# CLAM: Concise Linear Algebra Manipulation Language

Jeremy Andrus
Robert Martin
Kevin Sun
Yongxu Zhang

# Tutorial: Input and Output

**Image my_img = imgread(**<*file-name*>**);**

**imgwrite(**<*image-identifier*>**,** <*img-type*>**,** <*file-name*>**);**

Your first program (Image Copier/Converter):

```
Image input = imgread("source.jpg");
imgwrite(input, "png", "dest.png");
```

Also can use command line arguments:

```
imgread(1); /* reads first  argument */
imgwrite(input, "png", 2); /* writes to second arguments */
```

# Tutorial: Running Your Program

**> ./clam program1.clam**
Translates to C, compiles, outputs to a.out
**> ./clam -i program1.clam -o out**
Translates to C, compiles, outputs to "-o" file
**> ./clam -c program1.clam**
Translates to C, prints C code to clam_gen.c
**> ./clam -t program1.clam**
Debugging: Print abstract syntax tree

# Tutorial: Channels

**Channel**s are arrays of values for each pixel, such as Red, Green, and Blue.

**Image**s come with these three default channels when read.
Access using "**:**" operator (img:Red, img:Green, img:Blue)
Set/create channels using "=" operator

```
Image img1 = imgread("image.jpg");

img1:temp = img1:Blue;
img1:Blue = img1:Red;
img1:Red = img1:temp; /* swap channels */

/*Only Red, Green, and Blue channels are written:*/
imgwrite(img1,"jpg","image.jpg");
```

# Tutorial: Calculations

**Calc**ulations, which are to be applied to each pixel,
can be defined in two ways:
matrices (containing weights for neighboring pixels)
or C strings (containing references to the same pixel in other channels)

```
Calc m<Uint8> := [1 / 9] {1 1 1, 1 1 1, 1 1 1};
Calc Lum := #[(3*Red + 6*Green + 1*Blue) / 10]#;
```

C string calculations can be added to Images,
creating new channels (with the same name as the Calc):

```
srcimg |= Lum; /* srcimg:Lum is now valid */
```

The values of the Channel will be calculated on first use.

**Calc**s must have names! (defined once with "**:=**"):

```
srcimg |= #[Red + Green + Blue]#; /* INVALID */
```

Adding matrix calculations to Images is meaningless,
but they have other important uses...

# Tutorial: Kernels

Kernels are ordered collections of calculations.

```
Calc sobelGx<Uint8> := {-1 0 +1, -2 0 +2, -1 0 +1};
Calc sobelGy<Uint8> := {+1 +2 +1, 0 0 0, -1 -2 -1};
Calc sobelG<Uint8> :=
    #[sqrt(sobelGx * sobelGx + sobelGy * sobelGy)]#;
Kernel k = @sobelGx | @sobelGy | sobelG;
/* Calcs can refer to preceding Calcs in kernel */
/* "@" means generate value, but don't generate channel */
/* (this will make sense when we see convolutions) */

Calc sobelTheta := #[arctan(sobelGx/sobelGy)]#;
k |= sobelTheta;
/* don't have to add all Calcs at once */
```

# Tutorial: Convolutions

The "**" operator takes a **Channel** reference and a **Kernel**,
applies the calcs in sequence (matrices are applied to the specific **Channel** given),
and generates an **Image** with all the channels (**Calcs**) defined in that **Kernel** not prefixed with "@"

Continuing the previous example:

```
Image edges = srcimg:Lum ** sobel;
/* edges:sobelG and edges:sobelTheta now valid */
/* but not edges:sobelGx or edges:sobelGy */
```

# Tutorial: Full Program (Sobel Operator)

```
Image srcimg = imgread("someimage.jpg");

Calc Lum := #[(3*Red + 6*Green + 1*Blue)/10]#;
srcimg |= Lum ;

Calc sobelGx <Uint8> := [1 / 1]{ -1 0 +1 , -2 0 +2 , -1 0 +1 };
Calc sobelGy <Uint8> := [1 / 1]{ +1 +2 +1 , 0 0 0 , -1 -2 -1 };
Calc sobelG <Uint8> :=
   #[sqrt(sobelGx * sobelGx + sobelGy * sobelGy)]#;
Calc sobelTheta <Angle> := #[arctan(sobelGy / sobelGx)]#;

Kernel sobel = @sobelGx | @sobelGy | sobelG;
sobel |= sobelTheta;
Image edges = srcimg:Lum ** sobel;

output : Green = edges : sobelG;
output : Red = edges : sobelG;
output : Blue = edges : sobelG;

imgwrite(output, "jpg", "edges_of_someimage.jpg");
```

# Sobel Operator

Calculates gradient of intensity function,
i.e. detects edges: