same the central region is a dot, but as they grow unequal it distorts into an oval. Second, it rotates the coordinate plane through an angle; this is just a rigid mapping on the x and y coordinates. Finally, it multiplies the exponential with a sinusoid with a wavelength and phase offset. This introduces lobes that can be symmetrically or anti-symmetrically placed along or aside the central axis.

The filter takes five parameters: theta, the angle of rotation; sclx and scly, the sigmas for the exponential; lambda, the wavelength of the sinusoid (which will be called wavelen in the code since lambda is a reserved word); and phi, the phase of the sinusoid.

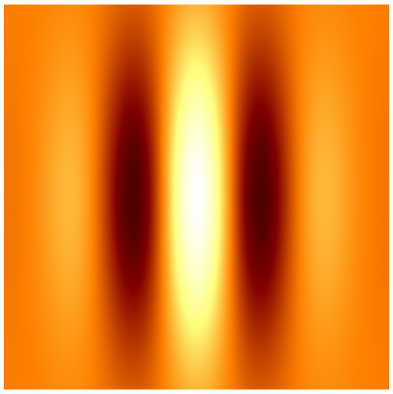
$$x' = x\cos(\theta) + y\sin(\theta)$$

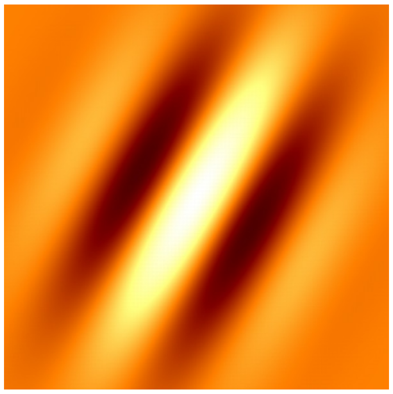
$$y' = -x\sin(\theta) + y\cos(\theta)$$

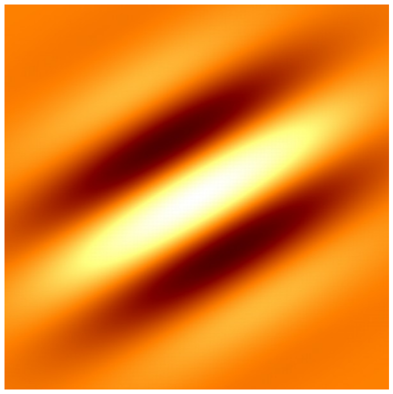
$$Gabor(x,y) = e^{-((x'/sclx)^2 + (y'/scly)^2)/2}\ \cos((2\pi x'/\lambda) + \phi)$$

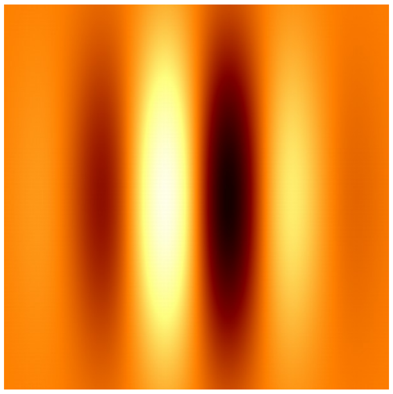
where we have dropped the normalization constant.

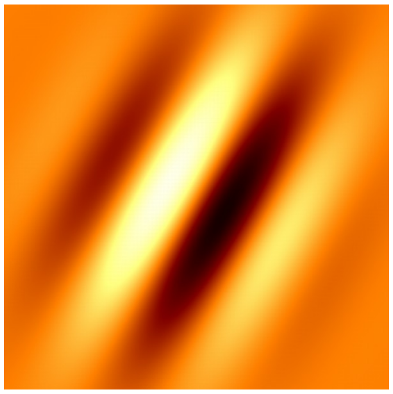
A heat map shows the strength of the kernel components, with dark being negative, red neutral, and white positive. These maps show the unrotated filter and at 30 and 60 degrees for the symmetric and asymmetric cases. Because the sinusoid is a cosine along x, if sclx > scly an angle of 0 degrees is oriented vertically.

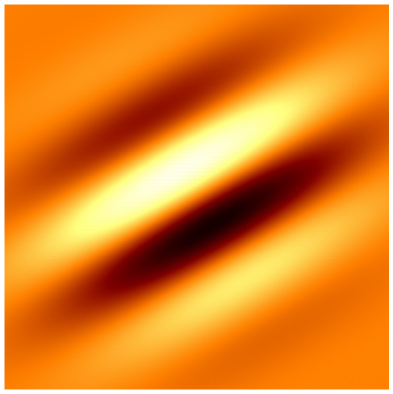

Symmetric Gabor, theta 0, phi 0


Symmetric, theta 30, phi 0


Symmetric, theta 60, phi 0


Asymmetric, theta 0, phi 90


Asymmetric, theta 30, phi 90


Asymmetric, theta 80, phi 90

Our programs in this section will implement this function and its convolution with an image. In practice we can reduce the number of parameters. We need scly >> sclx to stretch the response, with sclx = 2.8 being a common figure for Gaussians and scly being on the order of the 0.5 - 1.5 X the filter size. The size determines the angular resolution, with 13x13 being the minimum if we want to vary theta by 10 degrees; 23x23 gives the best resolution before the gains taper off. phi is either 0 for the symmetric case or 90 for the asymmetric (ie. the cosine becomes a sine). The wavelength is about the size of the filter to detect steps. If it is half or smaller you'll see a positive and negative lobe on each side of the center line and the filter will respond to thin lines.

The plot_kernel.chpl program generates these heat maps. You might find it helpful for visualizing the kernel, although we won't discuss it. The logic is fairly straightforward: scale the coordinates to the filter space, evaluate the filter at each pixel and color code the result. The program takes each of the parameters as command-line arguments, ie.

```
plot_kernel --theta=<degrees> --sclx=<#> --scly=<#> --wavelen=<#>
            --phi=<degrees>
```

Reasonable defaults have been provided. The `--outname` argument lets you specify the PNG file to create (the default is filter.png). You can compile the program with

```
make plot_kernel
```

The heat maps above were based on a 13x13 kernel drawn/sampled over a 700x700 image, with sclx = 2.8, scly = 4.0, and a wavelength of 5.0. The last two parameters differ from our preferred values of scly = 6.0 and wavelen = 12.0 and were chosen to demonstrate the kernel's features clearly.

Subdomains and Subranges

In the previous chapter we mentioned slicing and subranges, where you define a second range as a sub-set of a first. The same can be done for domains. Besides the clarity of intent, the main benefit subdomains offer is the provable validity of array accesses, which may let us avoid bounds checks or help the compiler with optimization, although neither is currently done by Chapel.

Just as a subrange is created by putting a second range specification in brackets or parentheses after the base, so too domains. Provide one range for each rank in the domain. The result is an intersection of the two. Here is where open-ended ranges, ones without either a start or and end, become useful, as they don't constrain one side of the base. The result will be a range with one endpoint taken from the base and the other from the second, whichever end was known.

Say we have a 2D domain representing an image array

```
    Aimg : domain(rank=2) = { 0..ncol-1, 0..nrow-1 };
```

Let a kernel have a "diameter" of `size` x `size`, where `size` is odd. The radius `r = (size-1)/2`, which means that it extends `r` pixels on either size of a center point (x, y):

```
    Akernel : domain(rank=2) = { x-r..x+r, y-r..y+r }
```

Then we can form a subdomain of `Aimg` that is big enough for the kernel. We can do this either by repeating the subrange

```
    Aconv = Aimg[x-r..x+r, y-r..y+r]
```

or by using the kernel area to provide the subrange

```
    Aconv = Aimg[Akernel]
```

Since the second version doesn't repeat the subranges, it's the better choice.

You can declare a subdomain by referencing the parent.

```
    var Aconv : subdomain(Aimg);
```

You can change the indices generated by the subdomain with the domain functions as long as they're also valid in the parent.

Chapel provides many ways of working with domains, ranges, and arrays since they're such fundamental data structures. Let's look at examples for each. The source can be found in ex_rangeop.chpl. Compile and run it with

```
    make ex_rangeop

    bin/ex_rangeop
```

Range Commands

To demonstrate subranges and see the effect of steps, alignments, and counts, we'll start with a base range that is a simple sequence from 1 through 10.

```
var rng1 = 1..10;
```

We can use a `for` expression to print the members

```
writeln("base range ", rng1, " = ", for i in rng1 do i);
> 1..10 = 1 2 3 4 5 6 7 8 9 10
```

If we apply a subrange with an open start to `rng1` we effectively change the end point

```
writeln(rng1[..8], " = ", for i in rgb1[..8] do i);
> 1..8 = 1 2 3 4 5 6 7 8
```

Similarly, an open end would only change the start point. A new step applies to the sub-sequence, and a new alignment can shift the first generated value.

```
writeln(rng1[2.. by 3], " = ", for i in rng1[2.. by 3] do i);
> 2..10 by 3 = 2 5 8

writeln(rng1[3.. by 3 align 2], " = ", for i in rng1[3.. by 3 align 2] do i);
> 3..10 by 3 align 2 = 5 8
```

Here the align 2 would make 2 be the first generated value (ie. 2, 5, 8, 11, ...; also 2, -1, -4, -7 ...), which is incremented by the step until it falls within the start and end points.

If the subrange has no start point, then its alignment is unknown and this propagates to the result. A bad alignment means the range cannot be used for iteration, and the program will halt with an error message. Therefore, in this next example we cannot use the `for` expression to print values, because they cannot be generated. If we provide the alignment, however, then it works.

```
writeln(rng1[..8 by 3]);
> 1..8 by 3

writeln(rng1[..8 by 3 align 2], " = ", for i in rng1[..8 by 3 align 2] do i);
> 1..8 by 3 align 2 = 2 5 8
```

If we provide a count of the number of elements to generate, then this is done on the subrange. The order of the qualifiers is important, as they are applied from left to right. In the first example we step by 2 through the three counted elements 3, 4, 5, giving 3 and 5. In the second we step by 2 to the original end point, so 3, 5, 7, 9, and take the first three of those.

```
writeln(rng1[3.. # 3 by 2], " = ", for i in rng1[3.. # 3 by 2] do i);
> 3..5 = 3 5

writeln(rng1[3.. by 2 # 3], " = ", for i in rng1[3.. by 2 # 3] do i);
> 3..10 = 3 5 7
```

We can also do addition and subtraction to change the start, end, and align parameters.

```
writeln(rng1 + 2, " = ", for i in rng1+2 do i);
> 3..12 = 3 4 5 6 7 8 9 10 11 12
```

```
writeln(rng1 - 2, " = ", for i in rng1-2 do i);
> -1..8 = -1 0 1 2 3 4 5 6 7 8
```

Although we haven't said it explicitly, the range bounds can be negative. We'll use this when defining the Gabor kernel.

Range objects support many methods for adjusting their parameters and testing for membership. The members themselves are accessible: `low` the start point, `first` the first generated element, `high` the end point, `last` the last generated element, `stride` the step, `alignment` the number that must be part of the unbounded sequence, and `length` or `size` for the number of values the sequence will generate.

The `member()` method tests if a range generates a value or if all values of a subrange are included in the base.

```
writeln(rng1.member(9));
> true

writeln(rng1.member(11));
> false

writeln(rng1.member(2..5));
> true

writeln(rng1.member(2..11));
> false

writeln(rng1.member(2..));
> false

writeln(rng1.member(..8));
> false
```

For the methods that adjust the range, let's use another base sequence where we've provided a stride and alignment.

```
var rng2 = 1..20 by 4 align 3;
writeln(for i in rgb2 do i);
> 3 7 11 15 19
```

The `expand()` method adds its argument to the end point and subtracts it from the start. You can see this in how Chapel prints the range to the left of the equals sign.

```
writeln(rng2.expand(2), " = ", for i in rng2.expand(2) do i);
> -1..22 by 4 = -1 3 7 11 15 19
```

The `exterior()` method creates a new range that begins immediately after the base and whose end point is shifted by the positive argument. If the argument is negative then the new range lies before the base and the start point shifts.

```
writeln(rng2.exterior(3), " = ", for i in rng2.exterior(3) do i);
> 21..23 by 4 align 3 = 23

writeln(rng2.exterior(-3), " = ", for i in rng2.exterior(-3) do i);
> -2..0 by 4 align 3 = -1
```

That is, for a positive argument modifying the base range `start..end` we have a new series `(end+1)..(end+arg)`, and the stride and alignment do not change. For a negative argument the new series runs from `(start-arg)..(start-1)`. We have in effect added the points just outside the base range, as you can see in the change in the start and end points. Because we have an alignment restriction, though, we generate only one of the three values.

The `interior()` method with a positive argument picks that many values at the end of the sequence, or at the start with a negative argument. That is, the new series becomes `end-arg+1..end` and `start..start+arg-1`, respectively, and again the stride and alignment do not change.

```
writeln(rng2.interior(3), " = ", for i in rng2.interior(3) do i);
> 18..20 by 4 align 3 = 19

writeln(rng2.interior(-3), " = ", for i in rng2.interior(-3) do i);
> 1..3 by 4 align 3 = 3
```

As with the exterior example, we're taking the three innermost points at either end but the alignment condition causes only one of them to be generated.

The `offset()` method changes the alignment. The argument is added to the current alignment, and the result is taken modulo the step.

```
writeln(rng2.offset(3), " = ", for i in rng2.offset(3) do i);
> 1..20 by 4 align 2 = 2 6 10 14 18

writeln(rng2.offset(-3), " = ", for i in rng2.offset(-3) do i);
> 1..20 by 4 align 0 = 4 8 12 16 20
```

For the first example the new alignment is (3 + 3) % 4 = 2, and for the second (3 - 3) % 4 = 0. Although 0 is outside the start and end values, it is the anchor for the series of step 4.

The `translate()` method shifts the whole sequence: start, end, and alignment.

```
writeln(rng2.translate(3), " = ", for i in rng2.translate(3) do i);
> 4..23 by 4 align 2 = 6 10 14 18 22

writeln(rng2.translate(-3), " = ", for i in rng2.translate(-3) do i);
> -2..17 by 4 align 0 = 0 4 8 12 16
```

Domain Commands

In general we access the indices that are part of a domain by simply using the domain's name. We can do this in a `for` loop or one of its variants (`forall`, `coforall`) and the loop will iterate over each index tuple that is part of the domain. Let's set up a small domain to see this. Again we can use a for expression to print the its members. You'll see that the second rank varies the most; it is the innermost loop.

```
dom1 = { 1..3, 4..6 }
writeln(dom1, " = ", for i in dom1 do i);
> {1..3, 4..6} = (1, 4) (1, 5) (1, 6) (2, 4) (2, 5) (2, 6) (3, 4) (3, 5) (3, 6)
```

Chapel will automatically iterate a domain over all its values if you provide it as an argument to a function that takes a tuple to hold its indices. The return value of such a function can be assigned to an array. The argument needs to be defined either as a tuple, `fn(ind : 2*int)`, or destructured but with the type itself as a tuple, fn((x, y) : (int, int)).

```
    proc domfn1((x, y) : (int, int)) {
      writeln(x, " + ", y, " = ", x+y);
      return x+y;
    }
    var arr1 : [dom1] int;
    var arr2 : [dom1] int;

    arr2 = domfn1(dom1);
    > 2 + 4 = 6
    > 3 + 4 = 7
    > 2 + 5 = 7
    > 3 + 5 = 8
    > 2 + 6 = 8
    > 3 + 6 = 9
    > 1 + 4 = 5
    > 1 + 5 = 6
    > 1 + 6 = 7

    writeln("arr2");
    writeln(arr2);
    > arr2
    > 5 6 7
    > 6 7 8
    > 7 8 9
```

Note that the write functions support a pretty-printer for arrays. All pairs of x and y indices are passed to domfn1 (although not necessarily in order, since Chapel automatically makes the calls in parallel), and the sum of the coordinates is assigned to the corresponding element of array arr2.

We said that you cannot use ranges with strides in domains. Instead, you can give the domain itself a stride using the by keyword. The stride applies to all ranks.

```
    writeln(dom1 by 2, " = ", for i in dom1 by 2 do i);
    > {1..3 by 2, 4..6 by 2} = (1, 4) (1, 6) (3, 4) (3, 6)
```

Similarly, you can specify the count per range. This takes a tuple for each rank.

```
    writeln(dom1 # (2, 1), " = ", for i in dom1 # (2, 1) do i);
    > {1..2, 4..4} = (1, 4) (2, 4)
```

As with ranges, domains support the expand(), translate(), interior(), and exterior() methods. All return new domains, which means they can be chained together. They do not necessarily generate subdomains, however. The arguments are generally tuples that represent the changes along each rank. We'll start by defining a new base domain before applying them.

```
    var dom2 = { 1..10, 20..30 };
    writeln(dom2);
    > {1..10, 20..30}
```

```
writeln(dom2.expand(5,3));
> {-4..15, 17..33}

writeln(dom2.expand(5,3).expand(-2,-1))
> {-2..13, 18..32}

writeln(dom2.translate(4,5))
> {5..14, 25..35}

writeln(dom2.exterior(2,3))
> {11..12, 31..33}

writeln(dom2.exterior(-2,-3))
> {-1..0, 17..19}

writeln(dom2.interior(2,3))
> {9..10, 28..30}

writeln(dom2.interior(-2,-3))
> {1..2, 20..22}
```

You can find these examples in ex_domainop.chpl. Compile and run them with

```
make ex_domainop

bin/ex_domainop
```

Array Commands

The domain underlying an array can be used as an iteration source for loops or iterated operations, as we saw above. The loops generate the values of the array, not the indices. Use `array.domain` to get the indices. Slicing also applies.

Arrays can be operated upon by scalar values, in which case the operation applies at each index. Arrays of the same size and indices may also be combined or set equal, which copies the values over.

```
var dom1 = {1..3, 4..6};
var arr1 : [dom1] int;
var arr2 : [dom1] int;
var arr3 : [dom1] int;

arr1 = 3;
writeln(arr1);
> 3 3 3
> 3 3 3
> 3 3 3

arr1 += 5;
writeln(arr1);
> 8 8 8
> 8 8 8
> 8 8 8

arr2(1,4) = 1; arr2(1,5)=2; arr2(1,6)=4;
```

```
arr2(2,4) = 2; arr2(2,5)=4; arr2(2,6)=8;
arr2(3,4) = 1; arr2(3,5)=8; arr2(3,6)=2;
writeln(arr2);
> 1 2 4
> 2 4 8
> 1 8 2
```

```
arr3 = arr1 / arr2;
writeln(arr3);
> 8 4 2
> 4 2 1
> 8 1 4
```

```
arr3 = for (x, y) in arr3.domain do x+y;
writeln(arr3);
> 5 6 7
> 6 7 8
> 7 8 9
```

We can declare a variable as an alias to an array using the `reindex()` method. The domain declaration for the alias is mapped to the domain of the source, and must be smaller than the source. For example, if we want to shift a kernel centered about 0 to a point (x,y) in an image we can do

```
var kernel : [-r..r, -r..r] real;
var shift = kernel.reindex({y-r..y+r, x-r..x+r});
```

The indices for shift are now centered about (x, y). The re-indexed domain must have the same number of elements as the range provided as the argument. Here we're swapping the x and y coordinates because our image arrays are in this order.

Just to emphasize: if we start with the domain behind an array and take slices we are sure to have valid indices. You can go out of bounds if you manually calculate them.

You'll find these samples in ex_arrayop.chpl. Compile and run them with

```
make ex_arrayop

bin/ex_arrayop
```

Gabor Filters (gabor_v*.chpl)

We're now ready to code our filter. We'll try several different approaches to looping and defining subdomains to see which feels best.

To facilitate working with subdomains, we've added an extra constructor to the `clrimage` class. You'll find it in ip_color_v2.chpl. This version takes an existing image and copies the domain but not the data. This way we'll be sure that both access the same underlying representation. We'll use it so the input image and convolution result have the same domain. This constructor overloads the previous, and Chapel will call the right version depending on whether you provide two integers as arguments, the width and height, or another image.

```
class clrimage {
  var ncol : int;                          /* width (columns) of image */
```

```
    var nrow : int;                      /* height (rows) of image */
    var npix : int;                      /* number pixels = w * h */
    var area : domain(rank=2);           /* array bounds by x and y */
    var c1 : [area] real;                /* first color plane (L, H, Y) */
    var c2 : [area] real;                /* second color plane (A, S, U) */
    var c3 : [area] real;                /* third color plane (B, V, V) */

    proc clrimage(w : int, h : int) {
      ncol = w;
      nrow = h;
      npix = w * h;
      /* This automatically resizes the arrays. */
      area = {0..nrow-1, 0..ncol-1};
    }

    proc clrimage(orig : clrimage) {
      ncol = orig.ncol;
      nrow = orig.nrow;
      npix = orig.npix;
      area = orig.area;
    }
  }
```

The general flow of the program is the same for all versions. It starts by sanity checking the command-line arguments and converts the two angle parameters in degrees to radians. These are constants set at runtime. It then reads the PNG file and generates a greyscale image using either the LAB or YUV space. It next runs the Gabor filter, creating the size x size kernel and convolving it across the image. Finally it scales the result back to an 8-bit image for display and writes it to disk.

The only procedure that changes between the versions is `run_gaborfilter()`.

Since our image domain is in (y,x) order we also swap the indices in the kernel.

```
  proc gabor_kernel(theta : real, sclx : real, scly : real,
                    wavelen : real, phi : real, kernel : [] real) {
    for (y, x) in kernel.domain {
      /* rotated x, y coordinates */
      const xrot = ( x * cos(theta)) + (y * sin(theta));
      const yrot = (-x * sin(theta)) + (y * cos(theta));
      /* square of coordinate after scaling */
      const x2scl = (xrot * xrot) / (sclx * sclx);
      const y2scl = (yrot * yrot) / (scly * scly);

      kernel(y,x) =
        exp(-(x2scl + y2scl) / 2.0) * cos((2.0 * pi * xrot / wavelen) + phi);
    }
  }
```

You'll notice reading the source code that we're using the Time module. This provides timers with up to microsecond resolution for testing how long parts of the program take to run. We'll use the times to compare the five approaches in the Performance section. You simply call `start()` and `stop()` methods on each timer, and use `elapsed(TimeUnits.milliseconds)` to get the interval between them.

Compile the programs with

```
make gabor_v[123456]
```

The program takes many possible command line arguments:

```
--inname=<file>        PNG file to read
--outname=<file>       PNG file to write result to
--space=[LAB|YUV]       which color space to use for greyscale conversion
--size=<int>            kernel diameter, must be odd
--theta=<real>          rotation angle of filter, in degrees
--sclx=<real>           scaling parameter for x axis
--scly=<real>           scaling parameter for y axis
--wavelen=<real>        wavelength of sinusoid
--phi=<real>            phase offset of sinusoid, in degrees
```

Only `--inname` and `--outname` must be provided, the others have default values. A typical run with the default filters would be

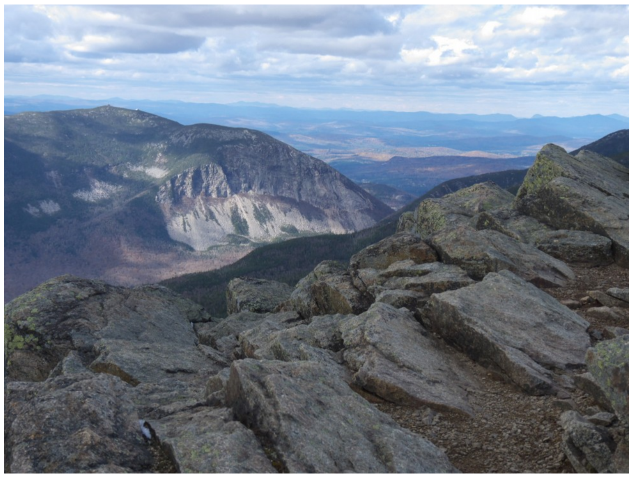
```
bin/gabor_v1 --inname=francnotch.png --outname=g1.png
```

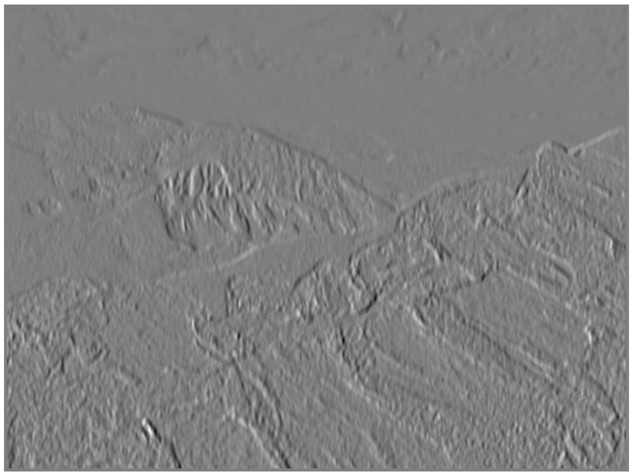
By the way, since there are so many parameters, Chapel has the option of placing them in a file and using it from the command line. Say config.vals contains

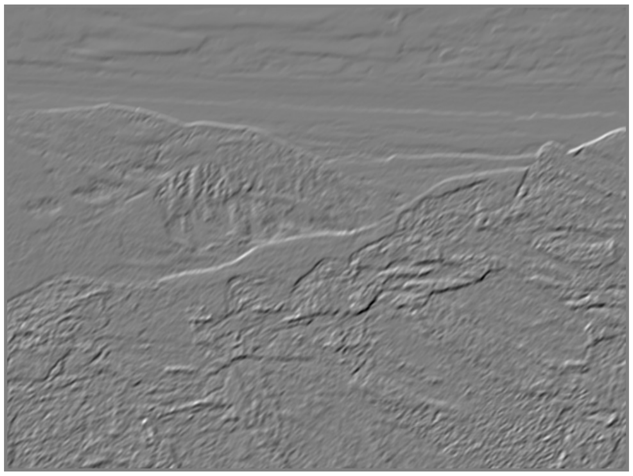
```
size=15
theta=45.0
wavelen=14.0
```

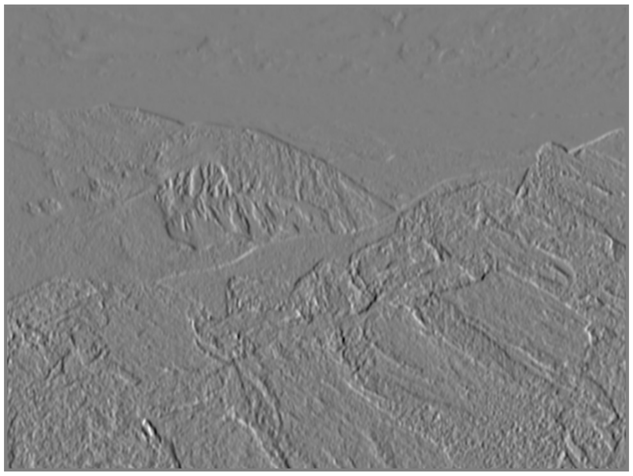
Then you can pass the file with the `-f` flag, not needing to put these three on the command line.

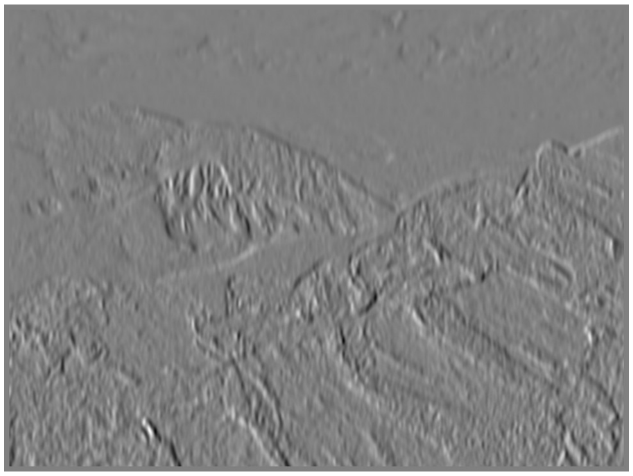
```
bin/gabor_v1 -f config.vals --inname=francnotch.png --outname=g1.png
```

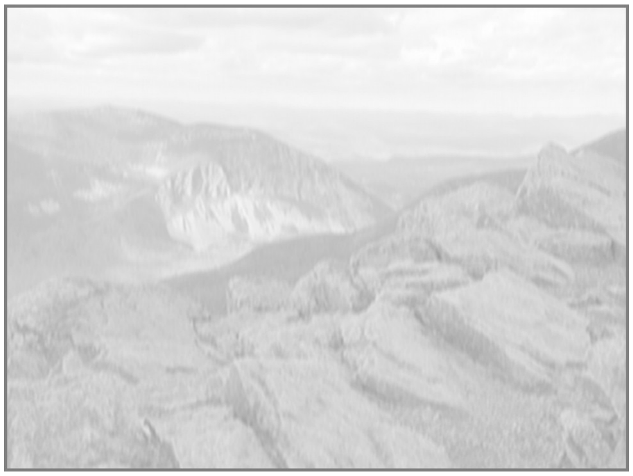

Original image francnotch.png


Default Gabor filter


Gabor theta=45, with strong lines up and right


Gabor, scly=2.8, with sharper features


Gabor size=21 wlen=20, blurred, stronger response


Gabor phi=0, with DC offset

Indexing with Offsets (gabor_v1.chpl)

Our first convolver is a straightforward sum using index arithmetic to center the kernel over each pixel. We define two domains. The first is for the kernel

```
const r = (size - 1) / 2;
const Akern : domain(2) = { -r..r, -r..r };
var kernel : [Akern] real;
```

where `r` is the radius or half-width and the domain is centered at (0, 0). The second is for the interior of the image so that the kernel falls completely inside.

```
const Ainside = img.area.expand(-r, -r);
```

where `img` is the greyscale input. We could have also used a subrange

```
const Ainside = img.area[r..img.ncol-r-1, r..img.nrow-r-1];
```

but the expand method better explains the intent of what we're doing.

We set up the kernel with

```
gabor_kernel(theta=theta, sclx=sclx, scly=scly, wavelen=wavelen, phi=phi,
             kernel = kernel);
```

which populates the kernel array following the Gabor function definition.

We initialize the result to 0 using an implicit iteration over the array's bounds

```
gabor.c1 = 0.0;
```

Next we have a nested loop. The outer `for` covers each pixel in the inner subdomain, while the inner covers each kernel element and the underlying pixel. The coordinate and kernel indices are in the correct order both in the loops and to the arrays.

```
for (y, x) in Ainside {
  for (kj, ki) in Akern do
    gabor.c1(y,x) += img.c1(y+kj,x+ki) * kernel(kj,ki);
}
```

Because we can use negative values in our ranges and have centered the kernel at (0,0), the source pixel in `c1` is just the center pixel shifted by the kernel index. We only use the `c1` plane because both the LAB and YUV transforms put the luminance image there.

One danger of this approach is that we can generate illegal indices for `img.c1` if the kernel ever goes outside the image. In other words, we depend on defining `Ainside` correctly. If we contracted `img.area` by less than `r`, then either `x+ki` or `y+kj` would go out of bounds, and the program would crash. We'll next use subdomains to make sure this can't happen.

Indexing with Zippers (gabor_v2.chpl)

Our second version will slice the image area by the kernel at each pixel before starting the inner loop. This guarantees that the indices will be valid. Our definitions for the `Akern` and `Ainside` domains don't change, nor does the kernel array definition. Using Ainside here doesn't mean we're trying to avoid going out of bounds, but that we don't have to worry about partial overlaps.

We'll set up a separate subdomain for the convolution as a variable since we'll be changing it at each pixel

```
var Aconv : subdomain(img.area);
```

We need two sets of indices when we do the multiplication in the inner loop, one in the image and one in the kernel. The way to combine multiple iterators into one is with a zipper. It takes any number of ranges or iterators and with each step of the loop gets the next value of each, returning all in a tuple. Each component must have the same length.

```
zip(Aconv, Akern)
```

will produce one matching index (which is a tuple) from each domain, and then another, and so on. The zipper generates a tuple of two tuples, so we'll use destructuring to pull out the indices.

First we need to define the `Aconv` subdomain for each pixel. This is just a slice of the image area. Then we generate the coordinates and indices and carry out the multiplication.

```
for (y, x) in Ainside {
  Aconv = img.area[y-r..y+r, x-r..x+r];
  for ((yc, xc), (kj, ki)) in zip(Aconv, Akern) do
    gabor.c1(y,x) += img.c1(yc,xc) * kernel(kj,ki);
}
```

Although it seems that we can still get the `Aconv` subdomain wrong, this is actually safe. We are no longer doing arithmetic on the array indices and slicing guarantees that Aconv is indeed a subdomain and that (xc, yc) will always be in bounds. We do have to use Ainside because if the kernel would overlap pixels outside the image then the sliced area would have a different size than Akern and the zipper would fail.

Indexing with Translations and Zippers (gabor_v3.chpl)

The third version replaces the `Aconv` area with a translation of the kernel to the new pixels. Remember that translations create new copies of the domain, so they don't affect the original. We then zipper the two domains together so that pixel coordinates match the overlying filter, multiply, and add.

```
for (y, x) in Ainside {
  for ((yc, xc), (kj, ki)) in zip(Akern.translate(y,x), Akern) do
    gabor.c1(y,x) += img.c1(yc,xc) * kernel(kj,ki);
}
```

Reductions and Intermediate Results (gabor_v4.chpl)

The fourth variant starts with the zipper of the second. You may remember back in the beginning of this chapter we said that a convolution was a multiplication of every pixel with its kernel element, followed by a sum of those products. This is a reduction, and we'll see more of them in the next chapter when we talk about parallel programming. A reduction takes a set of values and returns one; in this case, a sum. The `reduce` command is an expression with the form

```
    <operation> reduce <data>
```

This variant splits the calculation into these two steps. It uses the same framework as the second variant, with `Aconv`, and defines an intermediate array to hold all the products

```
    var prod : [Akern] real;
```

Our inner loop will populate this array, and once it's done we perform the reduction.

```
    for (y, x) in Ainside {
      Aconv = img.area[y-r..y+r, x-r..x+r];
      for ((yc, xc), (kj, ki)) in zip(Aconv, Akern) do
        prod(kj,ki) = img.c1(yc,xc) * kernel(kj,ki);
      gabor.c1(y,x) = + reduce prod;
    }
```

We shouldn't expect this to be the fastest variant, because the reduction is a second pass through the kernel. However, Chapel will automatically perform it in parallel.

Array Slicing (gabor_v5.chpl)

In the fifth version we'll use array aliasing to shift the indices of the kernel array so it aligns with the image grid. We can then address the kernel with image coordinates. The domain and array definitions are the same as with gabor_v1.

We'll define a variable inside the loop with the alias.

```
    for (y, x) in Ainside {
      const ref shifted = kernel[Akern].reindex({y-r..y+r, x-r..x+r});
      for (kj,ki) in img.area[shifted.domain] do
        gabor.c1(y,x) += img.c1(kj,ki) * shifted(kj,ki);
    }
```

In effect the alias appears to move the kernel above the destination pixel. Notice that we're able to use the kernel indices for the image array. We first tried doing this by shifting the domain using `translate()`, but the problem is that that's a "translation by", that is, an incremental shift in the ranges from the previous position. Since we can't be sure what our previous pixel was, we would need to do some math on the existing range compared to the desired to get the delta. What we would really like is a "translation to" method.

Array Multiplication and Reduction (gabor_v6.chpl)

For the final version we'll try to avoid array indices altogether to do the multiplication. As with the previous version we create an alias called shifted for the kernel that is centered on the current pixel. We will also slice the pixel array over the same area. Since the two arrays now are aligned, we can multiply them directly, letting Chapel iterate over both domains element by element. We'll then do a reduction on the result.

```
    for (y, x) in Ainside {
      const ref shifted = kernel[Akern].reindex({y-r..y+r, x-r..x+r});
      const overlay = img.c1[y-r..y+r, x-r..x+r];
      gabor.c1(y,x) = + reduce (shifted * overlay);
    }
```

Defining overlay as a subdomain of c1 using shifted does not compile

```
    const overlay = img.c1[shifted];     /* no good */
```

Performance

If you've been running these programs as we go along, you will have noticed that the run times vary dramatically. The table gives the average time in milliseconds for the key routines.

	rgb_convert	run_gaborfilter	display_color	
v1	397	15209	79	index math
v2	398	36415	79	domain + zipper
v3	398	35990	79	translate + zipper
v4	399	56795	79	zipper + reduction
v5	398	20469	80	array alias
v6	398	53966	79	array multiplication

These runs were made on the francnotch.png image with the default parameters. The consistency of the LAB conversion at the start and 8-bit conversion of the results gives us confidence in these measurements, as these routines don't change between program versions.

The preliminary conclusion, without going deeper into the compiled code (which is beyond our goals for this exercise), is that zippering iterators hurts performance and that the reductions in gabor_v4 and gabor_v6 are especially costly, despite Chapel doing it internally in parallel. The extra domain created in _v2, compared to the translation in _v3, has been optimized away, leaving the zipper as doubling the time to do the convolution. The reduction adds another 50% to the time. The simplest access pattern of using a single set of indices and offsetting them into the image array, either directly by calculation in gabor_v1 or indirectly by array aliasing in gabor_v5, is best.

The Chapel compiler does have a `--fast` flag that turns off runtime safety checks such as array bounds. If you re-compile the programs with these programs using '`make CHPLFLG=--fast all`', the results are dramatically faster. You can also set the environment variable CHPL_FAST to any non-empty string. This will cause the compiler to use the `--fast` flag for any compilation.

	rgb_convert	run_gaborfilter	display_color	improve base
v1	278	207	6	73 X
v2	278	2346	6	16
v3	282	2249	6	16
v4	278	3344	6	17
v5	278	644	6	32
v6	279	3541	6	15

That's a speed-up of the `run_gaborfilter()` routine by one or two orders of magnitude! As a check, the resulting images do match those generated without the `--fast` flag – we haven't corrupted the operation of the program. The array aliasing in _v5 hasn't helped as much as the direct access in _v1, but these two are 4 to 5 times faster than the zipper and reduction variants. Again we see a 50% penalty for the reductions, and little difference between the zippers.

We're still new to the language so figuring out what is happening here – if we are seeing any optimizations from subdomains, if we're using subdomains properly, and how to improve the run time – isn't something we can do yet. It would be nice if Chapel, with its support for subdomains, could get closer to the raw performance without the compiler flag.

Wrap-Up

In this chapter we've used a convolution example, an area multiply followed by sum, to talk about subranges, subdomains, and array operations. We've looked at six different implementations of the convolver using arithmetic on array indices, subdomains to generate guaranteed correct indices, a multiply-and-reduce approach, and array aliasing. The direct addressing schemes are the fastest. We've learned:

- A subdomain should be used because it is provably a subset of the base domain.

- Ranges, domains, and arrays support slicing, which is using a second range to sub-sample the original. Slicing defines a subdomain.

- Ranges and domains support expanding//contracting and translating their bounds, and generating elements in the exterior just outside the range or in the interior.

- Domains and arrays can be passed as arguments to scalar functions, including built-in operators, or used with other iterators. Chapel will automatically iterate the function over each element of the domain or array.

- Array aliasing is a re-mapping of the indices of an array using the `reindex()` method. With it we can shift a kernel about an image.

If you monitored the programs as they ran, you noticed that they were all serial. We can speed up the performance by taking advantage of Chapel's parallel programming support, which we'll look at next.

Exercises

1. Generate a heat map for a Gaussian and Laplace of Gaussian filter. Define a grid of say 50 pixels per unit in the filter space. Calculate the filter function for each x, y and map it to RGB channels. For example, a heat map that runs from block through red and yellow to white is

```
    r = 2*t     g = 2*t - 0.5     b = 2t - 1
```

where t, the function value, lies between 0 and 1.0. The colors are clipped at 0 and 1, then scaled to an 8-bit range. Store each in an `rgbimage` and save as a PNG. (plot_kernel.chpl contains our solution for the Gabor filter and was used for the figures for this chapter.)

2. Write a Laplacian of Gaussian edge detector. Either save the convolution result as a PNG and highlight values near mid-scale, ie. near 0, or pass the pixels through a threshold marking only those that have a different sign than their neighbor (implying that a zero lies between).

3. An integral image I is one where the value at a pixel (x, y) is the sum of all pixels with smaller coordinates, ie.

$$I(x,y) = \sum_{x'=0}^{x} \sum_{y'=0}^{y} image(x',y')$$

This allows us to quickly calculate sums over areas. For example, if we want to average in a 5x5 square about a pixel, then we would do

```
(I(x+5,y+5)-I(x-5,y+5)-I(x+5,y-5)+I(x-5,y-5)) / 25
```

Write a library routine to generate the integral image, leaving it in a `clrimage`.

4. The asymmetric (phi=0) Gabor filter gives a large background signal. Write a high-pass filter to remove it. What remains?

Files

A tarball with the code and image for this chapter can be found here. A zip file is here.

A PDF version of this text is here.

PARALLEL PROGRAMMING (gabor_par, lab_limits)

Introduction

Chapel offers language support for task-driven and data-driven parallelism. In task parallelism you define actions that can be done independently. These get started in separate threads and synchronization variables control the flow between tasks. For data parallelism you specify a chunk of work to be done and Chapel splits it up into smaller sets, executing each separately. This is often followed by a serial step to combine the individual results, the well-known map-reduce technique, where map describes applying a function to data in parallel and reduce collapses the results.

We have two examples that will serve as good, reasonably simple introductions to these two techniques. We often want to run Gabor filters as a bank, changing the angle of rotation but otherwise running the same filter on the same data. Each filter runs independently; this is a good fit for task parallelism. The second exercise of the Color Conversion chapter asked you to evaluate the L*A*B* conversion for all possible RGB combinations, 256*256*256 = 15,777,216 data points, to find the limits of the transform. We want to evaluate the same `rgb_to_lab()` function at each pixel, and then collapse the results to a single minimum or maximum: data parallelism.

The directory for this chapter is `parallel`. You'll be using the C functions for reading and writing PNG images in img_png_v3.c and img_png_v3.h, and the color library ip_color_v2.chpl. Use `make name` to build the programs, leaving off the '.chpl' suffix. Executables get placed in the `bin` sub-directory, intermediate files (which you should never have to look at) in `build`. The image for this chapter is francnotch.png, the same one used for the Gabor filters.

The programs and image for this chapter are here (as zip file). A PDF version of the text is here.

Data Parallelism (lab_limits)

The basic Chapel statement for operating on data in parallel is `forall`. As with `for`, if followed by `do` it takes a single statement to execute on the data, otherwise braces allow for a block of statements. Chapel controls how the iteration is split: there may be a separate task for each iteration, they may be grouped, or it may even execute completely serially. The statements after the `forall` will only be executed when all iterations are done.

Like `for`, `forall` can be used as an expression, returning a collection of values. You may not break out of a `forall`, nor return.

Conceptually you can think of the body of the loop forming a procedure that can be called in a separate task. Any variables from outside the loop are passed as arguments to that procedure. This isolates it from the world outside the scope of the loop, preventing race conditions. Normally this argument list is without any intents, ie. all variables except arrays and the synchronization types we will talk about with task parallelism are passed as `const in`; the others have type `ref`. You can change the intent by adding a `with` clause to the `forall`

```
forall (y,x) in img.area
  with (ref sum : real, ref cnt : int) {
    ...
}
```

You cannot use the `out` or `inout` intent, as these can cause data races (as can assigning to a ref).

Because arrays are passed by reference, you need to ensure that accesses do not overlap between different iterations. Chapel will not detect these. If we're doing a column sum with

```
var colsum : [0..ncol-1] int;
forall y in 0..nrow-1 do
  for x in 0..ncol-1 do
    colsum[x/2] += img(y,x);
```

both even and odd columns get written to the same array value, ie. `x/2` rounds to 0 for `x = 1` and there will be a race at `colsum[0]`. If you swap the loops, putting `x` in the `forall`, then you may also not be safe because the range may split so that `x = 0` and `x = 1` are in different tasks. It would be safe to use the outer loop index directly, without alteration so that no accesses could collide. Any inner loops are run serially, so it doesn't make sense to embed `forall`s.

As we've said a couple times, the process of turning these intermediate results into a single value is called a reduction. You have a limited number of operators that can be applied: addition `+` and multiplication `*`, logical and `&&` and or `||`, bit-wise and `&`, or `|`, and xor `^`, and `min` and `max`. The format of the `reduce` command is

```
<operator> reduce <iterator>
```

If you need the partial results as you go along, use `scan` instead of `reduce`. Scan returns an array of the same size as what it iterates over, where each value is the cumulative sum of values to that point.

There are two other operators for `reduce` and `scan`: `minloc` and `maxloc`. These take a zipper for the iterator with two ranges, one the data for which we want the minimum or maximum value, the other a “name”. (What we would really like is a version that uses a domain or range and just returns the indices at the extreme value, without the zipper.) The reduction returns a tuple with the minimum or maximum and the corresponding element from the names list.

Consider two examples of (ab)using reduce and scan. The first populates a 1000x1000 image with the square of the distance of the pixel (ie. `x**2 + y**2`) at each point.

```
var img1 : [0..999, 0..999] int;
forall (x,y) in img1.domain do img1(x,y) = (x**2) + (y**2);
```

It then fills a column sum array in parallel using `forall`. We use the domain `.dim()` method, which takes as argument the rank index, to extract the domain range. The domain's first rank is run in parallel, and since the column sum depends on this index the addition is safe. These two loops are equivalent:

```
    var colsumA : [img1.domain.dim(1)] int;
    forall x in img1.domain.dim(1) do
      for y in img1.domain.dim(2) do
        colsumA(x) += img1(x,y);

or  forall (x,y) in img1.domain do
      colsumA(x) += img1(x,y);
```

Reducing this sums all 1,000,000 pixel values.

```
var grandsum = + reduce colsumA;
writeln("grandsum  ", grandsum);
> grandsum  665667000000
```

We can actually write this as a one-liner without using the `colsumA` storage. We use a `reduce` to do the column sum by providing the column as a fixed value in the slice. Wrapping this in a `forall` generates a collection of these sums, one for each column, which the outer `reduce` then collapses to the grand total. This should match the value we just got.

```
var grandsum2 = + reduce (forall x in img1.domain.dim(1) do
                           + reduce img1[x,img1.domain.dim(2)]);
writeln("grandsum2 ", grandsum2);
> grandsum2 665667000000
```

The second example uses `scan` to generate the solution to Exercise 3 of the Gabor Filter chapter, creating an integral image. Scan produces an array with an incremental sum along one axis. To declare an array of arrays, use separate brackets, one for each rank. First we populate a small test image

```
var img2 : [0..4, 0..4] int;
forall (i,j) in img2.domain do
  img2(i,j) = (i*5) + j;
writeln(img2);
> 0 1 2 3 4
> 5 6 7 8 9
> 10 11 12 13 14
> 15 16 17 18 19
> 20 21 22 23 24
```

Then we declare storage for the results of the scan, an array with the same number of columns and rows but split. We use `scan` to get the cumulative sum, element by element, of each column. We'll hard-code the ranges to keep things short, rather than using `domain.dim()`.

```
var colsumB : [0..4][0..4] int;
forall j in 0..4 do colsumB(j) = + scan img2[0..4,j]
for j in 0..4 do writeln(colsumB(j));
> 0 5 15 30 50
> 1 7 18 34 55
> 2 9 21 38 60
> 3 11 24 42 65
> 4 13 27 46 70
```

We need the loop to print the array or Chapel will print it as a 1D list on one line. Note that the values along each row are the cumulative sum of the corresponding column. Because the scan proceeds along a column, it stores the result in rows; we need to transpose the matrix.

```
forall j in 0..4 do
  for i in j+1..4 do
    colsumB[i][j] <=> colsumB[j][i];
```

Now if we do another scan in the opposite direction we get the sum of all those with smaller indices. This is the integral image.

```
var intimg : [0..4][0..4] int;
forall i in 0..4 do
```

```
  intimg(i) = + scan colsumB[i][0..4];
for i in 0..4 do writeln(intimg(i));
> 0 1 3 6 10
> 5 12 21 32 45
> 15 33 54 78 105
> 30 64 102 144 190
> 50 105 165 230 300
```

You'll find this example in ex_datapar.chpl. Compile and run it with

```
make ex_datapar

bin/ex_datapar
```

By the way, Chapel provides a shortcut for `forall` statements. You can put the everything except the `forall` in square brackets before a single statement to iterate over. The first loop above to initialize `img1` could have been written

```
img1 = [(x,y) in img1.domain] (x**2) + (y**2);
```

Limits Found in Serial (lab_limits_ser)

For our initial, serial solution to Exercise 2 of the Color Conversion chapter, we choose to track the extreme values for each color plane and their RGB "position" as we cycle through all the combinations. The code is straightforward but repetitive:

```
var l, l_a, l_b : real;
var lmin = max(real);
var lmax = min(real);
var lminpos, lmaxpos : 3*int;
/* Similarly for A, B. */
lmin = max(real);
lmax = min(real);
for r in 0..255 {
  for g in 0..255 {
    for b in 0..255 {
      rgbpix_to_lab(r : c_uchar, g : c_uchar, b : c_uchar, l, l_a, l_b);
      if (l < lmin) {
        lmin = l;
        lminpos = (r, g, b);
      }
      if (lmax < l) {
        lmax = l;
        lmaxpos = (r, g, b);
      }
      /* Similarly for A, B. */
    }
  }
}
```

`l_a` and `l_b` are used for the A and B planes to avoid confusion with 'b'lue. At the end of the loop we just print the minimum and maximum values and their position:

```
(rpos, gpos, bpos) = lminpos;
writef("  L  %7.2dr (%3i %3i %3i)", lmin, rpos, gpos, bpos);
(rpos, gpos, bpos) = lmaxpos;
writef("  -  %7.2dr (%3i %3i %3i)", lmax, rpos, gpos, bpos);
```

(A quick aside about the formatting codes for `writef()`. The function is similar to C's `printf()`, except that the letter codes per type differ: `%i` `int`, %u `uint`, %r `real`, `%s` `string`. %m for `imag`, and `%z` `complex`. `x` before an integer code prints in hexadecimal, `b` binary, and `o` octal. `d` before a real or imaginary prints the number in decimal format, `e` forces exponential, and otherwise `writef()` picks the most compact. More details are in the modules/standard/IO.chpl file.) We chose to deconstruct the position tuples before printing; alternatively, we could have done

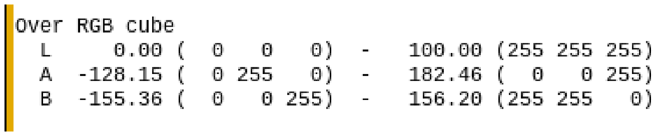
```
writef("  L  %7.2dr (%3i %3i %3i)", lmin, lminpos(1), lminpos(2), lminpos(3));
```

The `for` loop is a serial command. Changing this is what's needed to have work done in parallel.

Compile and run this with

```
make lab_limits_ser

bin/lab_limits_ser
```

The program takes no command-line arguments. As a check, it first prints the white point to confirm L=100.0, A=B=0.0. If you monitor your CPU usage, you'll see that the program operates serially on one core.



```
Over RGB cube
  L     0.00 (  0   0   0)  -   100.00 (255 255 255)
  A  -128.15 (  0 255   0)  -   182.46 (  0   0 255)
  B  -155.36 (  0   0 255)  -   156.20 (255 255   0)
```

Output from the LAB limits program

Parallelism with Local Updates (lab_limits_parv1)

Our first parallel takes the outer `for` loop and changes it to `forall`. This causes the compiler to complain about all the variables the loop uses that are declared outside the loop. To fix this, we'll replace the scalars we used to track the extrema and their location for each plane with arrays. We'll store the limits per `r`, the color we use in the `forall` loop, so there is no problem accessing each value (within a parallel task r is constant).

```
var lmin, lmax : [0..255] real;
var lminpos, lmaxpos : [0..255] 3*int;
/* Similarly for A, B. */
lmin = max(real);
lmax = min(real);
forall r in 0..255 {
  for g in 0..255 {
    for b in 0..255 {
      var l, l_a, l_b : real;
      rgbpix_to_lab(r : c_uchar, g : c_uchar, b : c_uchar, l, l_a, l_b);
      if (l < lmin(r)) {
        lmin(r) = l;
        lminpos(r) = (r, g, b);
      }
      if (lmax(r) < l) {
        lmax(r) = l;
        lmaxpos(r) = (r, g, b);
      }
      /* Similarly for A, B. */
    }
  }
}
```

To get the absolute extrema we reduce over the arrays. We want the value of `l[min|max]pos` corresponding to the minimum/maximum, and zip the value array with the position to extract the location.

```
    (limit, (rlim, glim, blim)) = minloc reduce zip(lmin, lminpos);
    writef("  L  %7.2dr (%3i %3i %3i)", limit, rlim, glim, blim);
    (limit, (rlim, glim, blim)) = maxloc reduce zip(lmax, lmaxpos);
    writef("  -  %7.2dr (%3i %3i %3i)\n", limit, rlim, glim, blim);
    /* Similarly for A, B. */
```

Compile and run this with

```
    make lab_limits_parv1

    bin/lab_limits_parv1
```

If you monitor your CPU you should see multiple cores in use. The run time will also decrease.

Parallelism with Per-Pixel Storage (lab_limits_parv2)

The second version is conceptually cleaner. Rather than tracking intermediate values for each pass of the parallel loop, we store all the points and then do a global reduce at the end to find the extrema and their position. In effect we're doing an LAB conversion on a 256*256*256 image, without using the `clrimage` framework. This costs memory to store the LAB values at each RGB point, and would not be practical for large data sets.

We'll define a domain cube for our RGB volume, and three arrays to hold the converted colors. The loop to populate the arrays is simple

```
    const cube : domain(rank=3) = { 0..255, 0..255, 0..255 };
    var l, l_a, l_b : [cube] real;

    forall (r,g,b) in cube do
      rgbpix_to_lab(r : c_uchar, g : c_uchar, b: c_uchar,
                    l(r,g,b), l_a(r,g,b), l_b(r,g,b));
```

Since each array element is only written by one `r`, `g`, `b` combination, this is safe from race conditions.

The reduction is also simple

```
    (limit, (rlim, glim, blim)) = minloc reduce zip(l, cube);
    writeln("  L  %7.2dr (%3i %3i %3i)", limit, rlim, glim, blim);
    (limit, (rlim, glim, blim)) = maxloc reduce zip(l, cube);
    writeln("  -  %7.2dr (%3i %3i %3i)\n", limit, rlim, glim, blim);
    /* Similarly for A, B. */
```

Compile and run this with

```
    make lab_limits_parv2

    bin/lab_limits_parv2
```

The output will match the other two versions.

Performance

Timing results from 10 runs give

```
    lab_limits_ser     6.89 +/- 0.01 s
    lab_limits_parv1   2.98 +/- 0.02 s     improvement to serial: 2.3x
    lab_limits_parv2  11.51 +/- 0.03 s     degradation to serial: 0.6x
```

The programs were compiled without the `--fast` flag. The small standard deviations show good repeatability. All four CPU cores were used for the two parallel results. The whole-image reductions in _parv2 dominate the performance; you can see the pause between the printing of each limit. The code is certainly simpler than in _parv1, but it comes at a cost of extra memory and run time. The _parv1 improvement over the serial version is smaller than we might expect; typically four threads give a speed-up of 3.0 - 3.5 X. In comparison to the 1.11 (April 2015) release, the parallel versions have slowed by 10-15% whereas the serial version hasn't changed.

If we insert timers and look at the cost of the forall loop and the reductions we find (values in ms)

```
                       forall     reduce
    lab_limits_parv1    2956          2
    lab_limits_parv2    2704       8680
```

Doing the extra work in the loop in _parv1 adds 10% to the timing, but eliminates the cost of the reduction. Note that there are six reductions in either program, so it is taking more than a second to do each over the whole cube. This is despite the reductions themselves being done in parallel – each core remains occupied for the run of _parv2.

Task Parallelism (gaborbank)

Chapel provides three statements to define and launch tasks. The body of each is wrapped in a function, much as `forall` does, with externally declared variables being provided as arguments to that function. The `begin` statement takes a single statement or block and executes it in a "fire-and-forget" fashion. Chapel starts the task in the background and does not wait for it to finish. The `cobegin` statement takes several statements and executes each as a task, waiting until all have finished. The `coforall` statement treats its body as a task that is called for every iteration. Unlike a `forall` loop that may or may not split its interactions into separate tasks, the `coforall` will do so. Each of these keywords can be followed by '`with (<argument list>)`' to give non-default intents to the outside variables, just like `forall`.

```
    begin run_gaborfilter(img, size, theta, sclx, scly, wavelen, phi_rad);
```

runs a Gabor filter asynchronously.

```
cobegin {
  run_gaborfilter(img, size, 0, sclx, scly, wavelen, phi_rad);
  run_gaborfilter(img, size, pi/2.0, sclx, scly, wavelen, phi_rad);
}
```

runs two filters in orthogonal directions in separate tasks. Whether these tasks run in parallel or serial depends on the computing resources available.

```
coforall theta in 0..135 by 45 do
  run_gaborfilter(img, size, DEG2RAD(theta), sclx, scly, wavelen, phi_rad);
```

runs four filters in four separate tasks. As with the other `for` variants, `do` is used if there is only one statement in the loop body; otherwise, use braces around the block.

A `cobegin` can be used when you have a known number of tasks to run in parallel when you're coding. A `coforall` is used when the amount is indeterminate. We'll use `coforall` in the gaborbank program, but to give an example of `cobegin`, let's try writing a parallel sort. The QuickSort algorithm works recursively through an array. At each step it picks an element and splits the array into two, those values that are smaller and those that are bigger. This is called partitioning the array and its result is that the picked element has been placed correctly but the sub-arrays to either side are still unsorted. So, we recur on each of those – and this can be done in parallel. The top level looks like:

```
proc quicksort(data : [] real) {
  var q : int;         /* index of pivot */

  q = partition(data);
  cobegin {
    quicksort(data[..q-1]);
    quicksort(data[q+1..]);
  }
}
```

which says we find the pivot point (which places the `q`'th biggest element at that index) and then recur on the sub-arrays formed by the slices. This recursion happens in parallel thanks to the `cobegin`.

ex_quicksort.chpl implements the algorithm as presented in "Introduction to Algorithms" by Corman et. al. The partition function is:

```
proc partition(data : [] real) : int {
  const pivot = data(data.domain.last);
  var i = data.domain.first - 1;
  for j in data.domain.first..data.domain.last-1 {
    if (data(j) <= pivot) {
      i += 1;
      data(i) <=> data(j);
    }
  i += 1;
  data(i) <=> data(data.domain.last);
  return i;
```

using the swap operator `<=>`. The `domain.first` and `domain.last` members are the bounds of the sub-array, inclusive; there is also `domain.low` and `domain.high` which are limits of the range and can differ from first and last if the alignment is not zero.

The example program populates an array with random numbers, calls quicksort, and then checks that the values do indeed increase across the array. Compile and run it with

```
make ex_quicksort [CHPLFLG="-s nelt=#"]

bin/ex_quicksort
```

You can change the size of the array with the `nelt` config param, passed to the compiler with the make variable `CHPLFLG`. Unlike a runtime value which requires no space between the -s and variable name, the compiler accepts a space.

(If you were to compile this program and run it for a moderate sized array, say of 100,000 elements on our machine, you'd find your console filling with warning messages about memory allocation failures. This seems to be an artefact of the Chapel runtime, since the program will run to completion and pass its verification. At a sufficiently large `nelt` the program segfaults. QuickSort implementations often use a simple insertion sort once the sub-array size gets below some threshold, say 8 or 16 elements; for these small arrays the inefficiency of the insertion sort isn't a problem, and wins over the recursion. The example program contains a config param `--use_insert` to turn this behavior on or off and `--len_insert` to set the threshold. Using the insertion sort decreases the number of allocations the runtime does and increases the array size at which the warnings appear. Another option is to compile with `CHPLFLG=--fast` which turns off the runtime checks and the allocation failure messages.)

In a way `cobegin` and `coforall` are convenience statements that handle the synchronization of all threads at the end of their block. The `sync` command can also do this; it takes a block of statements that must complete before execution continues to the next command. This differs from a `cobegin` in that any task that is created within the scope of the sync, even in subroutines, must finish, while the `cobegin` only requires the statements directly within its block must.

```
sync {
  begin run_gaborfilter(img, size, 0, sclx, scly, wavelen, phi_rad);
  begin run_gaborfilter(img, size, pi/2.0, sclx, scly, wavelen, phi_rad);
}
```

The `serial` command can block a section of code from launching new tasks if a condition holds. This means that the section will execute serially. You can use this to avoid the overhead of launching tasks if there are too few of them. The form of the statement is

```
serial <condition> {
  /* commands, possibly with begin/cobegin/coforall */
}
```

or, if there's only one statement that follows, you can replace the braces with `do`. If you omit the test, then Chapel assumes it has the value true and will always execute the block serially.

As with `forall`, `return`, `break`, and `continue` statements are not allowed inside a concurrent block for any of these commands.

Gabor Filter Bank (gaborbank)

Because the Gabor filter responds most strongly to lines at a given angle, we often want to run several filters with different rotations on the same image. We call this a bank of filters, and the angular resolution depends on the filter's parameters, mostly size and scale.

We want to modify our filter from the last chapter to generate the bank. Without changing the convolution routine itself, we can take advantage of task parallelism do run the filters in parallel. The modified program is called gaborbank, and it can be compiled with

```
make gaborbank
```

It replaces the `--theta` command line argument with `--nrot`, the number of filters in the bank. Filters will be spaced evenly over 180 degrees; the angle wraps, so that from 180 to 360 you get an inverted response. We also delete the `--outname` argument and will save each result as "bank_<rot>.png", where <rot> is the filter rotation in degrees.

Most of the main procedure in the previous version has been pulled into a procedure called `filter_and_save()`. Here we can allocate a new image to store the result independently of other tasks, run the filter, convert the result to 8-bit, and save it to disk. We pass all the global parameters through to the function so it is self-contained. The `theta` parameter is now passed in degrees; for clarity, we've renamed `phi_rad` to emphasize it's in radians.

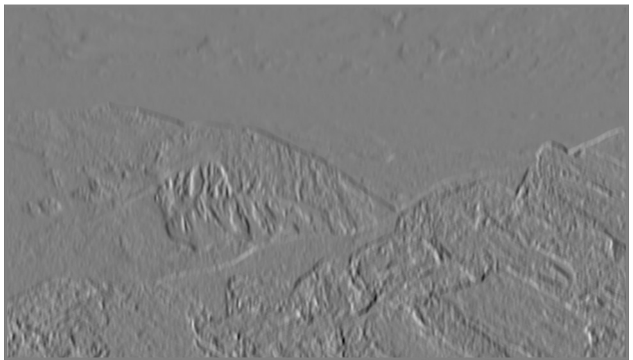
```
proc filter_and_save(img : clrimage, size : int, theta : int, sclx : real,
                     scly : real, wavelen : real, phi_rad : real) : int {
  const theta_rad = pi * theta / 180;
  var gabor : clrimage;
  var grey : rgbimage;
  var gabor2grey : conversion;
  var outname : c_string;
  var retval : int;

  gabor = new_clrimage(img);
  run_gaborfilter(img, gabor, size, theta_rad, sclx, scly,
                  wavelen, phi_rad);

  gabor2grey.how = rgbconvert.CENTER;
  gabor2grey.plane = clrplane.C1;

  retval = display_color(gabor, grey, gabor2grey);
  if (retval < 0) then return retval;

  /* This makes a 3-digit angle without using something like sprintf. */
  if (theta < 10) then outname = "bank_00" + theta + ".png";
  else if (theta < 100) then outname = "bank_0" + theta + ".png";
  else outname = "bank_" + theta + ".png";

  retval = PNG_write(outname, grey, CLR_R);

  free_rgbimage(grey);
```

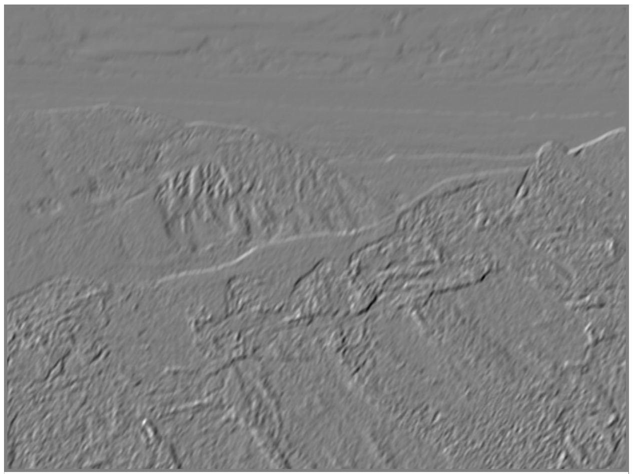
```
    return retval;
  }
```

Our `main()` becomes very simple. It reads the input PNG, does the conversion for the L or Y plane, and then uses a coforall to generate the bank.

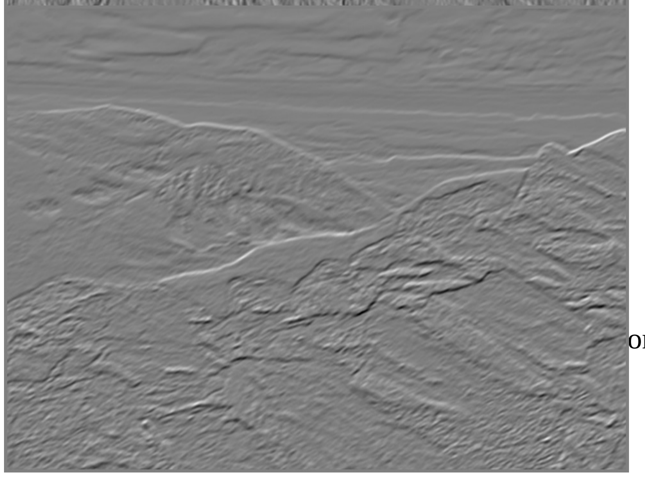
```
  proc main() {
    var rgb : rgbimage;
    var clr : clrimage;
    var retval : int;

    retval = PNG_read(inname, rgb);
    end_onerr(retval, png);

    retval = rgb_convert(rgb, grey, space);
    end_onerr(retval, rgb);

    coforall bank in 0..nrot-1 {
      const theta = (bank * 180) / nrot;
      filter_and_save(img, size, theta, sclx, scly, wavelen, phi_rad);
    }

    free_rgbimage(rgb);
    return 0;
  }
```

And that's it. If you run this program, you'll see that each core of your CPU is occupied. If you cycle through the filter responses you'll notice edges strengthen and weaken as it rotates, with vertical edges strongest for angles near 0 degrees and horizontal for 90 degrees. You may see that the filters are not run in order because there is no guarantee about when the `coforall` iterations will fire. This might not be a good use of `coforall` if the number of banks is more than the number of cores though. A `forall` would be better in this case.

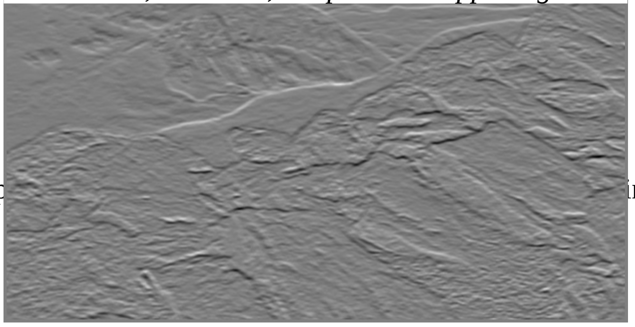




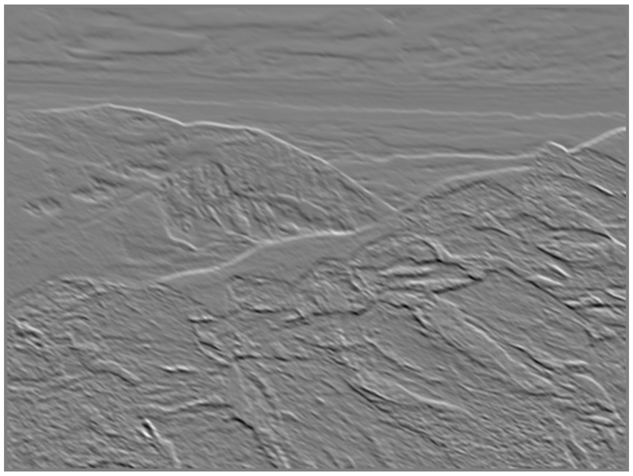
Gabor bank, theta=30, steep lines to upper right

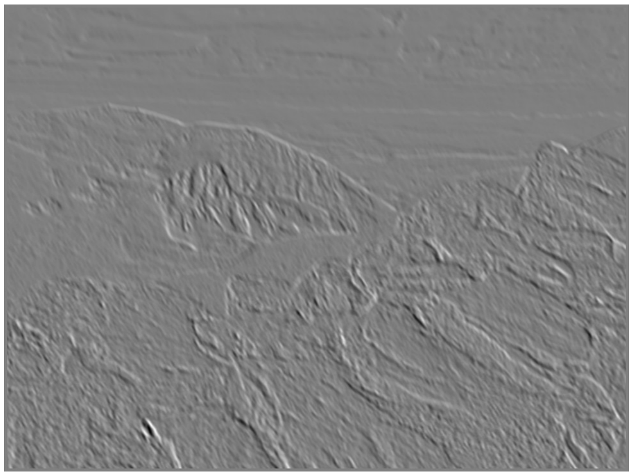


Gabor bank, theta=60, shallow lines to upper right



Gabor bank, theta=90, horizontal lines


Gabor bank, theta=120, shallow lines to upper left


Gabor bank, theta=150, steep lines to upper left

Synchronization

The Chapel language provides synchronization variables and atomic operations. Atomic operations allow multiple tasks to safely interact when reading or changing a value by limiting the operations that can be done and ensuring these cannot be interrupted. We'll talk about atomic variables in the next chapter. Synchronization variables are used to communicate between tasks by forcing one to wait until it can access the value, either for a read or a write. They come in two flavors. The first, `sync`, is indicated by placing the keyword before the type in the variable declaration

```
    var done$ : sync bool;
```

where the trailing $ on the name is traditional. sync variables normally operate in a read-write-read-write cycle, where a task will wait until the previous action occurs. The second are `single` variables which offer a write-once-read-many functionality.

```
    var fired$ : single bool;
```

Chapel pairs a flag with the value of the variable. It tracks whether the variable has a value – is "full" – or does not – is "empty". `sync` normally requires that the flag be empty before a write or full before a read to ensure the proper cycling; if it is not correct, then the task will wait. `single` does not place the requirement on the read, so there is no way to clear the flag for a second write; any read will block until the write occurs. Compound operators like += perform first a read, then a write, so they can be used safely with `sync` variables.

Any boolean, integer, real, or imaginary variable can be used for synchronization. Strings and complex types cannot. You cannot declare a structured type as `sync` or `single`, but you can declare members to be. Similarly you cannot say that an array is a `sync` or `single`, but the array can contain synchronized elements.

```
    var locks$ : [1..n] sync int;
```

We said 'normally' a couple times because there is low-level access to methods on these variables that can override the behavior. Using them in expressions or assignment statements calls down to the default methods. The method names encode the flag state at the beginning and end of the operation.

`done$.readFE()`	sync	read the variable when flag full then clear the flag

`done$.readFF()`	sync, single	read the variable when flag full, leave flag unchanged
`done$.writeEF(val)`	sync, single	assign value when flag empty, then set the flag
`done$.writeFF(val)`	sync	assign value when flag full, leave flag unchanged
`done$.readXX()`	sync, single	peek with non-blocking read, leave flag unchanged
`done$.writeXF(val)`	sync	non-blocking write, then set the flag if empty
`done$.reset()`	sync	assign default value and clear the flag, non-blocking
`done$.isFull()`	sync, single	non-blocking test if flag is set

Normal access to a sync variable is with `readFE()` and assignment is with `writeEF()`. Normal access to a single variable is with `readFF()` and assignment with `writeEF()`.

We'll see an example of a `sync` variable in use in the second version of the FAST corner detector.

Wrap-Up

We've converted two programs now into parallel versions. The first used data parallelism, where we carved up the data to analyze via a `forall` loop, analyzed each piece separately, and combined the results at the end. As long as the intermediate values are kept separate for the loop variable, our calculation is safe, although some re-organization of the code was needed to do this. Our second program used the existing convolver as a black box and instead divided the problem into independent analysis tasks. No reduction step was necessary, and the only code change was to place the work into a procedure and then use a `coforall` to call it. We've learned:

- `forall` is the basic looping structure when working on data in parallel. For tasks, we can use `begin` to fire-and-forget a task, `cobegin` to start a group of tasks in parallel and wait for the result, or `coforall` to start each iteration of the loop as a separate task and wait for the result. `coforall` only makes sense when there are enough computing resources to run all iterations in parallel. In general `forall` is sufficient.

- The statements after the loop, `cobegin`, or `begin` are treated as a function that is executed as a task. Variables from outside the scope of this function are passed in as arguments. You can modify the default intent (`const`) for them by a `with` clause. `inout` and `out` are not allowed.

- Arrays are passed by reference to this function and so their contents may be changed, with the risk of race conditions.

- A `serial` block executes its contents sequentially if its condition holds, irrespective of the presence of one of these four parallel control structures.

- A `sync` block waits until any task that is generated in its scope completes before the next command executes.

- A reduction collapses a collection to a single value. A scan generates as many values as the collection has, each the incremental application of its operator. The `minloc` and `maxloc` operators generate a tuple with the minimum/maximum and the value from a second zippered range of "names" (or, more usefully, indices).

- A `sync` variable is normally handled in a write-read-write-read cycle (although there are methods to read or write multiple times in a row). A `single` variable works in a write-once-read-many manner.

Variables of the primitive types except string and complex may be declared as either. These variables will cause a task to block until the appropriate access is made by another thread.

Next we'll build on this experience and write another parallel program that uses k-means clustering to quantize the colors in an image.

Exercises

1. Create a version of gaborbank that runs the filter at perpendicular angles. Since there are only two calls to the convolver, instead of a variable number, use a `cobegin`. Allow the rotation of one of the angles to be specified on the command line.

2. Modify the color conversions to run in parallel.

3. Modify the Gabor filter convolver to run in parallel.

4. A pyramid is a series of images of decreasing resolution, where the image at level p has half the size (width and height) as at p-1 and whose pixels are an average of the four corresponding below it. That is,

```
I(p,x,y) = (I(p-1,2*x,2*y) + I(p-1,2*x+1,2*y) + I(p-1,2*x,2*y+1) +
            I(p-1,2*x+1,2*y+1)) / 4;
```

A pyramid is often used to find interesting features at coarse resolution (which can be done quickly) and to translate this into an area to explore in more detail at lower levels. Write a program that calculates each level of the pyramid. How would you divide the work to be done in parallel?

Files

A tarball with the material for this chapter is here. A zip file is here.

A PDF of this text is here.

K-MEANS CLUSTERING (kmeans)

Introduction

k-means clustering is a general partitioning strategy. It assigns data points in a n-dimensional space to a number of buckets, effectively dividing the space and quantizing the values. We can use it in image processing to reduce the number of colors in an image to make finding regions of distinct color, or comparing colors, easier. In other words, we attempt to find the strongest clusters of values over the color space and label each pixel with the ID of the cluster to which it belongs.

This is an iterative process that runs until the population of the buckets stabilize. The heart of the algorithm is a per-pixel comparison with each cluster to find the best fit, which then modifies the cluster's boundaries for the next pass. This can be done in parallel, and is the focus of this chapter.

The directory for this chapter is `kmeans`. You'll be using the C functions for reading and writing PNG images in img_png_v3.c and img_png_v3.h, and the color library which is updated to ip_color_v3.chpl. Use `make name` to build the programs, leaving off the '.chpl' suffix. Executables get placed in the `bin` sub-directory and intermediate files in `build`. The image for this chapter is field.png.

The files for this chapter are located here (as zip file). A PDF copy of this text is here.

Aside: Clustering

The k-means clustering algorithm is simple. Begin by giving each bucket a position in the color space. The seeds can be generated randomly, although this tends to produce buckets that are initially empty. We will instead pick pixels at random from the image, making a fixed number of attempts for each and picking the one furthest from all other seeds assuming a minimum separation be met.

For each pass we assign each pixel to the bucket whose center of mass is closest. The center of mass is the average per color plane over all members of the cluster. We then compute the new center of mass from the assignments and look at the population of each bucket. When that has stabilized we are done. If not, any clusters that are unpopulated are re-seeded to midway between the two largest unused blocks and we start another pass.

We do not actually mark the pixels in any way during the pass. We only have to track the aggregates for the cluster. Specifically, the actions at each pixel are to

1. Find the closest cluster. (smallest distance to its center of mass)
2. Increment the count of pixels in the cluster for this pass. (`npix`)
3. Increment a sum of color coordinates for the next center of mass. (`sumc1`, `sumc2`, `sumc3`)

This can be done in parallel. After all pixels are done, then we need to finish processing the clusters. We calculate

1. The new center of mass (`c1cm = sumc1/npix`, `c2cm = sumc2/npix`, `c3cm = sumc3/npix`)
2. The difference in npix between previous pass and current pass. (`dnpix`)

When the largest `dnpix/npix` ratio of all clusters falls below a threshold of, say, 1%, then we are done. We can map each center of mass back to RGB and create a greyscale image with the ID of the cluster at each pixel.

The final algorithm is:

```
1. Seed clusters, ensuring some minimum distance between
2. Repeat a pass
   2a. Assign pixels to clusters, tracking population and color plane sums
       for centers of mass.
   2b. Calculate center of mass.
   2c. Place empty clusters between biggest.
   2d. Calculate cluster population change as fraction of
       population.
   until population change is small enough, or maximum number of passes
   has been reached.
3. Convert cluster colors back to RGB.
4. Create greyscale image holding cluster assignments.
```

Implementation (kmeans_v1)

We will not be presenting all the code in detail here, rather only discussing the Chapel implementation issues. The program can be found in kmeans_v1.chpl. Compile it with

```
make kmeans_v1
```

It takes four command-line arguments

```
--inname     PNG file to read
--outname    file to create with cluster ID assignments
--space      LAB, YUV, HSV, RGB - color space to use
--ncluster   number of clusters
```

Run it with

```
bin/kmeans_v1 --inname=field.png --outname=ids.png
```

You'll find a few changes in the ip_color module, now up to version 3. First there are two new members of the `clrimage` class that store the ranges for the columns and rows because we'll want to use each separately and `array.domain.dim()` is wordy. The domain for the image hasn't changed, but it's now built from these separated ranges. We also store the color space in a third new member so we don't have to pass it around as an argument. The third change is the addition of procedures to convert from any color space back to RGB. (This was Exercise 1 in the Color Conversion chapter.)

We perform the pixel assignment and creation of the ID image in parallel using `forall` loops. Our strategy for the pixel assignment is to create an array `ksplit` of temporary clusters to hold the intermediate sums and reduce them at the end. We iterate over the rows in the outer, parallel loop, and ksplit is specified by row, so this is safe. The procedure returns an array with the new cluster definitions.

```
proc iterate_pass(clr : clrimage, const ref kset : [] cluster) {
  /* intermediate storage, one per row */
  var ksplit : [clr.rows][kset.domain] cluster;
  /* results of pass */
  var knew : [kset.domain] cluster;
```

```
      forall y in clr.rows do
        for x in clr.cols {
          /* closest_cluster returns index of best cluster fit. */
          const closest = closest_cluster(clr.c1(y,x), clr.c2(y,x), clr.c3(y,x),
                                          clr.space, kset);
          ksplit(y)(closest).npix += 1;
          ksplit(y)(closest).c1 += clr.c1(y,x);
          ksplit(y)(closest).c2 += clr.c2(y,x);
          ksplit(y)(closest).c3 += clr.c3(y,x);
        }

      forall k in kset.domain do
        for y in clr.rows {
          knew(k).npix += ksplit(y)(k).npix;
          knew(k).c1 += ksplit(y)(k).c1;
          knew(k).c2 += ksplit(y)(k).c2;
          knew(k).c3 += ksplit(y)(k).c3;
        }
      forall k in knew {
        if (0 == k.npix) {
          k.skip = true;
        } else {
          k.c1 /= k.npix;
          k.c2 /= k.npix;
          k.c3 /= k.npix;
        }
      }

      return knew;
    }
```

The double bracket notation on `ksplit` indicates that we have an array each of whose elements is an array; this contrasts with a single bracket array addressed with a tuple (like the color planes) where the tuple is converted to an index. `cluster` is a record with fields to hold the population `npix`, the center of mass and partial sum per color plane, and a flag if we should ignore the cluster because the population is 0.

```
    record cluster {
      var skip : bool;                /* true to ignore cluster in set */
      var npix : int;                 /* number pixels assigned to cluster */
      var c1, c2, c3 : real;          /* center of mass of each color plane */
      var c1tmp : real;               /* temp needed to sum H */
      var r, g, b : c_uchar;          /* RGB equivalent of c1, c2, c3 */
    }
```

The operation of the procedure should be clear. We write out the loops as nested rather than using a single `forall` statement and tuple as the index variable in order to guarantee which is the outer loop. (The actual code has a special case for the H plane, because we need to account for angles wrapping around 0).

The other use of parallelism is with the ID image. Given an `rgbimage` called `quant` to hold the IDs, the parallel loop looks like

```
forall (y, x) in clr.area {
  var xy = (y * quant.ncol) + x;     /* flat pixel index */
  /* closest_cluster returns index of best cluster fit. */
  quant.r(xy) = closest_cluster(clr.c1(y,x), clr.c2(y,x), clr.c3(y,x),
                                clr.space, kset);
}
```

The `closest_cluster()` procedure checks a color (`c1`, `c2`, c3) against each cluster, using the procedure `cluster_sep()` to measure the distance in a sum-of-squares fashion, and picking the smallest. We simply track the minimum and location as we go along, rather than using a reduction.

Module: Random

The seeding step requires random numbers to pick pixels across the image. Chapel provides a standard module called Random that generates reals in the range 0.0 to 1.0. The module defines a class `RandomStream` whose constructor can take an integer seed; if none is provided, the system time is used. Use the method `getNext()` for a new random value, or `fillRandom(array)` to populate an entire array of reals (`fillRandom()` may also be called as a stand-alone procedure, in which case it creates a new stream). The random number generator is by default parallel-safe.

```
rand = makeRandomStream();
rand.getNext();
var seeds : [1..10000] real;
rand.fillRandom(seeds);
```

The module provides two different generators, NPB, which comes fro the NAS Parallel Benchmark, and PCG. In Chapel 1.17 the module has not yet stabilized and its documentation recommends using `makeRandomStream()` to hide any future changes. The NPBR algorithm can only generate real numbers, while PCG will work with any time. Our code will work with either generator, which means defining two procedures to covert a random real to a position within a range or a domain. `random_ranged()` returns an element from within the range:

```
proc random_ranged(rand : RandomStream, rng : range) : rng.idxType {
  const elt = rng.length * rand.getNext();
  return rng.first + rng.stride * (nearbyint(elt - 0.5) : rng.idxType);
}
```

The element is shifted by -0.5 so that the range is centered within the interval; because `nearbyint()` rounds -0.5 to 0, the limit stays valid. Notice how we can use the type of the range, `rng.idxType`, as the return declaration for the procedure and the cast because this is known at compile time. Ranges do not, unfortunately, have a method to return the nth element, or we would be able to easily support other non-rectangular types. For domains we build a tuple to hold a random value from each range, and then call `random_range()` to populate it.

```
proc random_domain(rand : RandomStream, dom : domain) : dom.rank*dom.idxType {
  var pt : dom.rank * dom.idxType;
  for i in 1..dom.rank do pt(i) = random_ranged(rand, com.dim(i));
  return pt;
```

```
    }
```

This handles any domain geometry. Again we are able to use `dom.idxType` in the declarations, as well as the `dom.rank` parameter. Domains also have the property `dom.numIndices`, but this is not a parameter like the rank and cannot be used at compile-time.

The seeding procedure itself picks a random pixel in the image for the first cluster and then tries a fixed number of pixels for the others, picking the one that is the furthest from all seeded clusters.

Module: Sort

The k-means algorithm is sensitive to outliers affecting the center-of-mass. This can cause clusters to shift into an empty volume of space, further from all data points than any other cluster, and so coming up empty during a pass. Rather than leaving them adrift, we move them halfway between the largest clusters, assuming these are the most likely to need sub-dividing. If there's more than one empty bucket then the first goes between the largest and second largest, the next between the second and third largest, and so on. This is somewhat naive; another strategy would be to track the standard deviation or density of each cluster and split the worst, or to check if the cluster is multi-modal. To do this we need to know the biggest clusters, or to order them by decreasing population. The Sort module provides the standard sorting routines: QuickSort, MergeSort, HeapSort, InsertionSort, BubbleSort, and SelectionSort. The `sort()` function is the general-purpose entry into the library and wraps a call to QuickSort. All work on 1D arrays of arbitrary type and require a `compare()` function that returns a negative number if item1 comes before item2, a positive number if after, and 0 if equal. (This matches C's `qsort()` convention.) We can overload the default function for our data type. We do this locally inside fillin_empties() so that we can use different comparisons in other places.

```
    proc fillin_empties(kset : [] cluster, space : clrspace) {
      proc DefaultComparator.compare(c1 : cluster, c2 : cluster) {
        return (c1.npix - c2.npix);
      }
```

This puts the clusters in ascending size. To reverse the order the module has defined a reverseComparor that inverts the result of the standard Comparator. We only need to use it; there's no additional code.

```
      sort(kset, comparator=reverseComparator);

      /* assign empties */
    }
```

The default comparator itself uses `<`, so you could overload that instead.

```
    record DefaultComparator {
      proc compare(a, b) {
        if a < b { return -1; }
        else if b < a { return 1; }
        else return 0;
      }
    }
```

Atomic Variables (kmeans_v2)

Atomic variables are restricted either in software or by hardware so that certain operations are guaranteed to be consistent between tasks: one will never see the variable in an intermediate state. This might be done by placing a lock around the variable so that only one task can interact with it at a time, or it may use CPU commands that protect the memory location. In Chapel any integer, unsigned integer, boolean, and real may be declared atomic. Simply place the `atomic` keyword before the type

```
    var pixsum : atomic int;
```

Although you cannot treat structural types as atomic, you can declare their members to be so.

You will not be able to use normal assignments or operators for this variable, neither to change its value nor access it. Instead, the variable will now support several methods:

`pixsum.read()`
: returns the variable's value

`pixsum.write(val)`
: places a new value in the variable

`pixsum.compareExchange(orig, new)`
: stores `new` as the value only if orig is the current value, otherwise does nothing
returns `true` if the value was changed, `false` if not

`pixsum.add(val)`
: equivalent to `pixsum += val`; not supported for bools; returns nothing

`pixsum.sub(val)`
: equivalent to `pixsum -= val`; not supported for bools; returns nothing

`pixsum.or(val), pixsum.and(val), pixsum.xor(val)`
: equivalent to `|=`, `&=`, `^=`; only supported for integers; returns nothing

`pixsum.fetchAdd(val), pixsum.fetchSub(val)`
: as `pixsum.[add|sub](val)` but returns the original value; not supported for bools

`pixsum.fetchOr(val), pixsum.fetchAnd(val), pixsum.fetchXor(val)`
: as `pixsum.[or|and|xor](val)` but returns the original value; only integers

`pixsum.testAndSet`
: sets the value to `true` and returns the old value; only supported for bools

`pixsum.clear`
: sets the value to `false`; only supported for bools

`pixsum.waitFor(val)`
: blocks the task until the variable has the given value

We can use atomic variables to replace the intermediate array during the k-means iteration pass. You'll find the changes in kmeans_v2.chpl. If you look at the `iterate_pass()` procedure in kmeans_v1, you'll notice that we only need to add and read `ksplit`. We add five atomic members to the cluster structure to handle the intermediate sums

```
    record cluster {
      var skip : bool;                    /* true to ignore cluster */
      var npix : int;                     /* number pixels in cluster */
      var c1, c2, c3 : real;              /* center of mass */
      var r, g, b : c_uchar;              /* back-converted RGB value */
      var sumpix : atomic int;            /* incremental sum for npix */
      var sum1, sum2, sum3 : atomic real;    /* incremental sum for c[1-3] */
      var sumh : atomic real;                /* second sum needed for H wrap */
    }
```

We could have declared `npix`, `c1`, `c2`, and `c3` to be atomic, but this would mean we would have to use the read/write methods everywhere else in the program. (OK, we did do a version like this, but it was harder to read; since our changes are limited to the `iterate_pass()` procedure, this seems to be the cleaner and easier change.)

The iteration pass is now simpler, using only the new clusters.

```
  forall (y, x) in clr.area {
    const closest = closest_cluster(..);
    knew(closest).sumpix.add(1);
    knew(closest).sum1.add(clr.c1(y,x));
    knew(closest).sum2.add(clr.c2(y,x));
    knew(closest).sum3.add(clr.c3(y,x));
  }
  forall k in knew {
    k.npix = k.sumpix.read();
    k.c1 = k.sum1.read() / k.npix;
    k.c2 = k.sum2.read() / k.npix;
    k.c3 = k.sum3.read() / k.npix;
  }
```

Again we've omitted the special case for the H plane; see the source for details.

Chapel's implementation of atomic variables is still in active development and what we've presented here describes the current implementation. Eventually you will be able to specify looser restrictions on accessing the variable, and you will be able to pass these conditions as arguments to all the methods above. There are also design considerations if you work with atomic variables when tasks are split over different pieces of hardware or inter operate over a network. You'll find details by digging into the documentation. CHPL_HOME/doc/rst/technotes/atomics.rst is the starting point.

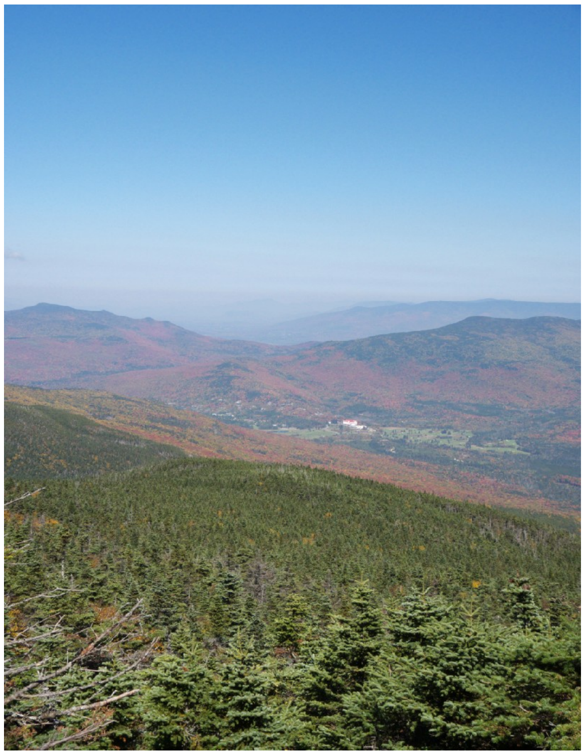
To use this program

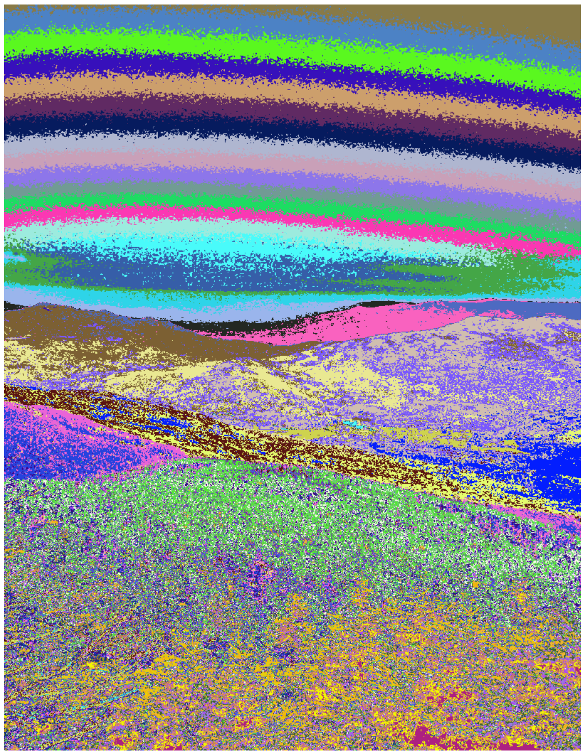
```
  make kmeans_v2

  bin/kmeans_v2 --inname=field.png --outname=ids2.png
```

The result will differ from kmeans_v1 because the random number generator has no seed. If you provide the same value in both programs in the '`const rand = new RandomSeed();`' line in `seed_clusters()` the results will be the same.

Performance

This program was the inspiration for this tutorial. We've been playing around with color quantization recently, and have threaded and CUDA versions running in C. You may notice that the accuracy of the color quantization isn't fantastic and that the algorithm can take a long time to settle. The inaccuracy is related to proper seeding and handling of empty clusters; splitting large clusters also seems necessary. The run time is tied to the relationship between colors in the scene and the number of clusters used; too many clusters leads to slow convergence, too few loses detail. This is an active area of research for us, so we can't offer solutions.

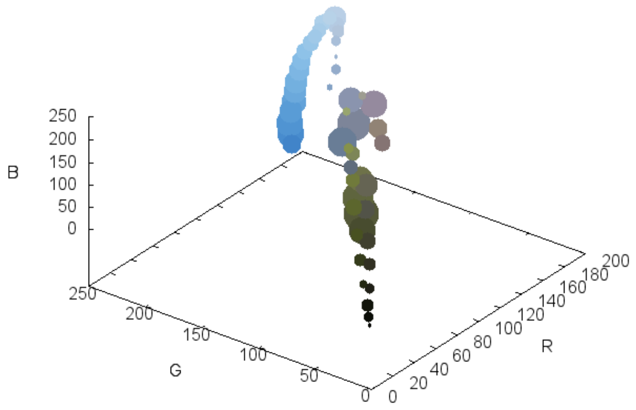

Original image field.png


Cluster IDs assigned random color

You can use gnuplot to show where the clusters are. The graph will plot a dot in the 3D R, G, B space for the cluster at its center of mass and with a size proportional to the number of pixels contained in it. First save the output from kmeans and delete all lines up to and including the table header. Then start gnuplot and type

```
    set xlabel "R" ; set ylabel "G" ; set zlabel "B"
    set nokey
    set view 45, 307, 1, 1
    basenpix = 5000
    splot 'datafile' u 6:7:8:($2/basenpix):($6*65336+$7*256+$8) with points pt 7
      ps variable lc rgb variable
```

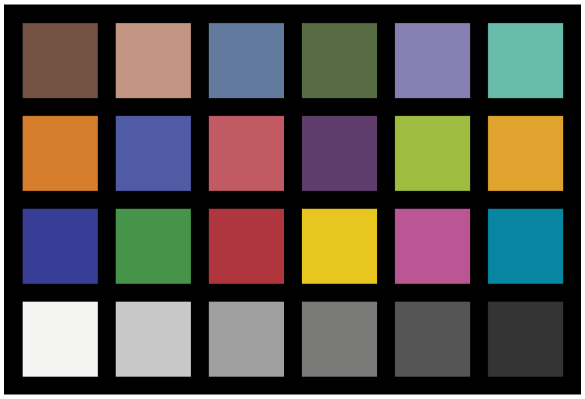
The value `basenpix` should be about the number of pixels in the smallest cluster and determines how big to draw each dot.

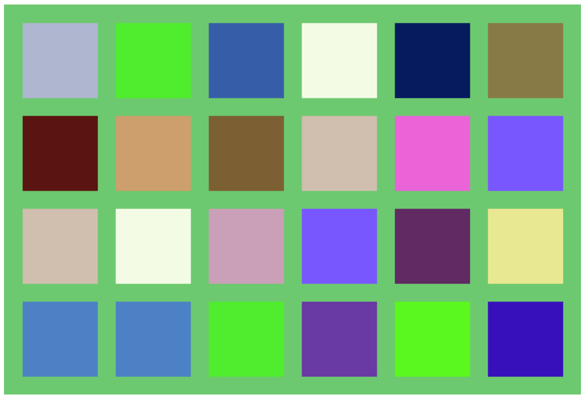


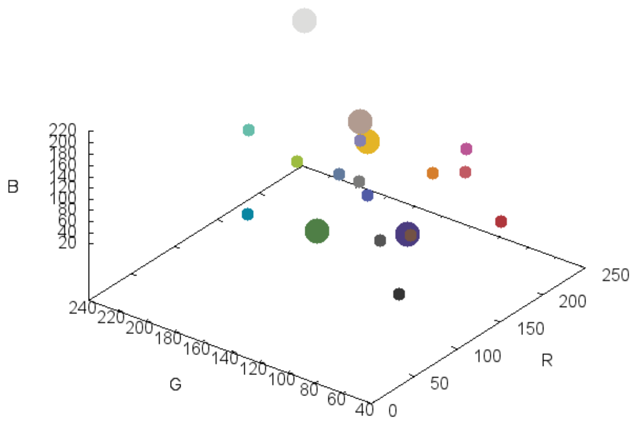
Clusters in the image

To demonstrate this inaccuracy, the color_checker.chpl program creates a PNG image colors.png with the colors of the Macbeth chart. Running kmeans on this image with `--ncluster=25`, you'll find not every block has a unique color. Although the random seeds will give you different results, the run below only found 19 colors of the 25 that exist (including the black bars between squares). In the gnuplot diagram there are 5 large dots for the blocks that have combined: (2,1) and (3,4); (4,1) and (2,3); (4,2) and (1,3); (6,2) and (4,3); (1,4) and (2,4), where the numbers are (column,row). (Block (5,4) looks similar to (3,4) but is really different.) Compile and run the program with

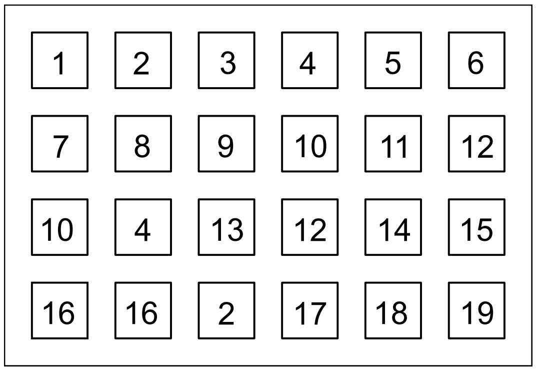
```
make color_checker

bin/color_checker
bin/kmeans_v[12] --inname=colors.png --outname=ids_clr.png
```


Color chart


colorized IDs showing many blocks merged



Clusters in color checker, excluding black



1	2	3	4	5	6
7	8	9	10	11	12
10	4	13	12	14	15
16	16	2	17	18	19

IDs assigned to color blocks

What we can compare is the performance of the two approaches we've taken. Running on the field.png image with 50 clusters and a fixed seed to RandomStream, we find run times (in seconds) of

	base	--fast	improve base	
v1	167	14	12 X	column sums
v2	145	15	10	atomic variables

Both programs ran the clustering in parallel on four cores. The base column gives the time the analysis took when the program was compiled normally, the `--fast` column when built with `CHPLFLG=--fast`. The improve column is the ratio of the base to the fast times.

Atomic variables do not hurt the performance of this program. Only when we take away the base sanity checking by the runtime do we see a small hit. As with the Gabor filter programs, we again see an order of magnitude improvement when we compile with `--fast`.

Wrap-Up

In this chapter we've used a divide-and-reduce approach to running the color clustering in parallel. We've also seen how we can use Chapel's atomic variables to eliminate the intermediate storage and reduce step; the performance of this approach is competitive with the first. We've learned:

- Integers, reals, and bools may be declared atomic. You can declare members of structures atomic, but not the structure itself.
- Atomic variables perform uninterruptible access and modification to their values. They require explicit `read()` and `write()` methods to do this. They support a limited number of operations, including `+=`, `-=`, `&=`, `|=`, and `^=`, as well as compare-and-exchange and test-and-set.
- The Random module provides a stream of values between 0.0 and 1.0. You can fix the seed for the sequence when you instantiate the stream.
- The Sort module implements the common sorting algorithms. There is no comparison function; you

must overload the relational operators `>` and `>=` on your data type, or `<` and `<=` if sorting in decreasing order.

In the next chapter we'll implement another parallel image processing program, looking for corners in images with the FAST detector. This will require we write our own iterator.

Exercises

1. Generate a back-mapped image. That is, create a color RGB image where instead of cluster IDs you store their corresponding RGB values. How do the results compare to the original image?

2. Write a Gaussian filter to smooth the initial image (in color space) before doing the quantization. How does this affect the quantization?

3. This implementation compares each pixel against each cluster to find the closest. The kdtree library in the RANSAC chapter is more efficient, especially as the number of clusters increases. Modify the kmeans program to use it, building a new tree before each pass and using `find_nearest_point()` to get the closest cluster.

Files

The files needed for this chapter can be found here. A zip file is here.

A PDF version of this text is here.

FAST CORNER DETECTOR (fast)

Introduction

We have implicitly used Chapel's concept of an iterator in our loops. Iterators are constructs that step through a sequence one item at a time. A loop processes these steps until there are no more. Iterators are provided for ranges, but we can also create our own. To demonstrate this, we'll program a circular iterator for a FAST corner detector.

The directory for this chapter is `fast`. You'll be using the C functions for reading and writing PNG images in img_png_v3.[ch], and the color library ip_color_v3.chpl. Use `make name` to build the programs, leaving off the '.chpl' suffix. Executables get put in the `bin` sub-directory, intermediate files in `build`. The sample image for this chapter is owl.png.

The files needed for this chapter are here (as zip file). A PDF copy of this text is here.

Aside: Corner Detectors

A corner is a 2D feature in an image, a point where there are strong gradients in orthogonal directions. If a block is aligned to the pixel grid then the vertical edge has a strong gradient over x but not y, and the horizontal edge has a strong gradient in y, not x. As you approach the point where the edges meet the opposite gradient begins to increase, until at the corner both are present. Alternatively we can look for patterns of "on" and "off" pixels in the neighborhood of a corner, where the difference between on and off is some threshold. Corner detectors are run on greyscale images.

The Harris corner detector tests the gradients. Using a differential edge detector to measure them, we smooth the data with a low-pass filter to reduce noise and then calculate the determinant and trace of the matrix of the second-order derivatives

$$\begin{vmatrix} \mathtt{dxxI} & \mathtt{dxyI} \\ \mathtt{dyxI} & \mathtt{dyyI} \end{vmatrix}$$

```
det = (dxxI * dyyI) - (dxyI * dyxI)
trace = dxxI + dyyI
dxxI = dxI ** 2
dyyI = dyI ** 2
dyyI = dyI ** 2
```

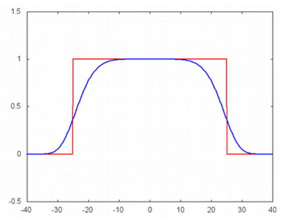
where we approximate the second-order derivatives by applying the first derivatives twice. The metric

```
    det - kappa * trace * trace
```

gives the strength of the corner detector. kappa is usually about 0.04 and smaller values pass more corners. You can threshold this value to mark corners in the image.

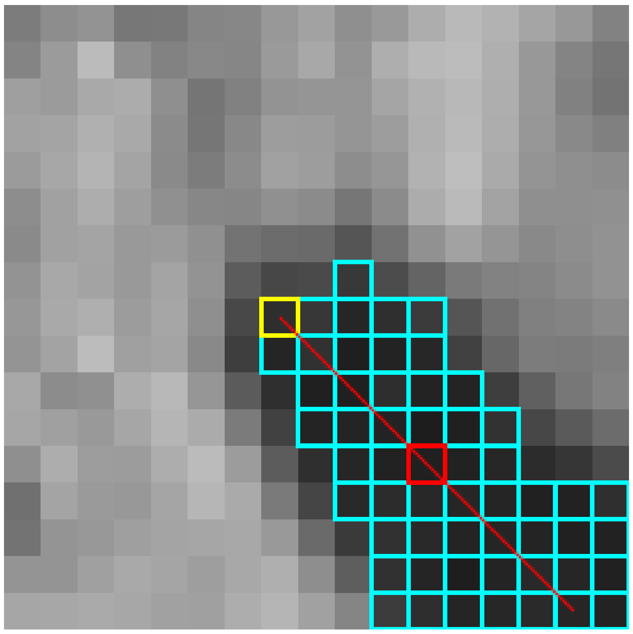
The SUSAN corner detector sums the difference between each pixel within a circle to the center (xc, yc). It uses a smoothing function (blue) instead of an abrupt threshold (red)

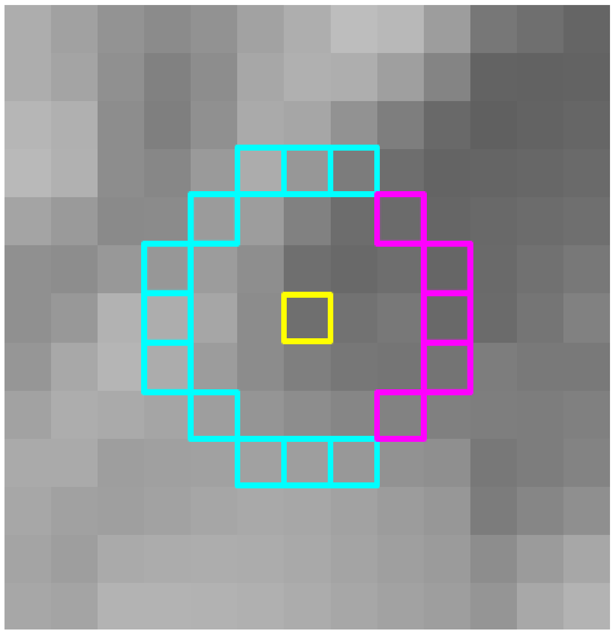
$$c = e^{-(\Delta I(x,y)/t)^6}$$



SUSAN threshold

where t is a threshold parameter. Note that this is inverted in the sense that similar intensities contribute the most. If this sum is below half the maximum possible over the spot, then the detector verifies it has found a corner by looking at the pixels that contribute to it. If their centroid is far from the center and all pixels along the line between the center and the centroid out to the edge of the circle belong to the set, then (xc,yc) is declared a corner. Because the function is independent of the direction of the intensity change the detector works equally well for a bright wedge on a dark background as for a dark wedge in light.


SUSAN members in cyan, centroid in red


FAST light corner in cyan, len 11, start (+1, +3)

The FAST corner detector scans the perimeter of a circle about its center, marking pixels if they are above or below the center by a threshold, or considering them the same. The detector requires that a significant fraction of the periphery be a continuous stretch above or below. For example, in a circle of radius 3 there are 16 pixels in the circumference and we may require 9 or 11 in a row be strongly different in the same direction. You may further add that the remaining pixels be near the center's intensity, although we will not do so.

Iterators

An iterator is defined like a procedure but with the `iter` keyword.. It may take arguments. It cannot overload another operator. Inside the body you can use `return` without any expression, and may also use the `yield` statement to give the next value. The iterator will be called repeatedly by the loop and will execute until it hits either a `return` and or `yield`. At the return it ends. At the yield it pauses, returning the yield's expression. The next time the iterator is called it resumes from the point it paused with the same context as before; that is, the iterator operates as a closure.

An example duplicating a simple range with a stride is

```
  iter whilerange(lo : int, hi : int, stride = 1) {
    var cnt = lo;

    /* Check validity of range. */
    if (((hi < lo) && (0 < stride)) || ((lo < hi) && (stride < 0)) ||
        (stride == 0)) then return;
    if (0 < stride) {
      while (cnt <= hi) {
        yield cnt;
        cnt += stride;
      }
    } else {
      while (hi <= cnt) {
        yield cnt;
        cnt += stride;
      }
    }
  }
```

```
  for i in whilerange(1, 5) do writeln(i);
  > 1
  > 2
  > 3
  > 4
  > 5

  for i in whilerange(-1, -5, -1) do writeln(i);
  > -1
  > -2
  > -3
  > -4
  > -5

  for i in whilerange(1, 10, 2) do writeln(i);
  > 1
  > 3
  > 5
  > 7
  > 9

  for i in whilerange(1, 5, -1) do writeln(i);
  (no output, iterator not executed)

  for i in whilerange(-1, -5, 1) do writeln(i);
  (no output, iterator not executed)

  for i in whilerange(1, 5, 0) do writeln(i);
  (no output, iterator not executed)
```

An iterator that is evaluated, either as an expression or during assignment to a variable or array, generates its entire sequence stored in a one-dimensional array.

```
    writeln(whilerange(1,10,2));
    > 1 3 5 7 9
```

This example is found in ex_iterator.chpl. Compile and run it with

```
    make ex_iterator

    bin/ex_iterator
```

Custom Iterators (fast_v1, test_fast_v1)

Our approach to implementing a new iterator evolved while working on this exercise. The original plan was to have a 'circumference' that would step around a circle with a given radius at a central point. For small radii, say r < 7, we could just list the points in a series of yield statements; the calculation wouldn't get any more efficient. For large radii we could calculate the points, which involves squaring and a square root, taking advantage of the symmetry of the problem by only doing this in the first quadrant and reflecting into the other three.

Ah, but what if want to do this at every point in an image? This calculation could be cached, of course, which means that in addition to the iterator we need to store the intermediate results. A class would seem the natural way to do this. For consistency we treat all cases the same, even for small radii, although this will be a little less efficient because the offsets to get the points will be stored in an array rather than hard-coded.

Class methods this() and these() (ex_class.chpl)

Both classes and records support two inherent methods we haven't talked much about yet. The `this()` procedure is called if you pass arguments to an instance.

```
    var circle = new circumference(radius);
    circle(center_x, center_y);
```

The method call on the instance circle is to `this()`, which we would define inside the class as

```
    class circumference {
      proc this(center_x : int, center_y : int) : int { ... }
    }
```

The `these()` procedure is called by a loop to iterate over the class. The iterator takes no arguments, and follows the normal construction rules.

```
    class circumference {
      iter these() { yield pt1; yield pt2; ... }
    }
```

`these()` is what we want to use, and we can abuse `this()` to get the behavior we want. Our class will offer two ways of calling it. First, we can completely specify the iteration – its radius, center point, and whether we want the circle to close by repeating the first point as the last (otherwise it stops just before) – when we construct the instance. Giving the instance a `these()` will allow us to use it on the loop line

```
    for (x, y) in new circumference(radius, center_x, center_y, closed) { ... }
```

As written, this is a one-shot approach since there is no way to change these parameters outside the constructor. The second way will take advantage of caching by providing a `set_radius()` method to pre-calculate the quadrant and then use the `this()` method to return an iterator for a given point. The radius is fixed while the iterator moves about the image so that the cost of caching is paid once. This means we will have to store the radius with the class.

```
var circle = new circumference(radius);
for (x, y) in circle(center_x, center_y, closed) { ... }
circle.set_radius(radius2);
for (x, y) in circle(center_x2, center_y2) { ... }
```

For the stand-alone mode we will need to implement a constructor and `these()`.

```
class circumference {
  var r : int;
  var x_center, y_center : int;
  var close_circle : bool;
  var Lquad : domain(1);          /* range for 1st quadrant points */
  var quad : [Lquad] 2 * int;     /* the points, as an (x,y) tuple */
  var quadcnt : int;              /* number of points in quad (incl.) */

  /* closed is an optional argument. */
  proc circumference(radius : int, xc : int, yc : int, closed = true) {
    set_radius(radius);
    x_center = xc;
    y_center = yc;
    close_circle = closed;
  }

  proc set_radius(radius : int) {
    if (r != radius) {
      r = radius;
      prep_quadrant();
    }
  }

  iter these() {
    const xc = x_center;
    const yc = y_center;
    const closed = close_circle;

    for i in 0..quadcnt {
      var (x, y) = quad(i);
      yield (xc + x, yc + y);
    }
    /* Second quadrant.  quadcnt-1 to not duplicate first point. */
    for i in 0..quadcnt-1 by -1 {
      var (x, y) = quad(i);
      yield (xc - x, yc + y);
    }
    /* Third quadrant.  Start at one to not duplicate point. */
```

```
      for i in 1..quadcnt {
        var (x, y) = quad(i);
        yield (xc - x, yc - y);
      }
      /* Fourth quadrant. */
      for i in 1..quadcnt-1 {
        var (x, y) = quad(i);
        yield (xc + x, yc - y);
      }
      if (closed) {
        var (x, y) = quad(0);
        yield (xc + x, yc + y);
      }
    }

    proc prep_quadrant() {
      var mid = 0;            /* start of second half of quadrant */
      var x, y : int;

      Lquad = 0..2*r;         /* slightly oversize the array */

      /* The actual routine sets up the quadrant automatically for
         radii between 3 and 7. */

      quadcnt = 0;
      x = r;
      y = 0;
      do {
        quad(quadcnt) = (x, y);
        quadcnt += 1;
        y += 1;
        x = nearbyint(sqrt(r**2 - y**2)) : int;
      } while (y < x);
      do {
        y = nearbyint(sqrt(r**2 - x**2)) : int;
        quad(quadcnt) = (x, y);
        quadcnt += 1;
        x -= 1;
      } while (0 <= x);
      /* quadcnt is inclusive. */
      quadcnt -= 1;
    }
  }
```

The `quad` array is the cache. `prep_quadrant()` handles each octant in the quadrant separately to ensure the ring is continuous, looping over y from 0 to 45° and x from 45° to 90°. The domain `Lquad` of the array is slightly bigger than necessary because we don't have a way to calculate the number of points in advance. The iterator steps through the cache for each quadrant, using the ring's symmetry to add or subtract the peripheral point from the center. The only tricky part is skipping the first or last point from the range to avoid overlapping the last point of the previous quadrant.

For the cached version we need a different constructor taking only the radius, `this()`, and an iterator over the cached quad. These get added to the class definition above.

```
proc circumference(radius : int) {
  set_radius(radius);
}

/* closed is an optional argument */
proc this(xc : int, yc : int, closed = false) {
  return circum_iter(xc, yc, closed);
}

iter circum_iter(xc : int, yc : int, closed : bool) {

  /* Same as the 'these' function above, except it uses the arguments
     and not the members. */
}
```

The program ex_class.chpl shows this framework, which we used to figure out how to use `this()` and `these()`. Compile and run it with

```
make ex_class

bin/ex_class
```

It has some differences to the final approach. In the example the `this()` method stores the center point as a class member and then returns the `these()` iterator. This would not work in a parallel loop because the center for each task would overwrite the others and the loop would be unpredictable. We had to define `circum_iter()` so that all variables are local to it. Each task sees a separate version and can operate independently.

There is one glaring bit of ugliness with this approach. If we cannot call `these()` from `this()` and be safe, we also cannot call `circum_iter()` from `these()`. Being an iterator, `these()` must contain a `yield` statement. It cannot call out to another iterator because it must return `void` (no value), unlike `this()` which can return the the iterator. Compilation would fail if you were to try this. So we are stuck with two nearly identical copies of the iterator; in fact, `circum_iter()` is just `these()` without the first three assignments.

Also ugly, but less glaring, is using `this()` as an iteration method. If you were willing to have a method call in the loop, you could delete it and directly use

```
for (x, y) in circle.circum_iter(xc, yc) { ... }
```

Also ugly is that the stand-alone version will leak memory because there's no way to call delete on the instance when the loop is done and the instance goes out of scope. We wrote this before learning that classes require manual management.

FAST Corner Detector (fast_v1.chpl, test_fast_v1.chpl)

The corner detector program takes a color PNG image, converts it to a LAB or YUV greyscale (ie. uses the L or Y planes), and then checks each pixel to see if it meets the FAST criteria. If so, it marks the pixel on a copy of the greyscale image. `main()` calls `mark_corners()` to make the copy and perform the check; each can be done in parallel `forall` loops. Using `forall` is all that needs to be done to have the program running in parallel.

The FAST test is done in the `is_corner()` procedure. It classifies each pixel in the circumference as being more than a threshold amount above (`MORE`) or below (`LESS`) the center, or within the threshold (`SAME`), using the `pixel_thrdir()` procedure. It then counts the number of consecutive `MORE` or `LESS` pixels, taking care to wrap around at the first point. If this count is within the acceptable range, then the routine returns `true`, the pixel is at a corner.

Compile the program with

```
make fast_v1
```

You can adjust the analysis parameters on the command line:

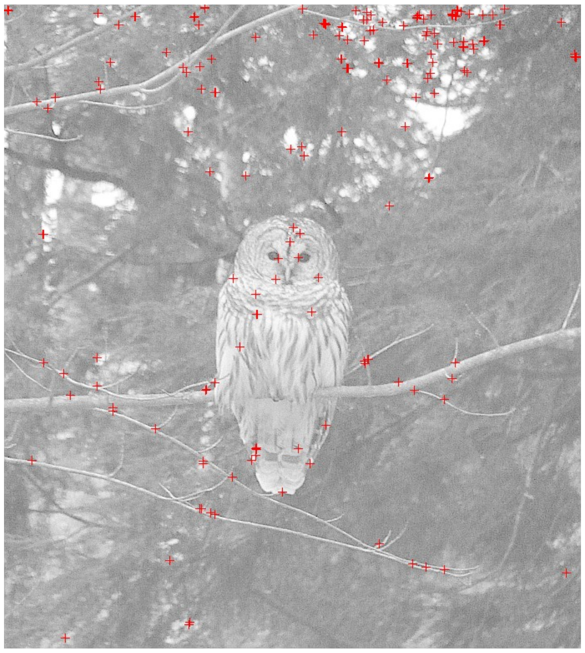
```
--inname=<file>        input file
--outname=<file>       greyscale copy with corners marked in red
--space=[LAB|YUV]       which greyscale plane to use, L or Y
--radius=#              radius of the detector
--minlen=#              minimum count of consecutive MORE or LESS pixels
--maxlen=#              maximum count of consecutive MORE or LESS pixels
--thr=#                 greyscale difference to label as MORE or LESS
```

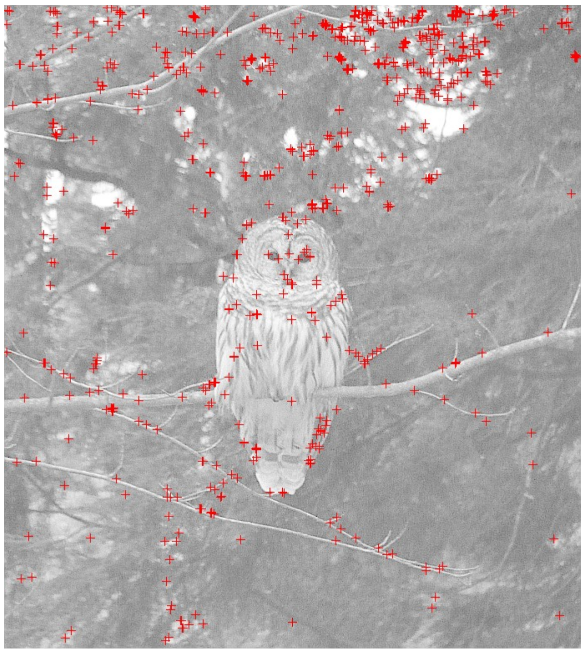
The FAST detector normally uses a ring of radius 3, which gives 16 pixels in the circumference. There's 20 for r=4, 28 for r=5, 32 for r=6, and 40 for r=7 using the patterns found in fast_v1.chpl. `minlen` and `maxlen` will therefore change depending on the radius, and are normally close to 3/4 of the circumference, representing a right-angle. We use minlen=10 and maxlen=13 as our defaults. Remember too that the data range for L is 0-100 and Y 16-235. The threshold should change with the color space. You'll find that varying this parameter will change the sensitivity of the detector: too low and there will be many false positives, too high and you will mark only the strongest corners.

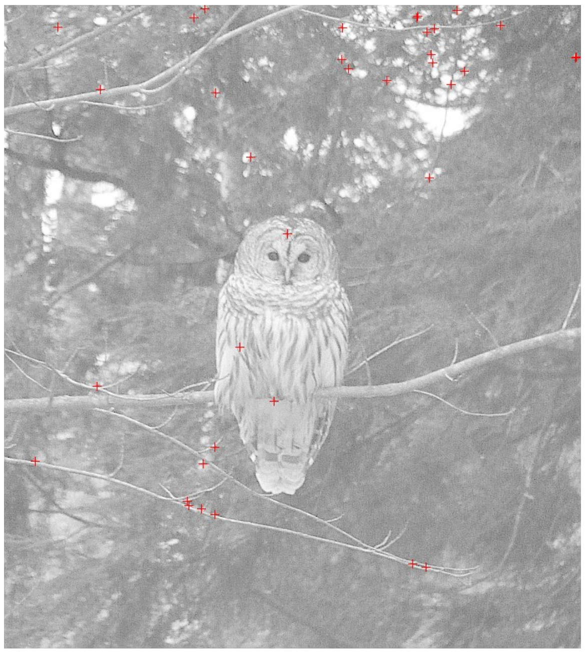
To run the program on the test image do

```
bin/fast_v1 --inname=owl.png --outname=owl_corners.png
```

Original image owl.png


FAST corners, thr=20

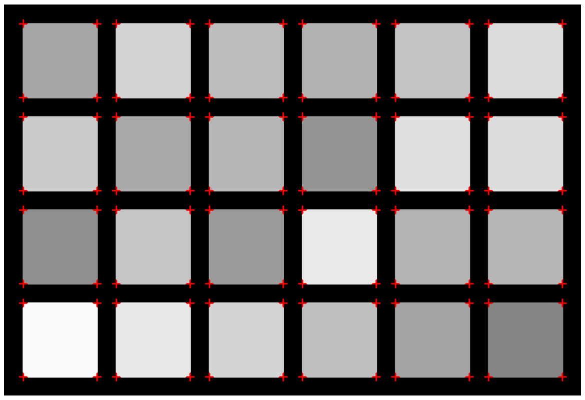

FAST corners, thr=15


FAST corners, thr=25

You can also run color_checker in kmeans to generate a rectangle grid of differently colored squares and use this as the input image.

```
../kmeans/bin/color_checker

bin/fast_v1 --inname=colors.png --outname=color_corners.png
```

The artificial scene is accurately marked. Natural scenes tend to be quite a bit noisier. Try different thresholds. For example, `--thr=20` on owl.png gives 181 corners, `--thr=15` 724, and `--thr=25` 36. What threshold in the YUV space gives a result similar to the default LAB?



FAST corners on color checker

The program test_fast_v1.chpl is a self-checking test bench for the circumference iterator and the `is_corner()` classification. Since it uses fast_v1.chpl as a module, we have the problem that there are two modules with `main()` routines. To compile you'll need the extra compiler argument to pick one

```
    chpl --main-module test_fast_v1 -o bin/test_fast_v1 test_fast_v1.chpl
         fast_v1.chpl
or  make test_fast_v1

    bin/test_fast_v1 --minlen=9
```

Corner Suppression (fast_v2, test_fast_v2)

If you run the detector on a natural scene, you'll notice the number of corners is very sensitive to the threshold, as we saw with the owl picture. Trees and plants in general create problems. The literature contains ideas of how to reduce the count. For example, a comment in Klette (Concise Computer Vision) mentions using a figure of merit, the largest difference in intensity between the FAST sequence and ring center, to rank the corners, dropping those below some threshold. Szeliski (Computer Vision: Algorithms and Applications) talks about adding a spatial separation criteria to the decision. We can combine these ideas to suppress corners that aren't locally maximum in the figure of merit sense. We will add a suppression distance parameter that will chain together all corners whose x or y coordinates are within this distance of another member of the group, keeping the one with the highest figure of merit. The others will be dropped.

We'll need an expandable, sortable list to hold the details about the corners, which we'll implement as a generic class. This will give us the chance to cover Chapel's support for generics.

Generic Classes and Records

Chapel considers a class, record, or function to be generic if you can customize the type of its members or arguments, or the size with parameters.

You can make a generic class or record in three ways. First, you can create a type member

```
    class chunkarray {
```

```
    type eltType;
  }
```

This identifier can be used in any position that a type normally would, including argument lists of methods, declarations, and return types.

```
  class chunkarray {
    type eltType;
    var data : [Lalloc] eltType;           /* our payload */

    /* This allows array-like access to the data, ie. instance(index).
       It returns an element at the index ind with the type of our
       payload. */
    proc this(ind : int) : eltType {
      return data(ind);
    }

    /* This will store an element of the generic type in the array. */
    proc append(const in val : eltType) {
      /* ... */
    }
  }
```

If you need a procedure to return a type, by the way, then put the intent keyword `type` after the declaration, without a colon

```
  proc get_type() type {
    return eltType;
  }
```

Arguments can also take a `type` intent, in which case the type, not the value, in the caller is passed to the procedure.

The second way of creating a generic class or record is to use a parameter member

```
  class chunkarray {
    type eltType;
    param initalloc : int;
    var data : [1..initalloc] eltType;
  }
```

Third, you can omit a type when you declare a member. The type will be assigned when the class or record is instantiated. A node in a linked list might be

```
  class listnode {
    var val;
    var next : listnode(val.type) = nil;
  }
```

The generic component to the class is set up with a type constructor. The compiler will generate one for each of these three cases with the usual rules: one argument in the order listed in the class with its name for keyword access. It does not include arguments for non-generic members. The constructor when called returns an instance type with the types of the members set correctly. Use this instance type in the declaration of a variable. If you initialize the `type` or `param` members, then these values become the defaults in the constructor; otherwise the argument must be provided.

As an example of a generic class, we'll develop a chunked array that we'll use to store the corners we find. We don't know in advance how many corners there are, so the data structure needs to grow. We'll want to access elements randomly, so a linked list will be too inefficient. We'll use an array list. To avoid having to re-size the array with each insertion once it fills, we over-size the growth using chunks. Re-allocations are needed only when the chunk fills, which amortizes the cost of the growth. Often array lists double in size when they grow. We'll use a fixed increment to keep the overhead small, at a cost of having to re-size more often.

For our example, we'll keep one array of data with two ranges, the first representing the allocated elements and the second the subset of actual data that's been stored. We'll have two parameters that set the initial array size and the amount to grow when full. Finally, we'll need a synchronized variable to guard access when adding elements; we want to be able to use the array safely within a parallel block. The declarations look like

```
    class chunkarray {
      type eltType;
      param INIT_ALLOC = 2;      /* values for demo */
      param GROW_ALLOC = 2;
      var Lalloc : domain(rank=1) = 1..INIT_ALLOC;
      var Ldata : domain(rank=1) = 1..0;
      var data : [Lalloc] eltType;
      var lock$ : sync int = 1;
    }
```

The class is generic because we have a `type` member; we'll need to provide that when instantiating. Notice that the `Ldata` range is initially empty because the end index is less than the beginning. We've also initialized `lock$` so its flag starts in the full state.

Adding an element means checking that we have room; if not, then we increase the `Lalloc` range which automatically re-sizes the data array. There is no method to expand a range on just one side, so we need to slice off the expansion on the lower end. Alternatively, we could avoid `expand()` by tracking the size ourselves, but we want the range to manage itself. The `lock$` variable guarantees that only one call is active at a time; we use the explicit `readFE()` and `writeEF()` procedures to make this clear. We use a `const in` intent for the value so we make a copy.

```
      proc append(const in val : eltType) {
        lock$.readFE();

        /* Slice the expansion to trim negative indices. */
        if (Ldata.high == Lalloc.high) {
          Lalloc = Lalloc.expand(GROW_ALLOC)[1..];
          /* For demo to show we're re-allocating. */
          writeln("grew array to ", Lalloc);
        }
        /* Now there's room for the new element. */
        Ldata = Ldata.expand(1)[1..];
        data(Ldata.high) = val;

        lock$.writeEF(1);
      }
```

We'll also provide `this()` and `these()` methods so we can use this class like an array. `this()` will return a copy of the valid data. You must provide parentheses, there is no version of the method without. It is not possible to return a reference to a slice in an array, so to access the actual array you would have to use the members directly. `this(int)` returns the element at an index. `these()` will step over just the data, not the entire allocation. Finally, we'll add a `size()` method to return the number of data points, which is also the largest index (inclusive).

```
    proc this() {
      return data[Ldata];
    }
    proc this(ind : int) : eltType {
      return data(ind);
    }
    proc these() {
      for d in data[Ldata] do yield d;
    }
    proc size() {
      return Ldata.high;
    }
```

To demonstrate this class, let's define a simple record to hold some details about a corner, the center point `xc`, `yc` and the length `len` of the consecutively differing pixels

```
  record corner {
    var xc, yc : int;
    var len : int;
  }
```

We create a new `chunkarray` to hold corners, and an instance of the record to show that the values will be copied into the array, not a pointer to the record. The `chunkarray` instantiation requires the generic type be passed as an argument.

```
  var corners = new chunkarray(corner);
  var details : corner;
```

We add a few data points

```
  details.xc = 100; details.yc =  50; details.len = 10;
  corners.append(details);
  details.xc = 150; details.yc =  75; details.len = 13;
  corners.append(details);
  details.xc = 125; details.yc = 100; details.len = 12;
  corners.append(details);
  > grew array to {1..4}
```

before we can show how to access the data

```
 writeln("corners data in " , corners().domain, "   size ",
         corners.size());
 > corners data in {1..3}   size 3

 writeln("corner #2 ", corners(2));
 > corner #2 (xc = 150, yc = 75, len = 13)

 for c in corners do
   writeln("corner at " c.xc, ", ", c.yc, "   len ", c.len);
 > corner at 100, 50   len 10
 > corner at 150, 75   len 13
 > corner at 125, 100   len 12
```

The full example is in ex_genclass.chpl. Compile and run it with

```
make ex_genclass

bin/ex_genclass
```

The default values for the parameters will of course be much higher in the actual program. We'll see another example of a generic class in the next section when we build a kd-tree.

Generic Procedures

A generic procedure is created through its argument list in one of five ways. You can also add clauses that restrict which generic function is called when the compiler resolves overloads. First, the intent of an argument can be `type` or `param`. If `type`, then you must pass a type for that argument and the argument can be used as a type in variable declarations and casts.

```
proc castto(type t, val : real) {
  return val : t;
}

writeln(castto(int(32), 314.159));
> 314

writeln(castto(uint(8), 314.159));
> 58
```

If `param`, then you must pass a parameter expression and compilation occurs with that value. Second, you can omit the type of an argument in which case the value provided by the caller determines it. The language spec (Formal Arguments without Types) gives a good example of a procedure to construct an arbitrarily sized tuple filled with a value of any type.

```
proc filltuple(param cnt: int, x) {
  var result : cnt * x.type;
  for param i in 1..cnt do result(i) = x;
  return result;
}

var ex1 = filltuple(4, 2.178);
```

```
writeln("  fill 4-tuple ", ex1);
>   fill 4-tuple (2.178, 2.178, 2.178, 2.178)

writeln("  type is ", ex1.type : string);
>   type is 4*real(64)
```

To print the type we only need to cast it to a string. The parameter sets the size of the tuple, and the unspecified type of the second argument determines the type of each element. You'll find this example in ex_genproc.chpl. The third way to define a generic procedure is with a query for a type. Like an argument without a type, the procedure call will determine which type is substituted for the query. Instead of getting the type through the variable, you can then use the type name directly. A queried type is indicated by a question mark preceding an identifier and that identifier is used in place of a type. The `filltuple()` example could be written

```
proc filltuple(param cnt : int, x : ?t) {
  var result : cnt * t;
  for param i in 1..cnt do result(i) = x;
  return result;
}
```

Queried arguments are used extensively in the Chapel source code. Here's the `safeAdd()` procedure in CHPL_HOME/modules/internal/ChapelUtil.chpl, which checks if it's possible to add two integers of the same type without overflow. The query on the first argument sets a requirement on the second – both must have the same type. The procedure `isIntegralType()` is defined in CHPL_HOME/modules/standard/Types.chpl; it is generic and returns `true` for `int` or `uint` types.

```
proc safeAdd(a : ?t, b : t) {
  if !isIntegralType(t) then
    compilerError("Values must be of integral type.");
  if a < b {
    if b >=0 {
      return true;
    } else {
      if b < min(t) - a {
        return false;
      } else {
        return true;
      }
    }
  } else {
    if b <= 0 {
      return true;
    } else {
      if b > max(t) - a {
        return false;
      } else {
        return true;
      }
    }
  }
}
```

```
proc isIntegralType(type t) param
  return isIntType(t) || isUintType(t);
```

The fourth way to specific a generic procedure is to pass a generic class or record as an argument type. We will have a procedure that takes a `chunkarray` of corners to suppress, which makes this generic.

```
proc suppress_corners(rawcnr : chunkarray(corner)) : chunkarray(corner) {
  /* ... */
}
```

You can query details about the generic type. If we had written

```
proc suppress_corners(rawcnr : chunkarray(?t)) : chunkarray(t) {
  /* ... */
}
```

then Chapel would assign `t` to `corner` during compilation and create a version of this procedure with the specific type `chunkarray(corner)`. The definition of the `&=` operator in CHPL_HOME/modules/internal/ChapelBase.chpl provides an example of this detailed query. If the width of the integer arguments is the same, then the primitive low-level function can be called. If not, then casting is needed and Chapel uses the explicit form.

```
inline proc &=(ref lhs : int(?w), rhs : int(w)) {
  __primitive("&=", lhs, rhs);
}
inline proc &=(ref lhs : uint(?w), rhs : uint(w)) {
  __primitive("&=", lhs, rhs);
}
inline proc &=(ref lhs, rhs) {
  lhs = lhs & rhs;
}
```

This detailed query also applies to the fifth type of generic procedure, which has an argument with an array of a generic type.

To help with overloading, generic procedures may have a `where` clause between the argument list/return type and block of code. The conditions in the clause must be true for Chapel to pick that version of the procedure to use. This technique is used extensively in the built-in modules. Consider the prototypes for the sorting functions found in CHPL_HOME/modules/packages/Sort.chpl.

```
proc heapSort(Data : [?Dom] ?elType, comparator : ?rec=defaultComparator) {
     where Dom.rank != 1 {
  compilerError("heapSort() requires 1-D array");
}
```

The clause restricts HeapSort to one dimensional arrays. The queries are used to extract the domain from the array to be sorted, the type of the array's elements, and the comparison function/record. In CHPL_HOME/modules/internal/ChapelTuple.chpl you'll find as one of the addition operators

```
inline proc +(x : _tuple, y : x(1).type) where isHomogeneousTuple(x) {
  /* ... */
}
```

Here the function `isHomogeneousTuple()` returns `true` if all the members of the tuple `x` have the same type. The type of `y` says that it too must be the same by matching the first tuple member, so that we can add the values safely. As a final example, the fillRandom procedure in CHPL_HOME/modules/standard/Random.chpl which fills an array with random values has the prototype

```
proc fillRandom(arr : [], seed : int(64) = SeedGenerator.oddCurrentTime,
                param algorithm = defaultRNG)
     where isSupportedNumericType(arr.eltType) {
  /* ... */
}
```

Implementation (fast_v2, test_fast_v2)

Our suppression of corners begins by modifying fast_v1 to generate details about each corner, namely its location and the maximum pixel difference figure of merit. We'll sort them first by their x position, then their y, so we can quickly abandon pairs whose separation along either coordinate is more than the parameter `suppsep`, which will typically be between one and two times the FAST radius (if one then the points must be within each other's FAST circle, if two then the circles will touch). We'll store all the close corners in a group, using a modified priority heap to keep track of the biggest, with ties resolved randomly. When all pairs have been processed we take the biggest and mark them on the greyscale image.

Now the details of the implementation, which is found in fast_v2.chpl. Compile it with

```
make fast_v2
```

The program takes the same command line options as fast_v1, plus `--suppsep=#` for the maximum difference between the x or y coordinates of the pair.

The procedure `is_corner_with_details()` has been added to return a `corner` record with the extra information we need. In addition to the center and figure of merit, called `dpix`, we track the starting point and length of the consecutive run of differing pixels. The procedure `is_corner()` is now a wrapper around this function that throws away the details.

`find_corners()` takes the details and stores them in a `chunkarray`. The corner detection is done in parallel, which is why the `append()` method in the `chunkarray` needs to be synchronized. It calls down to `suppress_corners()` to reduce the list, then cleans up before returning it.

```
proc find_corners(img : clrimage, spec : fastspec) : chunkarray(corner) {
  const circle = new circumference(spec.radius);
  const Ainside = img.area.expand(-spec.radius, -spec.radius);
  var corners = new chunkarray(corner);

  forall (y,x) in Ainside {
    var details : corner;
    if (is_corner_with_details(img, x, y, spec, circle, details)) {
      corners.append(details);
    }
  }
```

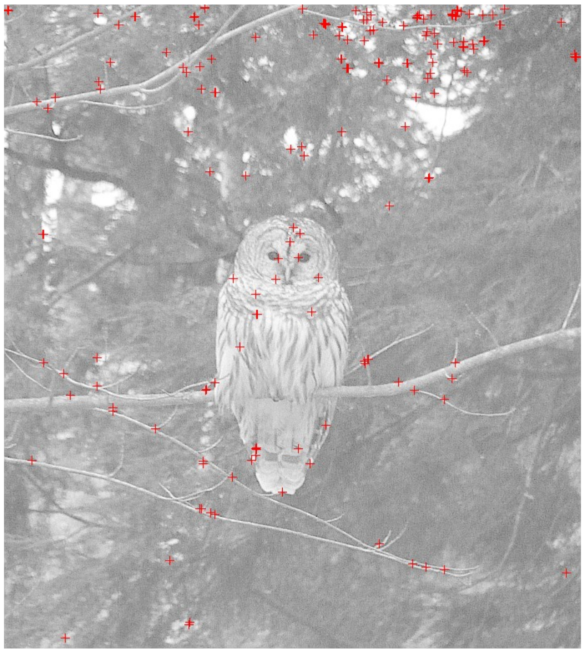
```
    var retval = suppress_corners(corners);

    delete circle;
    delete corners;
    return retval;
  }
```

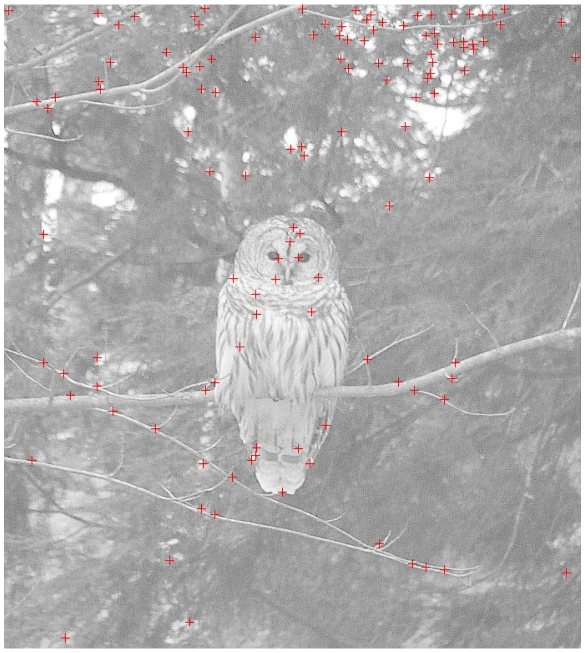
The suppress_corners() procedure uses a standard sort module, defining the comparison relationships between corners so that sorting is first by x, then y. It looks at all pairs that are close enough. The corner with the better figure of merit is registered as the parent of the other. We use a flat array whose elements represent the index of the parent, or -1 if the corner is a root (or hasn't yet been assigned), to hold the tree. Since we don't care about maintaining a proper priority heap through all levels of the tree – only the root needs to be the biggest – we go up from each of the two corners to their roots (which may well be the corners themselves if they haven't been placed yet in a tree) and put the smaller under the bigger; this is done in the set_parent() procedure. After processing all corners we then assemble those without any parents, in other words the best of each group.

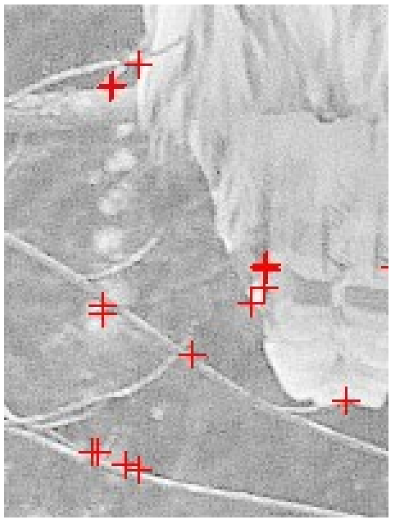
```
  proc suppress_corners(rawcnr : chunkarray(corner)) : chunkarray(corner) {
    var cnr = rawcnr();
    var suppcnr = new chunkarray(corner);
    var parent : [rawcnr.Ldata] int;

    /* Comparison function for sort, first by center x, then y. */
    proc DefaultComparator.compare(c1 : corner, c2 : corner) : int {
      if (c1.xc == c2.xc) then return (c1.yc - c2.yc);
      else return (c1.xc - c2.xc);
    }

    sort(cnr);

    /* Remember, sets all elements to -1. */
    parent = -1;

    for c1 in cnr.domain {
      for c2 in cnr.domain[c1+1..] {
        /* Sorted by x, so we don't have to check c2.xc < c1.xc. */
        if ((cnr(c1).xc + suppsep) < cnr(c2).xc) then break;
        if (abs(cnr(c1).yc - cnr(c2).yc) <= suppsep) {
          set_parent(cnr, c2, c1, parent);
        }
      }
    }

    for c1 in cnr.domain do
      if (parent(c1) < 0) then suppcnr.append(cnr(c1));

    return suppcnr;
  }
```

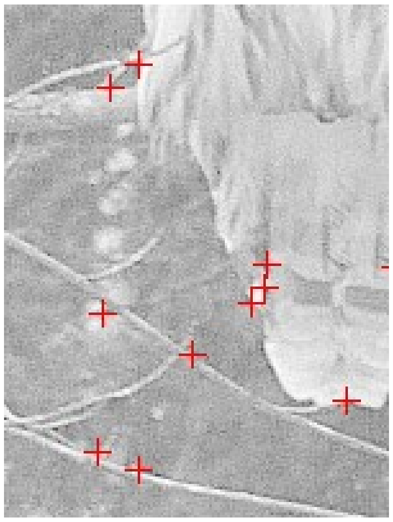
`mark_corners()` no longer does the FAST analysis. It now takes a list of corners, which could be the raw or suppressed sets, and places a small colored cross in a copy of the greyscale image.

If you run this version on the test image with the default threshold, the number of corners drops from 181 to 123. The suppression is clearest in the trees at the top, although there are doubles removed around the owl as the detail image shows. (For `--thr=15` the number drops from 724 to 361, for `--thr=25` from 36 to 33.)


FAST corners before suppression, thr=20


FAST corners after suppression


Detail before suppression


Detail after suppression

Wrap-Up

Like the Gabor filters, a corner detector is a per-pixel operation and therefore one that's very easy to have run in parallel. We've seen how an iterator works in Chapel, and how to implement a custom one. We've also seen how standard class methods allow it to be used to drive a loop. We've written a generic class to store the corners as they're generated, so we can eliminate many of the groupings that appear in natural scenes. We've learned:

- An iterator is defined like a procedure. It uses `yield` statements to provide a value and a point where execution will resume on the next call. return ends the process. You can call an iterator from a `for`, `forall`, or `coforall` statement.

- Classes have a `this()` method which is called when you use the instance as a function. You can pass arguments to this function.

- If a class has a `these()` iterator then you can use an instance where you would use a range in a loop statement. The loop will cycle through that iterator.

- Classes and records are generic if they contain a type or parameter member, or a member whose type is unassigned (and which gets defined by the value passed to the constructor when instantiating the class). They have a default type constructor that creates well-typed versions during compilation.

- Procedures are generic if they contain a type or parameter argument intent, an argument without a type (where the corresponding value in the caller is used), or a generic type as an argument. The argument list can contain queries, which are `?identifiers` that are replaced by the correct type and where the identifier can be used as a type either in the argument list or in the body. The queries can also refer to the details behind generic types, such as the size of integers, and can enforce relationships between argument types for overloaded procedures.

- `where` clauses define requirements that generic arguments must fulfill in order for that procedure to be called. This is used to remove ambiguity between overloaded generics.

A corner detector is an example of a feature detector that highlights unique aspects of an image. We'll use it to look for common points in two different images by randomly matching points and looking for the best agreement.

Exercises

1. Create a “spot” detector that covers the circumference and its interior. This can be used to implement a SUSAN corner detector (see http://citeseer.ist.psu.edu/viewdocs/summary?doi=10.1.1.24.2763 for the original paper describing the detector and its implementation). How does the SUSAN compare with the FAST?

2. Using a square kernel for a Gaussian low-pass filter leads to some distortion of the underlying circular symmetry. Write a “spot” convolver that takes a circular kernel.

Files

The programs and image for this chapter can be found here. A zip file is here.

A PDF version of this text is here.

RANSAC FEATURE MATCHING (ransac)

Introduction

Corners are an example of a feature which should be stable between images, independent of illumination, camera, or scene composition. This means we should be able to use them to align different images, for stitching them together in a panorama or matching up objects or comparing their content. If we can identify matching corners, we can determine how to map one image onto another, accounting for scaling, translation, and rotation. But with hundreds of corners an exhaustive search for the best fit is impossible, and there are certainly many that do not have corresponding matches and which, if used, would give a bad mapping.

RANSAC, short for Random Sample Consensus, solves these problems by sampling the solution space using a figure of merit that ignores mismatches and outliers. It forms a hypothesis, a proposed coordinate transform, by picking a random subset of corners and assuming they match. It then evaluates the hypothesis by counting the number of close pairs after transformation, nearer than some given distance. It makes many trials and picks the one with the most matches, then refines the result with the pairs.

Say that we have a linear model accounting for scale and translation between the coordinates (x, y) of image1 and (v, w) of image2

```
v = sx * x + dx
w = sy * y + dy
```

where sx and sy are scale factors and dx and dy offsets. We need two pairs of corners to determine these four unknowns, and once we have them we can transform the corners in image1 into the image2 plane. If a corner ends up closer to a corner in image2 than some threshold, we count that pair as a successful match. When done comparing all corners we can refine the model by calculating the best fit of its parameters over the matches using a linear regression

The RANSAC algorithm does assume there is a reasonably large overlap between corners in the two images and that noise (aka false corners) and outliers do not dominate. If the ideal or actual transform does not have many matches, then most trials will fail or give poor results and more will have to be performed. We haven't used this technique before, though, so part of the reason for this exercise is to see how well the approach works.

In this chapter we implement RANSAC for two models, one with the shift-translation (ST) model above and one with an affine transforms with rotation-shift-translation (RST). We won't be needing any parts of Chapel that we haven't already seen, but we'll find we use most of the features we've talked about.

The directory for this chapter is `ransac`. The programs will use the C functions in img_png_v3.[ch] for reading and writing PNG images and the color library ip_color_v3.chpl from the Color Conversion chapter. The FAST analysis has also been pulled out into a separate library, ip_corner.chpl. Use `make name` to build the programs, leaving off the '.chpl' suffix. Executables get put in the `bin` sub-directory, intermediate files in `build`. This exercise uses a set of transformed images, with bonds1.png the original, base image.

The programs and images are in a tarball here (as zip file). Here is a PDF version of this chapter.

Aside: kd-Trees (kdtree, test_kdtree)

The key step in the RANSAC algorithm is to identify matching pairs of corners after transforming the coordinates of one to the other's plane. This means searching a 2D plane for the nearest match, which is not well supported by normal data structures. A kd-tree partitions the plane efficiently. It is a binary tree where the coordinate used to divide the plane (volume, hypervolume) changes from level to level: in our case the root node splits along x, the first layer by y, the second by x, the third by y, and so forth. This allows searching in O(log n) time on average and O(n) worst case. Building the tree can be done in O(n log n) if a median search in linear time is available, as we saw at the start of the tutorial (An Example). Such a median search does exist but is a bit complicated to implement; a new method has just appeared with the same order of complexity.

Construction

Brown has recently published a new way to construct the tree, claiming that although it takes O(kn log n) time, where k is the number of dimensions, it is somewhat more efficient than the median search for k <= 4. [Russel Brown, "Building a Balanced k-d Tree in O(kn log n) Time", Journal of Computer Graphics Techniques, Vol. 4, No. 1, 2015.] It makes a sort of the corner coordinates along each axis at the start, and then shuffles the indices for each level of the tree, never doing a second sort.

The sorting can be done by any technique. Ties are resolved in a cyclic order through the coordinates. If we have three, then sort one first by x with ties resolved by y then z (xyz), the second by y with z then x breaking ties (yzx), the third by z using x then y (zxy). In Brown's paper and our implementation it is enough to store indices to the corners in an array, rather than moving the corner data structures around. Note that because we are sorting by index and not value the standard methods in the Sort module cannot be used; we'll modify the QuickSort example from the Parallel Programming chapter to work with indices.

Having sorted the corners by coordinate we then start building the tree layer by layer. We pick the middle, ie. median, element of the array as the node. We shuffle each sorted array so that those elements smaller than the chosen are in one half and those larger are in the other. What this means in practice is that we do nothing for the array whose primary sort coordinate is the one we've just used for the tree node (x, then y, then x, ...) because it's already in order. We scan the other arrays from start to finish, which guarantees we keep the same sort order, but use a comparison function based on the primary sort coordinate to determine if they go in the lower or upper half. We then recur on each of these halves; like the implementation in the Example, this can be done in parallel.

Here's an example of building a twelve node tree from 3D points. The initial sort gives

index	pt	xyz	yzx	zxy
1	(6, 4, 2)	12 (1, 9, 4)	10 (5, 1, 1)	10 (5, 1, 1)
2	(4, 7, 5)	3 (3, 1, 8)	3 (3, 1, 8)	6 (8, 9, 1)
3	(3, 1, 8)	8 (3, 4, 6)	4 (7, 2, 2)	9 (6, 3, 2)
4	(7, 2, 2)	5 (4, 3, 5)	9 (6, 3, 2)	1 (6, 4, 2)
5	(4, 3, 5)	2 (4, 7, 5)	5 (4, 3, 5)	4 (7, 2, 2)
6	(8, 9, 1)	10 (5, 1, 1)	1 (6, 4, 2)	7 (6, 4, 3)
7	(6, 4, 3)	9 (6, 3, 2)	7 (6, 4, 3)	12 (1, 9, 4)
8	(3, 4, 6)	1 (6, 4, 2)	8 (3, 4, 6)	11 (8, 6, 4)
9	(6, 3, 2)	7 (6, 4, 3)	11 (8, 6, 4)	5 (4, 3, 5)
10	(5, 1, 1)	4 (7, 2, 2)	2 (4, 7, 5)	2 (4, 7, 5)
11	(8, 6, 4)	11 (8, 6, 4)	6 (8, 9, 1)	8 (3, 4, 6)
12	(1, 9, 4)	6 (8, 9, 1)	12 (1, 9, 4)	3 (3, 1, 8)

Each column for the three coordinate permutations contains the points in order and the indices that refer back to the original list. In the zxy column, notice that (6, 3, 2) comes before (6, 4, 2) because the z and x values are the same, leaving y to resolve the difference.

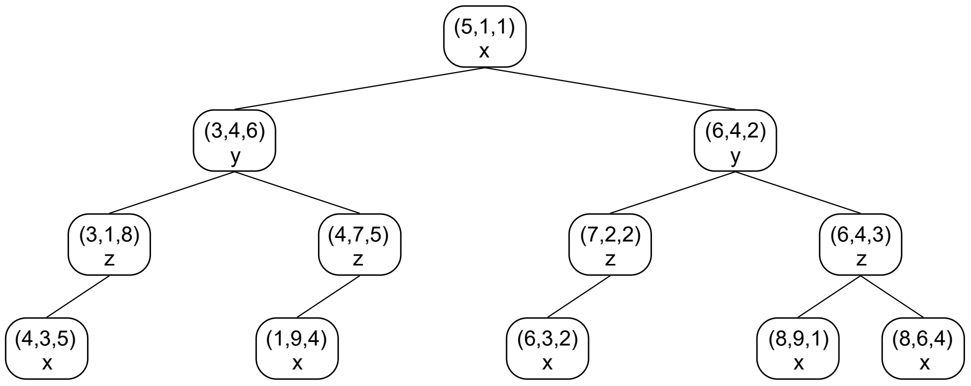
The first median is sixth in the sorted list for x (we can just take the lower of the two midpoints), or point 10 (5, 1, 1). We now have to shuffle yzx and zxy. The indices in all three arrays after the shuffle will be the same above and below the median, without disturbing their sorted orders. From the xyz list we find 12, 3, 8, 5, and 2 below the median and 9, 1, 7, 4, 11, 6 above. In the yzx array these are ordered 3, 5, 8, 2, 12 and 4, 9, 1, 7, 11, 6, and in the zxy array 12, 5, 2, 8, 3 and 6, 9, 1, 4, 7, 11. The shuffling is done by scanning the array from top to bottom, copying the index into the appropriate half of a temporary array and copying back when done.

```
       xyz             yzx             zxy
  12 (1, 9, 4)    3 (3, 1, 8)   12 (1, 9, 4)
   3 (3, 1, 8)    5 (4, 3, 5)    5 (4, 3, 5)
   8 (3, 4, 6)    8 (3, 4, 6)    2 (4, 7, 5)
   5 (4, 3, 5)    2 (4, 7, 5)    8 (3, 4, 6)
   2 (4, 7, 5)   12 (1, 9, 4)    3 (3, 1, 8)
  10 (5, 1, 1)
   9 (6, 3, 2)    4 (7, 2, 2)    6 (8, 9, 1)
   1 (6, 4, 2)    9 (6, 3, 2)    9 (6, 3, 2)
   7 (6, 4, 3)    1 (6, 4, 2)    1 (6, 4, 2)
   4 (7, 2, 2)    7 (6, 4, 3)    4 (7, 2, 2)
  11 (8, 6, 4)   11 (8, 6, 4)    7 (6, 4, 3)
   6 (8, 9, 1)    6 (8, 9, 1)   11 (8, 6, 4)
```

The middle of the top sub-array is the left child of the first node, and the middle of the bottom is the right child. We pick medians in each half of the y column now, giving 8 (3, 4, 6) as the left child and 1 (6, 4, 2) as the right (again rounding the midpoint down). We now reorder the x and z columns.

```
       xyz             yzx             zxy
   3 (3, 1, 8)    3 (3, 1, 8)    5 (4, 3, 5)
   5 (4, 3, 5)    5 (4, 3, 5)    3 (3, 1, 8)
                  8 (3, 4, 6)
  12 (1, 9, 4)    2 (4, 7, 5)   12 (1, 9, 4)
   2 (4, 7, 5)   12 (1, 9, 4)    2 (4, 7, 5)
  10 (5, 1, 1)
   9 (6, 3, 2)    4 (7, 2, 2)    9 (6, 3, 2)
   4 (7, 2, 2)    9 (6, 3, 2)    4 (7, 2, 2)
                  1 (6, 4, 2)
   7 (6, 4, 3)    7 (6, 4, 3)    6 (8, 9, 1)
  11 (8, 6, 4)   11 (8, 6, 4)    7 (6, 4, 3)
   6 (8, 9, 1)    6 (8, 9, 1)   11 (8, 6, 4)
```

When we have 3 nodes left in a sub-array, as we do on the next pass, then the middle becomes the root, the first its left child and the third its right. For the last sub-array, 7 (6, 4, 3) is placed in the third (z) layer with 6 (8, 9, 1) to its left and 11 (8, 6, 4) to its right. If there are only 2 nodes then we take the second as the root and the first as its left child, so 3 (3, 1, 8) goes as the left child of 8 (3, 4, 6), and 5 (4, 3, 5) as its left child. The complete tree looks like



Final kd-tree

Search

We follow the algorithm in "Algorithms in a Nutshell" by Heineman, Pollice and Selkow, O'Reilly, 2008 to use a kd-tree to find the node nearest a point. We first determine the leaf node where the point would be inserted (or, if the point matches a node, we have our answer). That is, traverse the tree from the root, going left if the point is less than the node in the cyclic sense we used building the tree, else right. There's no guarantee, however, that the leaf must be the closest, only that we've probably excluded half the tree. Now we have to check all parents of that node. Starting from the root, we measure the distance to the point, remembering it if it less than the best distance found so far. We then ask if the perpendicular distance, which is the difference in x coordinates if it's an x node or in y for a y node, is less than this minimum distance. If not, then we go down the tree in the direction of the point: the other side cannot be closer than the minimum. If it is, then we need to check both children.

```
find leaf node n under which point would be placed
dmin = distance between point, root
better = nearest(pt, root, dmin)
if better exists, return better
  else return n

nearest(pt, root, dmin)
  result = <none>
  if distance from root to pt < dmin, dmin = distance, result = root
  dp = perpendicular distance from pt to root
  if (dmin < dp) then
    if pt < root then better = nearest(pt, root.left, dmin)
      else better = nearest(pt, root.right, dmin)
    if better exists return better
  else
    better = nearest(pt, root.left, dmin)
    if better exists then result = better, dmin = distance pt to better
    better = nearest(pt, root.right, dmin)
    if better exists then result = better, dmin = distance pt to better
  return result
```

In the best and average case this is an O(log n) operation because we make two visits down the tree, once to find the leaf node and once to confirm that no parents are closer. In the worst case it becomes O(n) when we have to check both left and right children at every level when the perpendicular distance is less than the minimum. For small numbers of dimensions the kd-tree search is better than exhaustively checking all nodes, but as the dimensionality of the space increases to somewhere between 10 and 20, the chance of hitting the worst case quickly grows. Timing tests with the RANSAC program show a factor of 5 improvement over exhaustively searching the plane, from 41.1 s to 8.3 s per trial.

Implementation (kdtree, test_kdtree)

We'll implement the kd-tree with a generic class, using a generic record for each node. It will take a parameter for the number of dimensions (so the tree can be used outside this application) and a type for the data to store at each node; nodes will store this payload, the point, and some flags to indicate if they're leaf nodes. The tree will be built by first loading data and then constructing it. Since we cannot add data after the construction we'll need to use an internal flag to prevent this.

The class will have five public methods:

```
kdtree() - constructor for the generic elements and number of points
add_node() - stage a point and its payload to put in the tree
assemble_tree() - sort the nodes into the kd-tree
find_nearest_point() - return payload in the node closest to a point
dump() - print a table with the tree, meant for debug
```

The node record is

```
record kdpoint {
  type eltType;
  param ndim : int;
  var flags : uint(8);
  var pt : ndim * int;
  var val : eltType;
}
```

where `val` is the payload. The type and parameter are set when we use the node in the class

```
class kdtree {
  type dataType;
  param ndim : int;
  var Ltree : domain(rank=1);
  var tree : [Ltree] kdpoint(eltType=dataType, ndim=ndim);
  var Ldata : domain(rank=1);
  var data : [Ldata] kdpoint(eltType=dataType, ndim=ndim);
  var sortind : [1..ndim][Ldata] int;
  var nstored : int;
  var storable : bool = true;
}
```

Here we've separated the nodes into a unsorted array `data` and the kd-tree `tree`. The domains for each are defined separately so we can change them when we know the number of points. We use a flat array to store the tree where the left child of node `p` is at index `2*p` and the right at `2*p+1`. `nstored` will track how many points we've added, or where to store the next point that comes in. The `storable` flag will be set `false` when we build the tree; we'll use it to tell whether we can add more points. The `sortind` array stores the sorted indices we need.

You'll find the code implementing the five public methods in kdtree.chpl. We don't intend to go through it line by line, but do want to make several comments.

In the constructor we make sure there are at least two levels to the tree, even if it stores only one element. The routine that builds the tree assumes this (`place_node()` when the size of the array is 1). The tree otherwise is full, ie. has `2 ** depth - 1` nodes, where `depth` is the log2 of the number of elements, rounded up. In the Chapel standard math library `log2(val:int)` returns the largest power-of-two smaller than the value; it does not automatically cast to real.

The `add_node()` method simply checks that we haven't assembled the tree yet and copies the new point and value into the staging area, the `data` array.

`assemble_tree()` begins by setting up and sorting the indices of the points. It then calls `place_node()` which will determine where to split the array, place that point in the tree, shuffle the points and recur on each half. Unlike our QuickSort example and the kd-tree example at the start, we pass only the range and not a sliced array because `sortind` is a member. But the core logic – shuffle and recur in parallel using `cobegin` – is the same.

```
/* Lsort is subrange of data to process, cbase the primary sort
   coordinate, and pos the index in tree to put the point. */
proc place_node(Lsort : range, cbase : int, pos : int) {
  if (3 == Lsort.size) {
    /* first element to left child, middle to pos, last to right */
  } else if (2 == Lsort.size) {
    /* first element to left child, middle to pos, right is empty */
  } else if (1 == Lsort.size) {
    /* element to pos, left and right empty */
  } else {
     const midpt = (Lsort.first + Lsort.last) / 2;
     const cnext = if (cbase == ndim) then 1 else (cbase + 1);
     shuffle_sortind(Lsort, cbase, midpt);
     tree(pos) = data(sortind(cbase)(midpt));
     tree(pos).flags = cbase : uint(8) | HAS_L | HAS_R;
     cobegin {
       place_node(Lsort[..midpt-1], cnext, 2*pos);
       place_node(Lsort[midpt+1..], cnext, 2*pos+1);
     }
   }
 }
```

Since we're using a flat array for the tree, we need flags in the `kdpoint` record to mark if the node is a leaf (`IS_LEAF`), is empty (`IS_NIL`), or if it has a left or right child (`HAS_L` and `HAS_R`). These flags are defined as constants inside the class. We don't use an enumeration because of the problems using the constant as an `int` (Chapel can't figure out the type and you would need to cast every usage). We reserve four bits for coordinate planes, supporting up to 16. Although `shuffle_sortind()` modifies the array, it is safe to do so in parallel during the recursion because the ranges will not overlap.

The sorting method `sort_kdpoints()` is a modified QuickSort. For convenience we've defined three sets of comparison functions, with `point_<comparison>()` being the fundamental group. They use a loop through the coordinates, resolving the comparison with the first unequal pair.

```
inline proc point_lt(pt1 : ndim * int, pt2 : ndim * int, cbase : int,
                     noeq = false) : bool {
  for craw in cbase..(cbase+ndim-1) {
    const coff = if (craw > ndim) then (craw - ndim) else craw;
    if (pt1(coff) != pt(coff)) then
      return pt1(coff) < pt2(coff);
  }
  if (noeq) then halt("kd-tree does not accept equal points");
  else return true;
}
```

Here `coff` is wrapped to the range `1..ndim`. Since `cbase` varies, we cannot use a `for param` to unwrap the loop. This is somewhat inefficient for a small number of dimensions, but it is a general solution that works for all `ndim`. The `noeq` flag is needed because we don't want duplicate points while building the tree, but we do want to allow them when searching for the nearest node. The other two sets are wrappers around this, `dataind_<comparison>()` using the base coordinate and array index to get points from the `data` array via `sortind`, and `kdpoint_<comparison>()` that pulls the points from the nodes directly.

In the `shuffle_sortind()` procedure we've made one optimization by noting that the indices in all coordinate arrays are the same below or above the midpoint, only their order differs. We don't need to compare the points with the median while shuffling, because we already know that the primary coordinate array is in the correct order and tells us which half to place the index in. We use an intermediate array `lt` to mark them, and the decision to shuffle becomes a look up into `lt`.

The searching procedure `find_nearest_point()` follows the algorithm described above. It is straightforward code. Perhaps the only two points to make about Chapel is the use of a `for param` in the distance-squared calculation `dist2()`, and a clean definition of the perpendicular distance given the coordinate `cbase` used by the tree node

```
proc perpdist2(pt : ndim * int, node : int) : int {
  const cbase = tree(node).flags & COORD;
  return (pt(cbase) - tree(node).pt(cbase)) ** 2;
}
```

Here COORD is a bit mask to extract the coordinate we packed into the flags member.

The program test_kdtree.chpl is a self-checking test bench for several functions in kdtree. Compile and run it with

```
make test_kdtree

bin/test_kdtree
```

Aside: Transforms and Best Fits

RANSAC matching is a three-step process, beginning with an estimate of the transform between two images, then checking if it is a good mapping, and finally refining it with the pairs of matching features. There are two transforms we'll want to use, a linear coordinate mapping accounting for scaling and translation (ST) and a mapping that also includes rotation (RST). The ST model is simpler and was used to develop and test the program first. The math behind the steps involves solving a linear system for the first, evaluating the transform for the second, and a best-fit error minimization regression for the third.

Repeating the ST transform from (x, y) coordinates to (v, w),

$$v = sx \cdot x + dx$$

$$w = sy \cdot y + dy$$

Note that the order of the operations is important; a TS model would give v = sx * (y + dx) which has a different interpretation for the shift dx. This is just a linear equation which needs two pairs of points to fix the constants, (x1, y1), (v1, w1) and (x2, y2), (v2, w2)

$$v1 = sx \cdot x1 + dx$$

$$v2 = sx \cdot x2 + dx$$

$$w1 = sy \cdot y1 + dy$$

$$w2 = sy \cdot y2 + dy$$

Solving

$$sx = (v1 - v2) \; / \; (x1 - x2)$$

$$dx = v1 - sx \cdot x1 = v2 - sx \; x2$$

$$sy = (w1 - w2) \; / \; (y1 - y2)$$

$$dy = w1 - sy \cdot y1 = w2 - sy \; y2$$

Refining these values given many matching pairs is the same as performing a linear regression which minimizes the square error function

$$L = \sum_{i=1}^{n} (v_i - sx \cdot x_i - dx)^2$$

Take the partial derivative of L for each constant

$$\frac{\partial L}{\partial sx} = \sum_{i=1}^{n} -2 x_i \cdot (v_i - sx \cdot x_i - dx)$$

$$\frac{\partial L}{\partial dx} = \sum_{i=1}^{n} -2 \, (v_i - sx \cdot x_i - dx)$$

and set them to zero.   Re-writing,

$$\sum_{i=1}^{n} -2 \; x_i \, (v_i - sx \cdot x_i - dx) = 0$$

$$\sum_{i=1}^{n} -2\,(v_i \;-\; sx \cdot x_i \;-\; dx) \;=\; 0$$

or

$$\sum_{i=1}^{n} x_i v_i \;-\; sx \sum_{i=1}^{n} x_i^{\,2} \;-\; dx \sum_{i=1}^{n} x_i \;=\; 0$$

$$\sum_{i=1}^{n} v_i \;-\; sx \sum_{i=1}^{n} x_i \;-\; n\;dx \;=\; 0$$

Solving for sx and dx gives

$$dx \;=\; (\sum_{i=1}^{n} v_i \;-\; sx \sum_{i=1}^{n} x_i) \;/\; n$$

$$sx \;=\; (n \sum_{i=1}^{n} x_i v_i \;-\; \sum_{i=1}^{n} x_i \sum_{i=1}^{n} v_i) \;/\; (n \sum_{i=1}^{n} x_i^{\,2} \;-\; (\sum_{i=1}^{n} x_i)^2)$$

The solution for y and w is the same, just changing variables

$$dy \;=\; (\sum_{i=1}^{n} w_i \;-\; sy \sum_{i=1}^{n} y_i) \;/\; n$$

$$sy \;=\; (n \sum_{i=1}^{n} y_i w_i \;-\; \sum_{i=1}^{n} y_i \sum_{i=1}^{n} w_i) \;/\; (n \sum_{i=1}^{n} y_i^{\,2} \;-\; (\sum_{i=1}^{n} y_i)^2)$$

The RST transform is a specific case of an affine transform which preserves lines and the ratios between distances. The general equations are

$$v \;=\; sxx \cdot x \;+\; sxy \cdot y \;+\; dx$$

$$w \;=\; syx \cdot x \;+\; syy \cdot y \;+\; dy$$

and the specific values of the coefficients are

$$v \;=\; sx \cdot \cos(\theta) \cdot x \;+\; sx \cdot \sin(\theta) \cdot y \;+\; dx$$

$$w \;=\; -sy \cdot \sin(\theta) \cdot x \;+\; sy \cdot \cos(\theta) \cdot y \;+\; dy$$

That is,

$$sxx \;=\; sx \cdot \cos(\theta)$$

$$sxy \;=\; sx \cdot \sin(\theta)$$

$$syx \;=\; -sy \cdot \sin(\theta)$$

$$syy = sy \cdot \cos(\theta)$$

This is a system of four equations with three unknowns, so we have the additional requirement that

$$sxx \cdot sy = sx \cdot syy$$

with the inverse relationships

$$\tan(\theta) = sxy / sxx = -syx / syy$$

$$sy = (syy/sxx) \cdot sx = -(syx/sxy) \cdot sx$$

The RST mapping is a rigid rotation of the plane when this condition holds. The affine transform also includes a skewing or slanting which this model does not permit. Once again the order of the three operations is important.

This system has five unknowns, which we can get from three pairs of matching points that also satisfy the additional requirement.

$$v1 = sxx \cdot x1 + sxy \cdot y1 + dx$$

$$v2 = sxx \cdot x2 + sxy \cdot y2 + dx$$

$$v3 = sxx \cdot x3 + sxy \cdot y3 + dx$$

$$w1 = syx \cdot x1 + syy \cdot y1 + dy$$

$$w2 = syx \cdot x2 + syy \cdot y2 + dy$$

$$w3 = syx \cdot x3 + syy \cdot y3 + dy$$

The solution of the first three is

$$dx = \left(v3 - p3 - \frac{q3}{q2}(v2 - p2)\right) / \left(r3 - \frac{q3}{q2} r2\right)$$

$$sxy = (v2 - p2 - r2 \cdot dx) / q2$$

$$sxx = (v1 - sxy \cdot y1 - dx) / x1$$

where the intermediate values p, q, and r are

$$p2 = x2 \cdot v1 / x1$$

$$q2 = y2 - (y1 \cdot x2 / x1)$$

$$r2 = 1 - (x2 / x1)$$

$$p3 = x3 \cdot v1 / x1$$

$$p3 = y3 - (y1 \cdot x3 / x1)$$

$$r3 = 1 - (x3 / x1)$$

For the second set of three

$$dy = \left(w3 - s3 - \frac{q3}{q2}\left(w2 - s2\right)\right) / \left(r3 - \frac{q3}{q2} r2\right)$$

$$syy = \left(w2 - s2 - r2 \cdot dy\right) / q2$$

$$syx = \left(w1 - syy \cdot y1 - dy\right) / x1$$

using q and r from above and

$$s2 = x2 \cdot w1 / x1$$

$$s3 = x3 \cdot w1 / x1$$

Refining these values can also be done by minimizing the error function

$$L = \sum_{i=1}^{n} \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right)^2$$

which has partial derivatives to the constants

$$\frac{\partial L}{\partial sxx} = \sum_{i=1}^{n} -2 x_i \cdot \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right)$$

$$\frac{\partial L}{\partial sxy} = \sum_{i=1}^{n} -2 y_i \cdot \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right)$$

$$\frac{\partial L}{\partial dx} = \sum_{i=1}^{n} -2 \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right)$$

Setting these to zero

$$\sum_{i=1}^{n} -2\ x_i \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right) = 0$$

$$\sum_{i=1}^{n} -2\ y_i \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right) = 0$$

$$\sum_{i=1}^{n} -2 \left(v_i - sxx \cdot x_i - sxy \cdot y_i - dx\right) = 0$$

or

$$\sum_{i=1}^{n} x_i v_i - sxx \sum_{i=1}^{n} x_i^2 - sxy \sum_{i=1}^{n} x_i y_i - dx \sum_{i=1}^{n} x_i = 0$$

$$\sum_{i=1}^{n} y_i v_i \;-\; sxx \sum_{i=1}^{n} x_i y_i \;-\; sxy \sum_{i=1}^{n} y_i^2 \;-\; dx \sum_{i=1}^{n} y_i \;=\; 0$$

$$\sum_{i=1}^{n} v_i \;-\; sxx \sum_{i=1}^{n} x_i \;-\; sxy \sum_{i=1}^{n} y_i \;-\; n\,dx \;=\; 0$$

Solving for dx and inserting it in the other two equations reduces it to two unknowns

$$k \;+\; sxx \cdot l \;+\; sxy \cdot m \;=\; 0$$

$$p \;+\; syx \cdot q \;+\; syy \cdot r \;=\; 0$$

The solution to this, again using intermediate values to help, is

$$sxx \;=\; (r \cdot k \;+\; m \cdot p) \;/\; (m^2 \;-\; r \cdot l)$$

$$sxy \;=\; -(k \;+\; sxx \cdot l) \;/\; m$$

$$dx \;=\; (\sum_{i=1}^{n} v_i \;-\; sxx \sum_{i=1}^{n} x_i \;-\; sxy \sum_{i=1}^{n} y_i) \;/\; n$$

where

$$k \;=\; n \sum_{i=1}^{n} x_i v_i \;-\; \sum_{i=1}^{n} x_i \sum_{i=1}^{n} v_i$$

$$l \;=\; (\sum_{i=1}^{n} x_i)^2 \;-\; n \sum_{i=1}^{n} x_i^2$$

$$m \;=\; \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i \;-\; n \sum_{i=1}^{n} x_i y_i \;=\; q$$

$$p \;=\; n \sum_{i=1}^{n} y_i v_i \;-\; \sum_{i=1}^{n} y_i \sum_{i=1}^{n} v_i$$

$$r \;=\; (\sum_{i=1}^{n} y_i)^2 \;-\; n \sum_{i=1}^{n} y_i^2$$

and q turns out to be equal to m. syx, syy, and dy have the same solution except that w is used in place of v.

Implementation (ransac_st, ransac_rst)

Like the kd-tree, implementing the RANSAC algorithm doesn't require any new Chapel features. The ransac_st program implements the scaling-translation mapping and ransac_rst the rotation-scaling-translation. To compile them

```
    make ransac_st
```

```
or  make ransac_rst
```

The first procedure called once we have lists of corners in each image is align_corners(). In pseudocode it does

```
proc align_corners(corners1 : [?Lcnr1] corner, corners2 : [?Lcnr2] corner,
                   out bestmap : mapinfo) {
  build kdtree for corners2
  for i in 1..ntry {
    do {
      pick seeds and determine mapping
      if mapping valid {
        map corners1 onto corners2 plane
        match up corners
        if have enough matches, break out of while (trial is done)
      }
    } until too many failures to get a match (trial as a whole fails)
  }
  select best trial
  re-create mapped corners and matching pairs
  refine mapping based on pairs
  return mapping
}
```

When picking seeds we do not do so blindly; this would cause too many failed attempts. At a minimum we require the corners be similar: they must have about the same length and orientation (as determined by the sequence's start point) and must both be brighter or darker than the corner. We also require that the mapping make sense, as checked in `valid_map()`. This means the scaling in x and y must be about the same and bounded, and for RST that the extra condition on the affine coefficients holds. The bounds on sx and sy are necessary because there is a pathological situation where a large part of one image scales to a small, dense cluster of corners in the other, generating many false matches. This consistency check is done after picking both pairs in ST, but must be used to screen the third pair of seeds for RST. This requirement has a substantial cost, since we must calculate and validate the mapping for each corner (O(n)), but is necessary because the chance of finding a valid corner at random is too small. These screenings take the form of checking each potential corner and adding it to a list if it's acceptable, then picking which to use at random from the list.

Although the corner detection and validation parameters will change with the mapping, we ignore the effect. For example, limiting how much the start point of the ring can shift will ultimately restrict the rotations that will be accepted, and this parameter must be looser in RST than in ST. Our assumption is that the features are stable over the mapping values we allow, but part of the experimentation with this algorithm is to see if that is true.

We use the formulas in the previous section to convert our seeds into the transform, and the mapping of corners1 just applies the transform. Matching is done by taking the transformed corners1 and finding the nearest point in corners2 using the kdtree. If this node has not yet been paired and the corners are similar, then we record it and increment the number of matches. This is a little sub-optimal in that there may be multiple potential pairings, and we're just taking the first, not the best. The assumptions that `mapsep` is small and corner suppression separates the corners should reduce the chance of have multiple matches, but if the scaling differs very much from 1 then this may not be true. Checking this is also part of our evaluation of the algorithm.

Once done, if this number is more than a given fraction `matchfrac` of the number of corners, then we declare the trial a success, otherwise we record a failure and make another attempt. Once all tries are done, we pick the one with the greatest number of matches as the overall winner. We re-create the list of matching pairs and run the regression fit on them to generate the final mapping.

In addition to the command line arguments for changing the FAST detector, the RANSAC algorithm adds many more

```
--ntry=<int>          number of times to try to get a match
--nfail=<int>         number of failed seedings before abandoning try
--mapsep=<int>        max distance between matches after mapping 1 to 2
--seedsep=<int>       (RST) smallest distance between seeds
--matchfrac=<real>    fraction of corners that must match to accept
--dlen=<int>          difference in corner sequence length to be similar
--dst<int>            difference in start point coords to be similar
--outname=<string>    (RST) if provided, file name of image showing matches
```

There are also two parameters that define the maximum ratio between sx/sy or sy/sx (`SAME_S_RATIO`) and the absolute value for sx or sy or their inverses (`MAXSCALE`). The RST version also adds a limits on the mismatch between the affine scaling coefficients (`SAME_RIGIDITY`).

You'll find three test programs provided. test_map_rst is a self-checking test bench for the conversion between affine coefficients and RST description. With test_fit_[r]st you provide the mapping on the command line and the program runs just the corner mapping and matching, reporting the number of matches found. These programs require the full ransac_[r]st program. They define a second `main()` procedure that the `--main-module` compiler flag picks.

```
chpl --main-module test_fit_st -o bin/test_fit_st test_fit_st.chpl
     img_png_v3.h build/img_png_v3.o -lpng
```

To run test_fit_[r]st, use the appropriate input files and threshold and provide the mapping you wish to try. The programs will report the number of corners that match.

Operation

We've manually rotated, scaled, and shifted the base image to test the programs. The images have a moderate number of corners, and because they are based on the same image there should be good agreement between the features. The images are

		rot	sx	sy	dx	dy	# corners	after suppress
bonds1	base	0	1.00	1.00	0	0	2770	747
bonds2_s	S	0	1.25	1.30	0	0	1568	562
bonds2_t	T	0	1.00	1.00	150	300	2772	748
bonds2_st	ST	0	1.25	1.30	150	300	1575	565
bonds2_rt	RT	30	1.00	1.00	0	700	2278	651
bonds2_rst	RST	30	1.25	1.30	150	1210	983	398

Either program should be run with `--thr=10`, for example

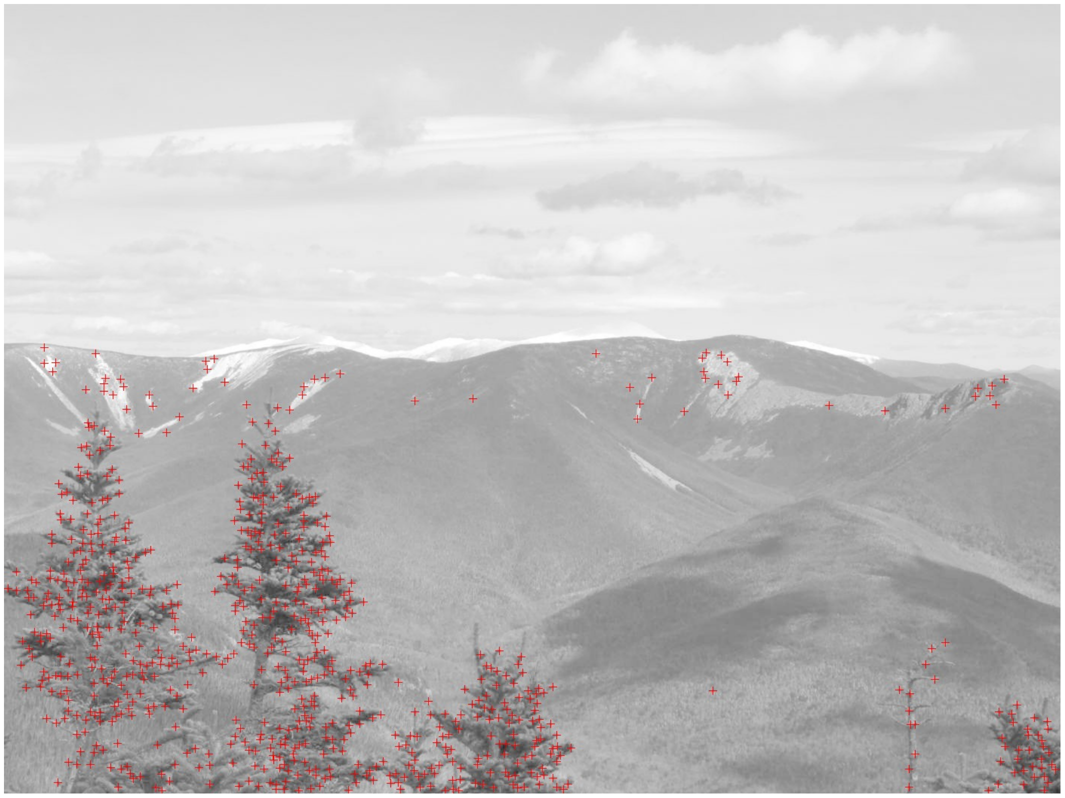
```
bin/ransac_st --inname1=bonds1.png --inname2=bonds2_st.png --thr=10
bin/ransac_rst --inname1=bonds1.png --inname2=bonds2_rst.png --thr=10
```

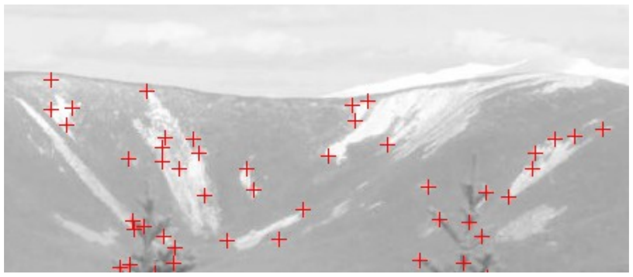
The FAST detector yields counts between 400 and 750 corners after suppression. The translation does not affect the total, as it should not, but the scaling and rotation do. The detail images show that the RST corners are a subset of the base; with one or two exceptions new corners have not been created.

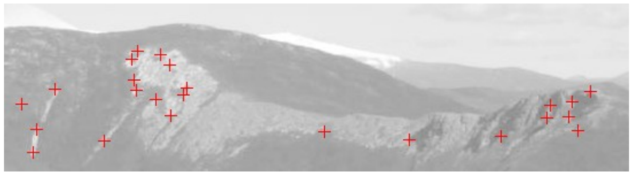
We ran both RANSAC programs for 20 trials on each test image against the base, with a threshold of 10 and all other parameters default. The tables present the average and one-sigma deviation for the transform, as well as the number of tries that successfully found a mapping (good or bad), the average number of re-seedings per try (including failures), and the time in milliseconds per try. The programs have been compiled with `--fast`.

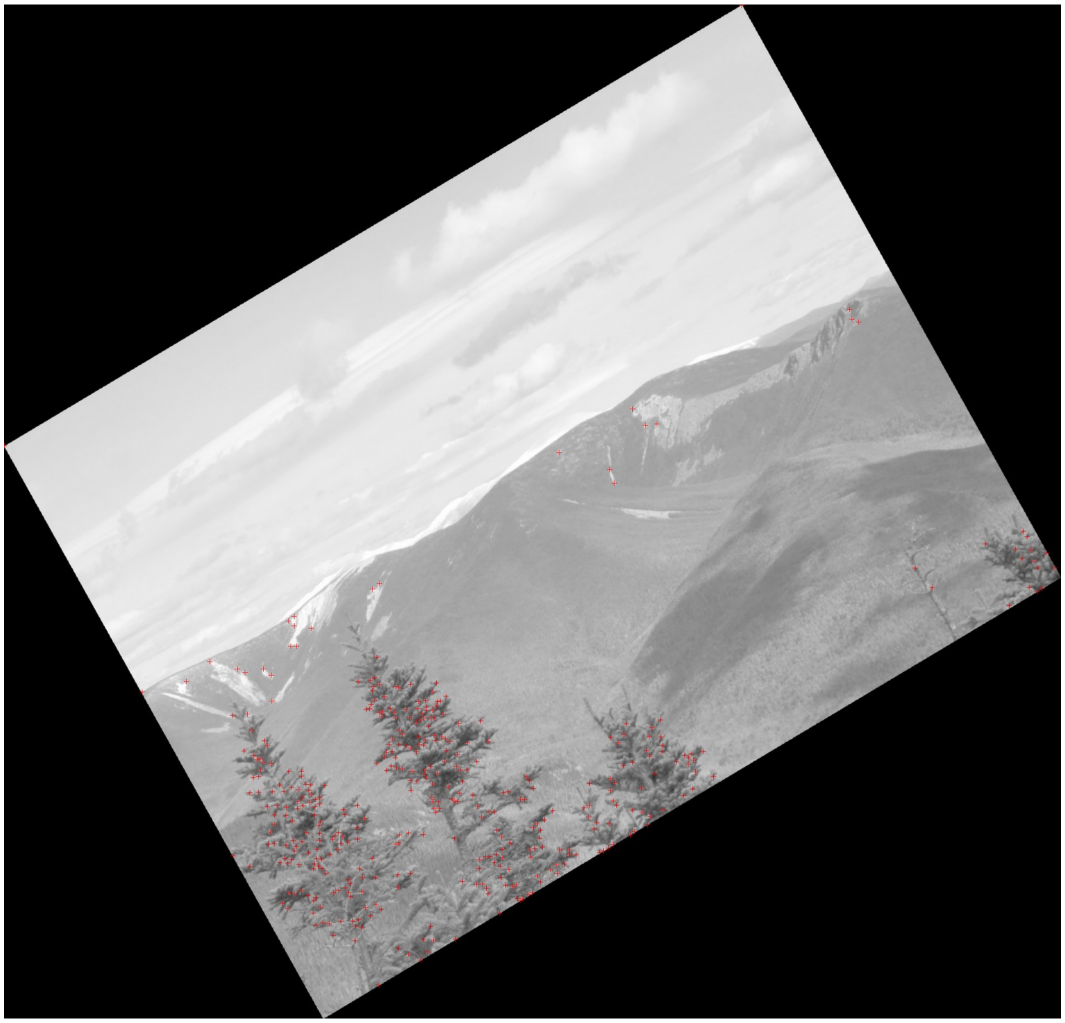
ransac_st --thr=10 --ntry=50 (default) differences to actual in red												
	sx			dx			sy			dy		
S	1.25	±0.0	0	-0.1	±0.0	-0.1	1.30	±0.0	0	0	±0.0	0
T	1.00	±0.0	0	150.0	±0.0	0	1.00	±0.0	0	300.0	±0.0	0
ST	1.25	±0.0	0	149.9	±0.0	-0.1	1.30	±0.0	0	300.0	±0.0	0
RT	-- no successes in any trial --											
RST	-- no successes in any trial --											

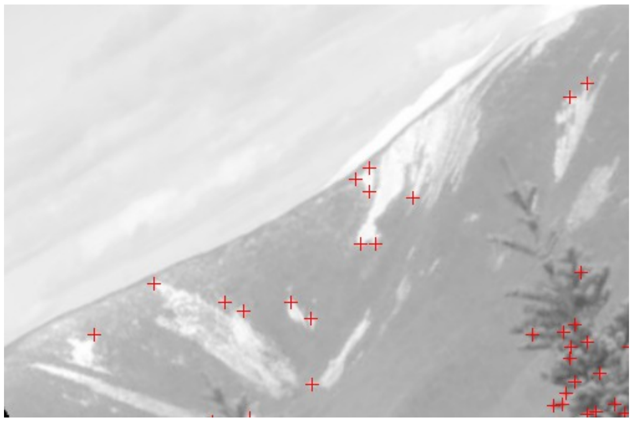
ransac_st --thr=10 --ntry=50 (default)			
	success [%]	re-seeds	time/try [ms]
S	49 ±8	1087 ±92	24 ± 2
T	99 ±0	212 ±36	7 ± 1
ST	53 ±6	1063 ±60	24 ± 2
RT	-- no successes --		
RST	-- no successes --		

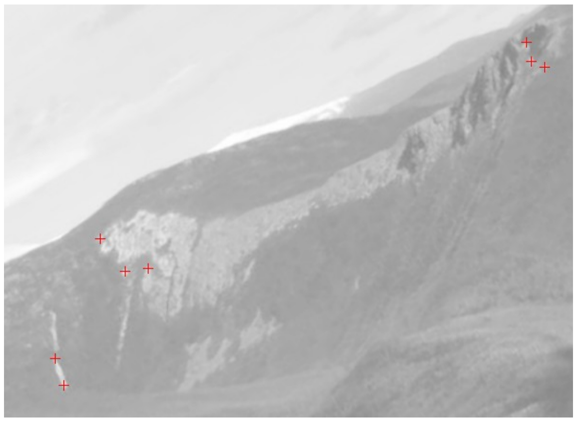

Base image bonds1.png, FAST corners with thr=10


Detail base image left (slides)


Detail base image right (Bondcliff)


RST image bonds2_rst.png, FAST corners with thr=10, translation cropped


Detail RST image left (slides)


Detail RST image right (Bondcliff)

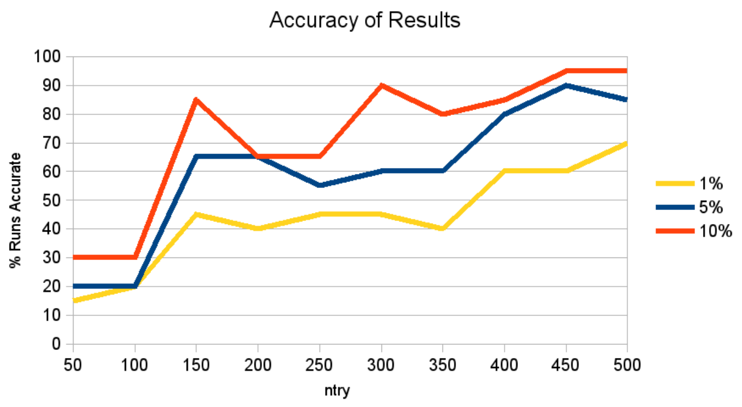
ransac_st performs very well with the unrotated test cases, producing accurate estimates of the scaling and translation. It has more trouble with scaling than shifting, as the success rate drops to 50% and the number of trial seeds needed for each match grows strongly, which is reflected in the timing. All mappings are accurate. As might be expected it had no luck with the rotated images.

ransac_rst --thr=10 --ntry=250 (default) differences to actual in red															
	sx			dx			sy			dy			theta (deg)		
S	0.88	±0.4	-0.38	214	±352	+214	0.87	±0.4	-0.42	435	±454	+435	-14.3	±53	-14.3
T	1.00	±0.0	0	152	±22	+2	1.00	±0.1	0	306	±50	+6	0.0	±2.0	0
ST	1.03	±0.3	-0.25	195	±225	+45	1.04	±0.4	-0.26	652	±528	+352	7.7	±52	+7.7
RT	1.00	±0.0	0	0	±0	0	1.00	±0.0	0	700	±0.0	0	30.0	±0.0	0
RST	1.25	±0.0	0	146	±146	-4	1.30	±0.0	0	1215	±22	+5	30.3	±1.2	+0.3

ransac_rst --thr=10 --ntry=250 (default)			
	success [%]	re-seeds	time/try [ms]
S	23 ±2	1322 ±22	39 ± 1
T	58 ±3	944 ±42	38 ± 1
ST	23 ±2	1318 ±22	39 ± 1
RT	76 ±2	792 ±39	27 ± 1
RST	45 ±3	1121 ±32	23 ± 1

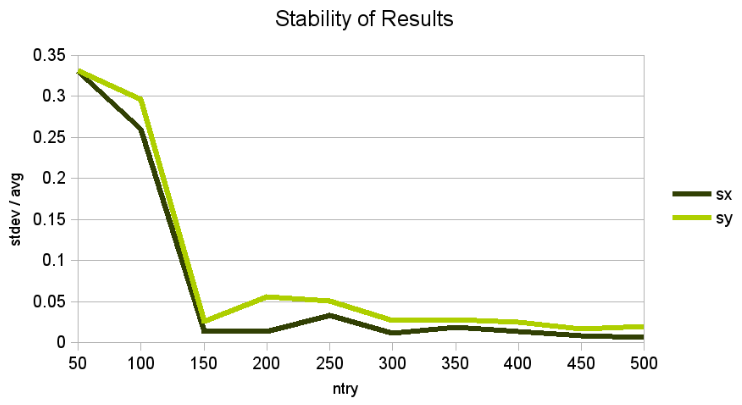
ransac_rst performs best on the rotated and translated images, but very badly with scaling without rotation. The S and ST tests have large errors, and the high standard deviations indicate the measurements are not stable. The number of successful tries is small, about half the rate of the good results, and this is reflected in the large number of seed trials for each and the corresponding run time. Still, the results on the rotated images, and the RST test in particular are good.

Does the performance of the RST version get better with more tries? Yes. We made 20 runs with `ntry` between 50 and 500 in steps of 50. The chart shows the percentage of the 20 runs that were accurate to within 1%, 5%, or 10% of the actual values. All five parameters had to meet the accuracy threshold for the run to count.



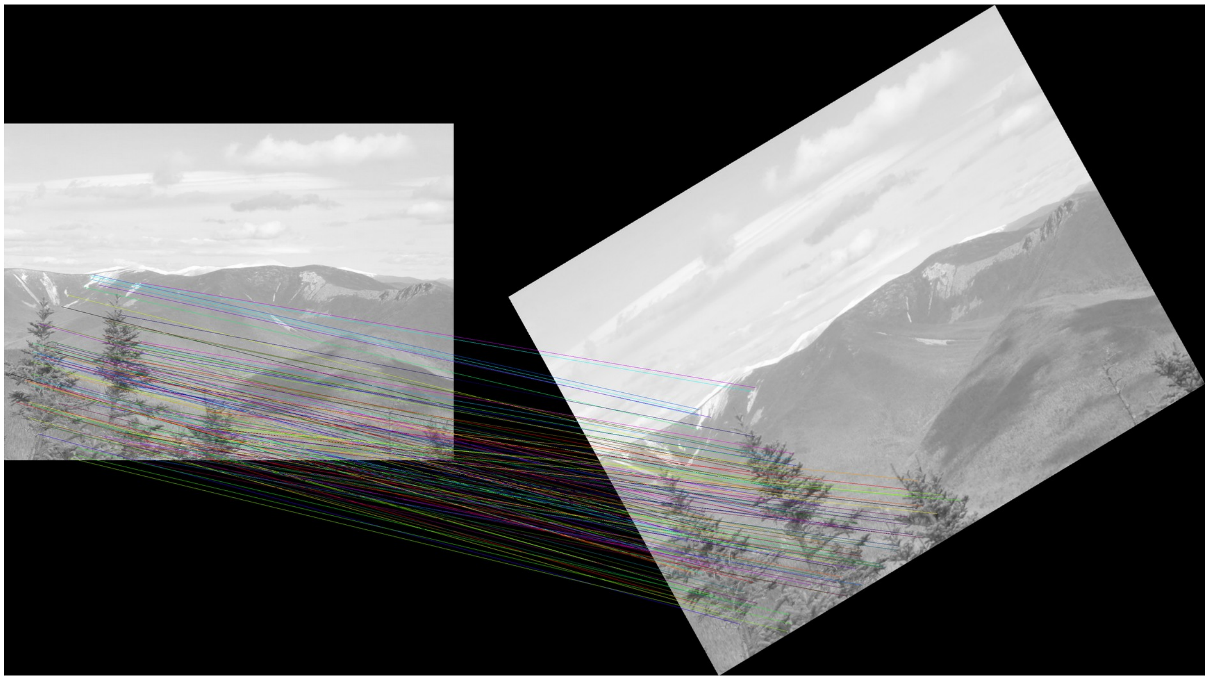
Accuracy of RANSAC RST results

For 450 and 500 tries, 19 of the 20 runs had sx within 1.25 ± 0.125, sy within 1.30 ± 0.13, dx within 150 ± 15, dy within 1210 ± 121, and theta within 30.0 ± 3.0 degrees. 14 of 20 runs had an sx of 1.25 ± 0.0125, sy 1.30 ± 0.013, dx 150 ± 1.5, dy 1210 ± 12.1, and theta 30.0 ± 0.3. Plotting the ratio of the standard deviation to the average for each parameter gives an indication of the stability of the measurement. There is a sharp decline at 150 tries for the scaling factors, and then a slow settlement. This means there is a broad maximum in the parameter space that is likely to be found given enough tries (150), but that the true alignment space is much smaller, as the accuracy results imply.

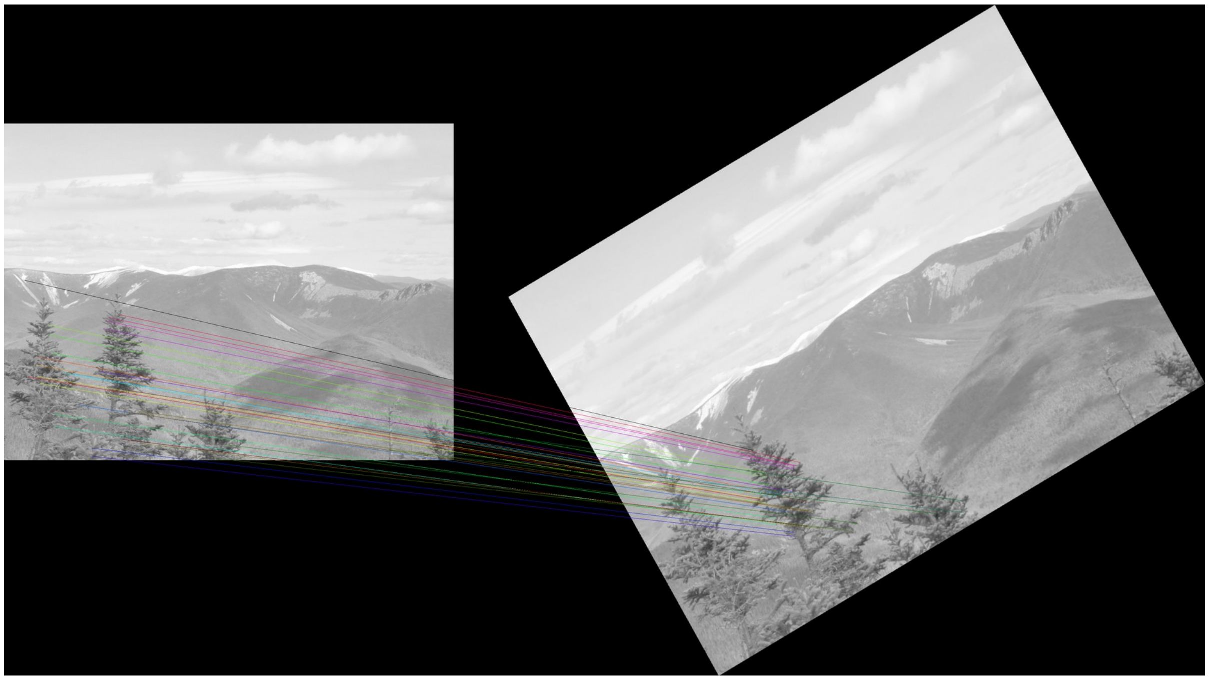


Stability of RANSAC RST results

We can see what is happening by placing the input images side-by-side and drawing lines between matching corners. (This has only been implemented in the RST version. The image is generated if `--outname` is given.) When the matching is poor the lines can clump, with large scaling factors that compress one image into a dense area of corners in another. You can see this in the bad image, where everything maps into the middle tree, including the corner in the slide at the middle of the left edge and the tree in the lower left corner. There doesn't seem to be a clear decision rule for separating bad and good results: the parameters of a bad match are still in the ranges we allow, and other tests have shown that adding requirements such as needing to have a wide distribution of matching corners over an image results in other pathologically bad matches.



Good RANSAC results, with theta=30.1, sx=1.25, sy=1.30, dx=150, dy=1211. Original image bonds1.png on left, RST image bonds2_rst.png on right, lines pair matched corners,



Bad RANSAC result, with theta=7.3, sx=0.48, sy=0.52, dx=850, dy=1439. Original image bonds1.png on left, RST image bonds2_rst.png on right, lines pair matched corners.
.

Wrap-Up

The RANSAC example offered nothing in the way of new language features. Instead it is a summary of what we have learned, using the C interface to read and write PNG images, the color conversion library to convert them to greyscale, and the FAST corner detector with its custom iterator and chunkarray generic class to store a list of corner features. We added a generic class kdtree to efficiently search the plane for nearest corners. We have used arrays, ranges, and slices extensively to process the images and lists. The program contains both task-based parallelism in the kdtree (cobegin) and data parallelism (forall) in the RANSAC analysis.

Whether we put the RANSAC algorithm in our toolkit is still uncertain. Its performance on real images, of natural scenes taken at different times by different people with different cameras, teases us with occasionally good results, but not consistently. Scaling in particular gives it difficulty. We have tried another approach with different features than corners, and the behavior is similar to what we see here: a rough alignment is reached, but there are many local maxima in the matching figure of merit (here the number of matching corners) that generate more matching pairs than the actual and give false results. The evaluation of the algorithm is still going on, but it leads us outside the scope of this tutorial.

Files

The programs and images for this chapter are here. A zip file is here.

A PDF copy of this text is here.

PRACTICAL MATTERS

Debugging

While developing the programs for this tutorial, we've learned several tricks for debugging that might come in handy: how to run a Chapel program in a debugger, how to look at the intermediate C code to find compiler bugs, how to improve parallel performance (with a discussion of Chapel's tasking model), and submitting test cases for bugs and feature requests.

Exceptions and Segfaults

If a program crashes with no error message printed to the command line, then you'll need a debugger to find the cause. The compiler has command line options to help do this.

When compiling, use the `-g` and `--savec` flags. `-g` adds debugging information (similar to a C compiler). `--savec` takes a directory (which will be created if it doesn't exist) and preserves the intermediate C files for line references.

```
chpl -g --savec=/tmp/chpl -o prog prog.chpl
```

Note that this will increase the compile time.

Then when running the program, use the `--gdb` flag to start the program in GDB.

```
./prog --gdb [other arguments]
```

Within GDB, just type run. The program will halt at the error and print the stack trace with the line of the error. You can inspect the program's state normally within the debugger. The compiler may have changed the names of your symbols, as we'll see next when we look at the generated code.

Generated Code

While debugging the ransac_rst program, we ran into a runtime error complaining about memory being freed twice. We found the problem by looking at the C code that Chapel generated from the program.

You'll see in the `pick_seeds()` procedure that we build a list of potential matches three times, for all but the first seed. Before picking one, we check that there is at least one possibility. If not, we abandon the attempt.

```
nmatch = 0;
for i in rng2 {
  if are_corners_similar(corners1(ind11), corners2(i)) {
    nmatch += 1;
    matches(nmatch) = i;
  }
}
if (0 == nmatch) then return false;
```

It was in the last line that the program was crashing with the message

```
ransac_rst.chpl:192: error: halt reached - array reference count is negative!
```

where line 192 points to the definition of the `matches` array.

The compiler has two options of interest. As we said above, the `--savec` option takes a directory for saving the generated C code. You can also add `--no-munge-user-idents`. The compiler will modify your symbol names to prevent collisions with internal symbols by appending _chpl. This option will (mostly) prevent that from happening. You'll find though that even with munging the symbols are still readable.

That (0 == nmatch) test compiled to:

```
call_tmp_chpl54 = (INT64(0) == nmatch_chpl);
if (call_tmp_chpl54) {
  ret_chpl = false;
  chpl__autoDestroy5(matches_chpl, INT64(192), "ransac_rst.chpl");
  chpl__autoDestroy2(call_tmp_chpl6, INT64(194), "ransac_rst.chpl");
  chpl__autoDestroy2(call_tmp_chpl7, INT64(195), "ransac_rst.chpl");
  goto _end_pick_seeds_chpl;
}
/* much code deleted */
_end_pick_seeds_chpl:;
chpl___ASSIGN_17(try_chpl, &_formal_tmp_try_chpl);
chpl__autoDestroy2(call_tmp_chpl7, INT64(195), "ransac_rst.chpl");
chpl__autoDestroy2(call_tmp_chpl6, INT64(194), "ransac_rst.chpl");
chpl__autoDestroy5(matches_chpl, INT64(192), "ransac_rst.chpl");
return ret_chpl;
```

The pattern we see the compiler using is to jump to a label at the end of the procedure to clean-up local allocations (which is the style we talked about in the Image Interface chapter). The problem is obvious: there are two calls to chpl__autoDestroy5(matches_chpl, ...), which raises the error. Even with the symbol mangling it was easy to find the problem code by searching for the array name given in the error message. The work-around is to use nested `if`'s instead of returning so that the clean-up only occurs at the end of the procedure. You'll find this in the current version of ransac_rst.chpl.

```
if (0 != nmatch) {
  /* determine third seed */
  if (0 != nmatch) {
    /* determine fourth seed */
    if (0 != nmatch) {
      return true;
    }
  }
}
return false;
```

Tasks

A system monitor that can show the load on each (virtual) core is a good tool for checking if your program is being run in parallel. While running ransac_rst, we noticed that the corner alignment was only running on two cores, despite a `forall` in the main loop.

You'll see a difference in behavior depending on which threading package you've compiled into Chapel. The quickstart version uses pthreads, the full version qthread (plus there are a couple other third-party options that we won't discuss). pthreads is a kernel-level library: threads are created and managed by the system, which is relatively expensive. In the qthread library the threads are managed in user space. Technically Chapel doesn't see threads, it works with an abstraction called a task that maps down to threads by each library. A task is created by a `begin`, `cobegin`, or `coforall`. There is some control over the resources allocated for threads - their number and stack size, for example - but in practice you should never count on being able to affect how threads execute, or how tasks are mapped to them.

In the pthreads approach, there is a single task pool. Threads are created as needed for the tasks (up to a configurable limit), and each thread runs its task to completion. Because threads involve system calls to create, they are not destroyed when their work is done. Instead, they look for another task to pick up. Thus, tasks will start quickly if there is an idle thread available, or less so if one must be created. If a task blocks (on a synchronized variable, for example) then the system scheduler idles its thread. Each thread guards against stack overflow by adding a guard page at the end of its stack that will raise an error if accessed.

For the qthreads library, the task pool is split into different queues. The library creates two kinds of threads, workers for task execution and shepherds for distributing work. When a task blocks the shepherds can re-assign a worker to another task. This is done in user space. There is one worker per core, and often multiple workers per task queue. qthreads also uses a guard page at the end of the stack to guard against overflows.

In practice the qthreads library performs better than the pthreads. There may be a higher cost at start-up because it creates more worker threads initially, but the pthread-by-pthread startup based on task demand may lead to a slower overall program. If the system creates many pthreads, then they will hang around waiting for new tasks, and this can lead to a deceptively high system usage as they poll for new work. You can see this if your system monitor distinguishes between the time spent in system calls (pthreads overhead) or userland (actual work). We have seen higher loads spread evenly across the cores with pthreads than qthreads so that it looks like more work is being done, but the program still runs slower.

The Chapel development team recommends splitting work coarsely if possible, with big loops, to minimize the thread overhead. They also suggest not using a `coforall` with many more iterations than the number of cores, because this just creates tasks that will have to wait for resources to free up. A `forall` is in general a better choice. Make sure the hardware configuration is in line with the runtime environment - don't create more locales than compute resources, for example. Until you need to fine tune a large application, there's no need to play around with the many environment variables and command line options that the libraries and Chapel support to tweak the number of threads or stack size. Details about these options are scattered through many files in CHPL_HOME/doc, like executing.rst and tasks.rst.

With the help of Greg Titus at Cray we were able to track down the problem and find a work-around. The top level matching loop in the program looked like:

```
proc align_corners(const corners1 : [] corner, const corners2 : [] corner,
                   out bestmap : mapinfo,
                   map1to2  : [] int, map2to1 : [] int) {
  var tries : [1..ntry] tryinfo;
  var rand :  new RandomStream(real);
  const minlen = min(corners1.domain.dim(1).last, corners2.domain.dim(1).last);
  const mincnt = nearbyint(matchfrac * minlen) : int;
  var tree = new kdtree(corners2.domain.size, int, 2);

  /* Build the kd-tree for quick lookup of mapped corners during match. */
  for i in corners2.domain {
    tree.add_node(corners2(i).center, i);
  }
  tree.assemble_tree();

  forall i in 1..ntry {
    do {
      if pick_seeds(corners1, corners2, rand, tries(i)) {
        var mapped1 : [corners1.domain] corner;
        map_corners(corners1, tries(i).map, mapped1);
        const matchcnt = count_matches(mapped1, corners2, tree);
        if (mincnt <= matchcnt) {
          tries(i).nmap = matchcnt;
          break;
        }
      }
      tries(i).nfail += 1;
    } while (tries(i).nfail < nfail);
  }

  /* Regenerate the best mapping and refine it with a linear regression.
     This is the final result. */
  const besttry = select_besttry(tries);
  var mapped1 : [corners1.domain] corner;
  map_corners(corners1, tries(besttry).map, mapped1);
  match_corners(mapped1, corners2, tree, map1to2, map2to1);
  refine_mapping(corners1, corners2, map1to2, map2to1, bestmap);

  delete rand;
  delete tree;
}
```

(This code compiled in Chapel 1.12. Later versions require some changes, for example for the random number generator. We haven't checked if these fixes are still necessary.)

We started by looking at a histogram of the timing of each trial, which is measured and stored in the tryinfo record. We noticed that although most trials took on the order of seconds to run, there were two that waited hundreds of seconds (and two that took tens of milliseconds). We do not expect orders of magnitude difference in the times.

We converted the loop to a work queue, which has a fixed number of worker threads that start a trial when they're free. The `coforall` guarantees the number of workers. The `nextIter` variable is incremented as each trial starts, making sure we cover the number specified.

```
    var nextIter : atomic int;
    nextIter.poke(1);
    coforall worker in 1..here.maxTaskPar {
      var i = nextIter.fetchAdd(i);
      while (i <= ntry) {
        /* perform trial */
        i = nextIter.fetchAdd(i);
      }
    }
```

`here` is a constant that Chapel creates. It is the locale in which the current task (the function call in this case) is running. `maxTaskPar` is a member of the locale class with the maximum parallelism available at the site.

This too only ran on two cores under qthreads, however. With the pthreads library it would use all four cores, albeit with a higher system load. This implies the problem lies with the threading library, which is not something we could debug. The timing still showed a large spread in values. With the work queue we could count how many trials each task in the `coforall` executed. Three were balanced, but one ran only one pass. That thread was blocking until the others finished all the trials. The next step was to isolate what was causing the block. We scattered timing statements throughout the loop and found the `pick_seeds()` call was where it would freeze. Checking the timing of `pick_seeds()` almost line by line, we found that the call to the random number generator was what was blocking.

We made one change to `align_corners()`, to move the random number generator into the loop. In principle this shouldn't be necessary; it's supposed to be parallel safe. But something in its implementation, perhaps an interaction with the synchronization variable the generator uses as a lock and the thread scheduler, causes the qthreads library to block. Because pthreads are scheduled by the kernel, we would expect their behavior to be different. The change was made in the process of trying different code paths, not deliberately. The loop you'll find in the program now is

```
    forall 1..ntry {
      var rand = new RandomStream();
    }
```

with everything else in the procedure unchanged. This costs an instantiation of the random number generator per trial, but avoids contention for its lock, and this restores the correct parallel performance. As with all the problems we've found, we submitted the problem to the development team and continue to use the work-around. (We haven't checked if the behavior has changed with the latest release.)

Bug Submission and Test Cases

Chapel is hosted on Github and uses its issue tracker for bugs. The New Issue button opens a ticket. You should provide a summary of the problem, how to reproduce it, and your Chapel setup and environment. See CHPL_HOME/doc/rst/usingchapel/privatebugs.rst for the guideline. You can also prepare a test case that becomes part of the nightly test run.

Procedurally this is a bit involved. Because the test case will become part of the release, you'll be asked to sign a Contributor's Agreement. The Chapel repository is found on Github at

https://github.com/chapel-lang/chapel

You'll notice it has several directories that do not appear in the release version. The agreement is found in one, at doc/developer/contributerAgreements. There are two versions, one for individuals and one for companies.

Once the agreement has been signed and sent to the development team, you'll need a Github account to be able to check in code. Make a clone of the master branch on your local machine. You'll find complete instructions in CHPL_HOME/developer/bestPractices/ContributorInfo.rst

Chapel uses a test system that automatically runs code samples it finds. In the development version, you'll find the test cases in CHPL_HOME/test. Many of these are copied into CHPL_HOME/examples for the release version. Not only does the system verify the correct behavior of the language and bug fixes, but it can also track the performance of programs and is used to track feature requests. The main test script scans the top directory recursively looking for files with a .chpl suffix. A test contains at least two files: a program to run and the output it should generate.

<file>.chpl - test program

<file>.good - expected output

The test system compiles the program, runs it, and compares the actual output to the expected. Mismatches count as a failure. If your program requires options to the compiler, anything other than a `chpl -o <tmpfile> <file>` command, then you can put them in specification files with the options on one line. If your program requires options to execute, they go in another file.

<file>.compopts - command-line options to compiler

<file>.execopts - command-line options to run program

The system also supports a directory-wide, instead of per-program, specification. For these two files the global versions are COMPOPTS and EXECOPTS. We prefer using the per-test version.

To create a test case, you'll need a place to put it in under the test directory. The sub-directory CHPL_HOME/test/users/<your name> seems to be the preferred location for your sample program and good output. Then run the program `start_test` (found in CHPL_HOME/util) on your directory. You should run `start_test` both with and without the `-performance` option; the nightly tests run with both. Once you've verified the test case is correct, sync your branch with Github and submit a pull request to the development team. They will verify the change and test case, ask you for edits if any are needed, and will finally do the merge with the master branch.

The development team also uses this system for feature requests. In addition to the two files above, you'll need to submit a description of the request. The first line of <file>.future must contain the type of change (feature, bug, edit to an error message) and a succinct summary. The rest of the file can contain details about the proposal. If the change raises a compiler or runtime error, then you can include another file to capture that. This will catch other changes to Chapel that break the incorrect behavior we currently have.

<file>.future - explanation of feature request

<file>.bad - output generated on a failing test

Note that the output in <file>.bad may contain information specific to a release: line numbers inside the compiler, for example. If you add a script <file>.prediff, it will be run on the output before comparisons are made. This might include sed commands to remove variable line. Make sure the file is flagged executable.

<file>.prediff - pre-processing of output before comparison to .good/.bad

Feature requests are not included in the daily regression testing. The team monitors them separately.

The test system can also be used to monitor the performance of programs. The system will examine the program's output for keywords or strings. A value after the keyword/string will be extracted from the output and appended to a file that tracks the value over time. There is support for graphically viewing the progress in a browser, as described in the documentation of the test system in CHPL_HOME/doc/developer/bestPractices/TestSystem.txt (on Github, not in the official release).

<file>.perfkeys - keyword(s) in output that gives a performance metric

There are also variations of the compiler and execution options files, since they may differ from the functional tests. Note that `--fast --static` are automatically added before the options you define.

<file>.perfcompopts - compiler command-line options during performance tests

<file>.perfexecopts - execution command-line options during performance tests

Let's see the test system in action. We've added the necessary files to the ransac directory to verify the output of ransac_rst. This required a change in the code, adding the config param `--fixrng` to use a known seed for the random number generator. This removes the variability when running the program: we should see the same output, with only the average time per try changing.

In ransac_rst.chpl:

```
forall i in 1..ntry {
  var rand : RandomStream();
  if (fixrng) {
    rand = makeRandomStream((2*i) + 1);    /* must be an odd seed */
  } else {
    rand = makeRandomStream();
  }
  /* rest of loop unchanged */
}
```

Our compiler options are

ransac_rst.compopts:

```
--fast -sfixrng=true ip_color_v3.chpl ip_corner.chpl kdtree.chpl img_png_v3.h
build/img_png_v3.o -lpng
```

ransac_rst.perfcompopts:

```
-sfixrng=true ip_color_v3.chpl ip_corner.chpl kdtree.chpl img_png_v3.h
build/img_png_v3.o -lpng --dynamic
```

You'll notice that most of the line contains the files we put on the normal compilation command. We add `--fast` so that the normal test bench runs quickly. `--dynamic` is needed to cancel the `--static` option that's added automatically and which on our system does not link.

For the execution options we list the input files and threshold.

ransac_rst.execopts and ransac_rst.perfexecopts:

```
--inname1=bonds1.png --inname2=bonds2_rst.png --thr=10
```

We will want to remove the variable timing result in the output for the functional test bench. A sed command to delete the line is enough.

ransac_rst.prediff:

```
#!/bin/sh
testname=$1
outfile=$2
sed '/avg time per try/d' $outfile > $outfile.2
mv $outfile.2 $outfile
```

This is also part of the line with the timing value we want to track. The full line is

ransac_rst.perfkeys:

```
avg time per try [ms]
```

Now in the ransac directory you can run the test bench.

```
cd ransac
start_test ransac_rst.chpl
```

You'll see the program setting up the test environment, compiling the program, and running it. It prints a summary with the pass at the end. A copy of the output is put in a log file. You can see which is used if you type

```
start_test -h
```

and look for the -logfile line. The default location is in the Chapel distribution directory. If you're working elsewhere, you'll want to change the location.

The performance test bench wants to write its results into the environment variable CHPL_TEST_PERF_DIR, which defaults to CHPL_HOME/test/perfdata/<machine>. Change this if you're not working directly in the distribution. You'll also need to rename the prediff file so that system does not run it and delete the timing data. (The development team recommends a config constant to turn off the variable output for functional tests, because the system can't skip the prediff script.)

```
mkdir /tmp/perfdat
export CHPL_TEST_PERF_DIR=/tmp/perfdat
mv ransac_rst.prediff ransac_rst.prediff.func
start_test -performance ransac_rst.chpl
```

Once this runs, you'll find a file /tmp/perfdat/ransac_rst.dat with the timing value. If you run the program repeatedly (and start_test has an option `-num-trials` to make multiple runs with only one compilation) you'll see them added to the end.

Resource Usage

There are several tools available to help track resource usage by programs.

For debugging memory usage, Chapel programs will run under valgrind. There are also command line options when you run a program. `--memStats` prints a table with the amounts that have been allocated and freed, `--memLeaksByType` the source of memory that's not been freed when the program ends, and `--memLeaks` a full list of all unfreed memory, including where in the program it was allocated.

If you compiled Chapel with the hwloc package, it contains several programs to discover the system's layout. They're found in CHPL_HOME/third-party/hwloc/install/<arch>/bin. `lstopo` will show the system's topology graphically. `hwloc-info` and `hwloc-ls` will print the information on the console.

The October 2015 Chapel 1.12 release contains a new monitoring tool called `chplvis`, found in CHPL_HOME/tools/chplvis. To use it you add two calls to start and stop the service, and can add tags to monitor smaller sections of your program.

```
use VisualDebug;
startVdebug(dirname);
/* do work */
tagVdebug(tagname);
/* do more work */
stopVdebug();
```

As the program runs it will write information about the tasks that are created and when they start and finish to a file in the given directory. The chplvis program graphically displays the resource usage on each node, such as CPU usage, number of tasks, and inter-locale communication. The graphical display doesn't seem very useful for our setup with its single CPU, but the raw data the VisualDebug module generates does look interesting. For the ransac programs it shows that the kdtree generates more tasks that there are computing units, meaning that many will be idled, while the forall loops used during alignment are tailored to the number of cores.

LOOSE ENDS

There are a few parts of Chapel that we haven't touched on. Here's a quick summary until we're able to provide a more detailed description:

– Locales are separate processing units with local memory. The runtime identifies the hardware configuration and provides it through an global array `Locales` that describes each unit, such as the number of cores it has, the size of its memory, how many tasks it can run, and a unique identifier. The `on` statement executes a code block on a specific locale.

– Domain maps control the binding of index ranges to locales. They can be created as standalone variables with the `dmap` type and bound to a domain by placing `dmapped <map>` after the domain declaration.

```
var dommap : dmap(Block(rank=2)) = new dmap(new Block((1..4,1..4));
var dom : domain(rank=2) dmapped dommap;
```

creates a 4x4 grid over which the domain indices will be divided in block fashion. Alternatively you can specify the map directly with the domain

```
var dom : domain(rank=2) dmapped Block((1..4, 1..4));
```

– Chapel provides four built-in domain maps called distributions; they are found in CHPL_HOME/modules/dists.

1. Block - each range is broken into several pieces according to the layout of the locales and all combinations of the pieces are run on different locales
2. Cyclic - range indices are taken modulo the number of locales along that range (locales may be arranged over multi-dimensional domains)
3. BlockCyclic - each range is broken into a block with a specified size and the blocks are assigned modulo the number of locales along that range
4. Replicated - each locale receives a copy of the domain and works independently, without communicating changes to the data to other locales

Chapel also has a well-defined interface called the DSI for creating new domain maps. It is explained in CHPL_HOME/doc/rst/technotes/dsi.rst.

– Associative domains and arrays use discrete values of any primitive type or class as indices, serving as a hash table or dictionary. The set of values can be changed with the `add()` or `remove()` methods, or using the += and -= operators; values are also added if used as the index to an array during assignment. Declare an associative domain by specifying the type. An initial set of values can be given between braces.

```
var counting : domain(string) = { "one", "two", "three", "four" };
var testarr : [counting] int;
testarr("four") = 4;
testarr("five") = 5;     /* or counting.add("five");  counting += "five" */
```

– Sparse subdomains are a list of values belonging to another domain. Indices are added like an associative domain. An array backed by a sparse subdomain has a default value `arrayname.IRV` for any index not provided in the list. You can only set values for indices that have been specified; they will not be automatically added to the sparse index list. Declare it by putting sparse before the subdomain type.

```
dom : domain(rank(2)) = { 1..ncol, 1..nrow }
var sparsedom : sparse subdomain(dom);
var sparseimg : [sparsedom] uint(8);
sparsedom += (1, 1);
sparseimg(1, 1) = 5;
sparseimg.IRV = 128;     /* all pixels but (1,1) have this value now */
```

– Set operations such as union, intersection, and membership tests are being added atop associative domains.

– Classes and records support an object hierarchy, where you can declare a superclass with the subclass:

```
class subclass : superclass { .. }
```

Subclasses inherit members and methods from their parents.

– IO is done to a file through a channel. These channels implement a generic `Reader` or `Writer` interface. The methods `readThis()`, `writeThis()`, and `readWriteThis()` can be overloaded to customize IO operations for a class or record. The IO standard module defines the operations supported.

– You can embed C code directly in a Chapel source file with an `extern` block.

```
extern {
  rgbimage *rgb = NULL;
  if (alloc_rgbimage(&rgb, 10, 10) < 0) {
    /* handle error */
  } else {
    /* work on image in C */
  }
  free_rgbimage(&rgb);
}
```

This requires enabling LLVM support in the compiler, as it is used to parse the C.

– CHPL_HOME/modules/standard and CHPL_HOME/modules/packages have many more libraries, including file system support (FileSystem, Path), a libcurl interface for file transfers (Curl), system calls (Sys, Time, Spawn), math and number functions (Math, BitOps, FFTW (Fast Fourier Transform), LAPACK, BLAS, GMP (GNU Multiprecision Library), Norm, UtilMath, LinearAlgebra), an interface to Hadoop systems (HDFS, HDFSIterator), tracking of network communication (CommDiagnostics, startInitCommDiags), regular expressions (Regexp), cryptography (Crypto, Random), and iterators that can steal work from others if they have nothing to do (AdvancedIters).

– CHPL_HOME/doc/rst/technotes describe language features that are still in progress, including partial compilation of source files into libraries, a REPL, more locale architectures including NUMA, LLVM support, and writing formatted output to strings.

As we have said, the doc, doc/rst/technotes, modules/standards, modules/packages, and examples/primers directories in CHPL_HOME contain a diverse collection of design and usage notes.

SUM-UP

This brings us to the end of our study of Chapel. It's time to look back and answer the question posed at the start: Is this a language we can use?

We've seen Chapel in action in pixel-level computations (color conversion and FAST corner detector), at the matrix level (Gabor filter convolution), and at a more general level (k-means clustering and RANSAC feature matching). We've used most, but not all, of the language's features, which we can summarize in a list:

Data Structures

- arrays (data), domains (indices), ranges (iteration)
- classes and records with embedded variables and procedures
- standard iterator interface any class or record can provide (`these()` method)
- tuples and enumerations

Parallel Programming

- data-level parallelism breaking up iterations (`forall`, `coforall`)
- task-level parallelism based on statements (`begin`, `cobegin`)
- synchronization across tasks (`atomic`, `sync`, and `single` variables)

Generic Programming

- classes and records programmable by type, fixed-value parameters
- function overloading and polymorphism, `where` clauses to specify type requirements
- queried arguments for types and details
- variadic argument lists

Language Fundamentals

- argument intents that control writeability and linkage to the caller
- standard control statements with keyword version for single statement and braced version for blocks
- primitive types `int`, `uint`, `real`, `imag`, `complex`, `bool`, where the bit size can be specified if needed

- programs grouped by modules, with an inherent module per file and access control (public/private)
- extended set of operators, including exponentiation and variable swap
- program constants that can be changed at runtime on the command line, or parameters set at compile time

Extensibility

- custom iterators
- custom domain maps

But what has it been like programming with the language?

Experience

Writing programs in Chapel has been pleasant. The language fits this problem domain well. Domains and slices are natural ways to work with images and clearly signal the intent behind the code. Having domain changes re-allocate arrays is a strong encouragement to properly set them up. Parallelism is easy to add to programs, but not entirely worry-free. You still need to keep in mind what data is crossing task boundaries and if there might be races (the default intents prevent this but can be circumvented), and it's not clear how a piece of code will act on actual hardware – the runtime is a bit mysterious. Generic classes and queried arguments provide the flexibility for re-usable code. Argument intents make the role of each argument clear, and supporting multiple output arguments is nice. Interfacing to C is straightforward although a little verbose when needing to use the C types. The automatic creation and parsing of command-line constants means never having to program a parser again. Chapel is an expressive language and that is apparent while coding.

We have also found developing in it to be somewhat frustrating. The flexibility of the language and its default behavior is partly to blame, as is the compiler. Part of Chapel's expressiveness depends on its defaults, automatically generated constructors, accessors, and iterators, and overloaded operators that uniformly handle different data types. They reduce clutter in the code and are mostly sensible and intuitive, but we found ourselves constantly butting up against them while compiling and debugging. The overall impression is that the language is a bit squirrelly as we were constantly reminded of how much was happening behind the scenes, and thus how unsure we were of exactly what was going on. The flexibility can be confusing. For example, if out of habit you type array references with brackets instead of parentheses the program will compile, but the behavior might not be what you expect (at least for us, "strange results" cleared up after switching the brackets). Or, if you make a typo in the name of a constructor you will end up silently using the default, which is not what you want; this happened while developing the circumference class. The behavior of a program can also be confusing. We're left with the impression that passing arrays around in structures, particularly records, is not reliable. The data seems to corrupt eventually, and in the worst case the program fails. This happened a few times developing the RANSAC program, and its final shape, with only primitives stored in the mapinfo and tryinfo records and the map1to2 and map2to1 arrays being re-generated with the best try, rather than caching them as the program proceeded, is a result. The compiler did not help. Error messages are sometimes indirect; if you forget to tell a `forall` loop that a variable outside the loop is being used, the warning message complains about a bad lvalue. The compilation stops on the first error, and since the current version is a bit slow the 'fix typo - compile - fix typo - compile' cycle is tedious. Chapel also only seems to look at code that is being used. If you test compile a stand-alone library it can come out clean, with errors only appearing when you start calling the library functions from the main program. The compiler can generate error messages rivaling Clojure stack traces if it can't resolve an overloaded function, "helpfully" listing a hundred valid type signatures.

Perhaps this is part of the learning process and we find ourselves in something like an uncanny valley of confidence where we're starting to assume we know the language while pushing on it and exposing the gaps in our understanding. We don't yet have a clear idea of what is happening, and there's little feedback about missed opportunities for optimization or warnings about inefficient code (say, flagging array accesses that may be out of bounds when the compiler can't prove they are within the array's domain). It's a bit like learning the CUDA model for memory and execution without the profiling tools to show how you can improve your performance: there's a lot of stumbling about and trial-and-error, and any conclusions drawn about best coding practices may be due as much to luck and circumstance as to valid reasons.

Performance

You might have noticed that for a High-Performance Computing language we haven't talked much about how fast Chapel programs run, other than to compare different implementations against each other. There's a few reasons for this. Most importantly the team has been focused on the language definition and only recently started working on optimizing the code the compiler produces. Their presentations from the end of 2014 acknowledge that performance is poor but is getting better. We shouldn't be surprised if our runs are slow. Another reason is that we can't compare the programs we've developed here directly with our internal versions, either because we've changed the approach or more often the data type. As an example we tend to work with integer images, and scale the greyscale plane and the Gabor kernel filter before running the convolution; our implementation here runs in floating point. There are also things we can't explain, such as where to best use `forall` (top level? everywhere? spot placement?). Finally, there is little feedback about how to optimize programs. It's a bit of a black box. Our impression, though, from these programs and others in the language shootout benchmarks is that Chapel is slow compared to C, and some constructs, namely reductions, are borderline unusable.

Maybe a story is in order. The k-means clustering was our first thought at writing a parallel program in Chapel because we had already gone through the exercise in C. Our serial version had run too slowly, so the first change we made was to split the image into equal parts, one per thread, converting the next-pass clusters into per-thread sub-totals and combining them at the end. In other words, it was the approach taken for kmeans_v1. The code edits took an hour and gave us about a 3.5X speed-up with four cores/threads. We then ported the algorithm over to CUDA, where the biggest changes needed were to fit into the GPU's memory model. The running time improved by a factor of 6. The Chapel kmeans programs, in comparison, were 15 times slower than the serial version, or 40% slower if compiled with `--fast`. Again, the differences between the C and Chapel versions affect the behavior of the clustering and the results are not directly comparable, but we have tried to have the same amount of work (same image so pixel count is the same, limiting the number of passes to be the same) so the rough result stands: the Chapel program is slower.

[For a better comparison, we've re-worked kmeans_v2.chpl to use only integers. This version converts each color plane to 8-bit, ie. `clrimage` -> `rgbimage`, and changes the types in the cluster record and procedure arguments from `real` to `int`. There are still differences in the algorithm that cause the Chapel version to take two or even three times as many iterations to converge, but the per-iteration time of the integer version is only 20-40% slower than the threaded C version. The per-iteration time of the float version is three or four times slower than the integer, which gives us the 40% slowdown to the serial C version.]

As Chapel evolves it is generating faster code. The Gabor kernels are running 2-4 X faster in 1.17 (April 2018) compared to 1.11 (April 2015); this is with --fast compilation. The kmeans programs are 20-30% faster, and both RANSAC versions 5 X. Without --fast the timing number are more stable. The Gabor kernel timing ranges from 20% slower to 40% faster, the kmeans even up to 20% faster, and RANSAC has a 2 X improvement.

Work In Progress

Chapel is still in active development. There are several language features that have not been implemented or will change. Some, such as strings and data hiding in classes/records, seem fundamental and pose the biggest risk for code re-work in the future. Others like the order of atomic operations over a distributed network hint at design issues that are outside our experience. (One reason for not trying locales and domain maps is that we haven't used that kind of hardware.) Many features are partially implemented and not yet well documented, or have bits and pieces of text describing them scattered about between the modules, docs, and examples directories. With each new release new find a few programs stop compiling. Fixes are small and make sense, but emphasize that the language and libraries are not yet stable. And of course there are bugs.

Any of these mean that there might be conclusions in these examples that are not, or should not, be valid, another example of trial-and-error learning. Two that come to mind are the decision not to use enumerations as constants or the recommendation not to use arrays-of-arrays as they cannot be used as arguments. Both apparently should work. This is not a reason we should reject using Chapel, only something to be kept in mind. The language will continue to evolve, and we must be prepared to move with it.

One piece we would like to see is CUDA support in the runtime, for both practical reasons and curiosity. Practical because it's the parallel environment most accessible to us and which has shown significant benefits, but we also wonder about how well the language maps to the GPU and handles the constraints of the hardware, especially the memory model.

Overall Conclusion

Early on, about a quarter of the way through this project, we made a placeholder note in the outline here that an example of a conclusion might be "Performance not there yet, will follow. Good for prototyping." That was a prescient remark, for it is our conclusion. Chapel is a comfortable language for programming for us and we expect it will get better with each half-yearly release. Its lineage is clear, with roots in the world of C and Unix, and this is the environment we use. Should we need to port a prototype back into that world, say the kd-tree class or the RANSAC algorithm, it seems straightforward enough to do. But first we need to determine if RANSAC can be tweaked and its accuracy with scaled images improved. Onward to the next project ...

Feedback

And that brings us to the end of the text. If you have comments or feedback, they would be very much appreciated. We can best be reached by e-mail at chapel_by_ex@primordand.com. Thank you for your time and attention.