MatCV - Proposal

Abhishek Walia (aw3011),
Anuraag Advani (ada2161),
Rahul Kapur (rk2749),
Shardendu Gautam (sg3391)

Contents

1 Introduction

1.1 Motivation

Our rationale behind MatCV is to come up with a syntax that makes matrix manipulation easier and more intuitive. Since many fields, such as computer vision and machine learning use matrix operations extensively, our language introduces some constructs that will allow beginners to get started easily. We named our language MatCV as we will primarily be focussing on matrix operations that will be more useful for computer vision related applications.

1.2 Description

MatCV will support primitive matrix operations such as transpose, inverse, determinant etc. We introduce a few constructs that will make looping over pixels, rows of a matrix, elements of a matrix, columns of a matrix as well as performing updates in these loops intuitive and more readable. Concatenation of matrices, creating matrices with zeros, read and display images, add and subtract pixels etc. are some other features that will be supported by the language. In addition to these features, we want to introduce another construct in which a programmer can create a matrix of functions, say F:

$$F = \begin{bmatrix} func1 & func2 \\ func3 & func4 \end{bmatrix} \tag{1}$$

And then can apply these corresponding functions to elements of another matrix with same dimensions:

$$A = \begin{bmatrix} 2 & 9 \\ 8 & -1 \end{bmatrix} \tag{2}$$

$$F(A) = \begin{bmatrix} func1(2) & func2(9) \\ func3(8) & func4(-1) \end{bmatrix} \tag{3}$$

2 Syntax

2.1 Data Types

MatCV will support following data types:

int	64 bit integers (32 bit integers will not be supported)
float	64 bit floating point numbers
boolean	True or False
matrix	m-by-n matrix which stores int/float type data
string	Stores sequence of UTF-8 characters

2.2 Operators

While considering operations between data types, we enforce some restrictions on the data types that can be used with each other. The operators we support are listed below:

Addition	+	Addition is supported between two matrices having the same dimensions. Addition of a matrix and scalar is not supported.
Subtraction	-	Subtraction is supported between two matrices having the same dimensions. Addition of a matrix and scalar is not supported.
Multiplication	*	Multiplication of two compatible matrices as well as multiplying a matrix and a scalar is supported.
Division	/	Division of two compatible matrices as well as dividing a matrix and a scalar is supported.
Transpose	'	Transpose of a matrix is supported.
Assignment	=	We assign an appropriate RHS to an appropriate LHS where type promotion is supported.

2.3 Comments

Multi - line and nested comments are supported:

```
/* This is a comment. Comments can be nested
and can be spread across multiple lines.
Comments have to be closed */
```

2.4 Keywords

MatCV will support following keywords:

row	used to iterate over the rows in a matrix
col	used to iterate over the columns in a matrix
ele	identifier to access each element in a matrix sequentially
var	declares a variable
const	modifies a variable to be immutable
if..else if..else	Supports standard conditional operations
for	loops over given elements
break	breaks out of loop
continue	returns control flow to the beginning of the loop
pixel	is a 1x3 matrix that is used to store RGB/YCrCb/HSV values corresponding to a pixel
exit	stops the program execution and returns control to the host environment

2.5 Library Functions

MatCV will provide some basic functions which can be extended to implement complicated functionality:

zeros(m,n)	returns a matrix containing only zeros of dimensions m x n
eye(m, n)	returns an identity matrix of dimensions m x n
inv(a)	computes the inverse of matrix A. Matrix inverse can also be computed by $\frac{1}{A}$
det(A)	returns the determinant of a matrix in float type
rank(A)	returns the rank of the matrix
readImage(imagePath)	reads an image from the given path
showImage(windowTitle,img)	shows the image in a new window with window title

Apart from these functions, we implement basic math functions such as sin(), cos(), round(), pow(), abs(), ceil(), floor(), log() etc.

3 Features

The following are a few features MatCV supports:

1. You can declare a new matrix using the following syntax:

   A = {1,2; 3,4};

   You can also declare a matrix of zeros of size 4x2 using

   A = zeros(4,2);

2. The **print** keyword can be used to print out information to the console. For example:

   A = {1,2; 3,4};
   print(A[0][1]);

   Will print 2 to the console. Row and column indexing starts from 0 in our language.

3. Row size and column size are stored as attributes for a variable internally represented as a matrix. If A is a matrix of size 5x7 then: **print(A.rowSize);** will print value 5 and **print(A.colSize);** will print the number of columns, that is 7.

4. Primitive matrix operations such as addition, subtraction, multiplication, transpose, inverse etc. are also provided by the language. You can invert the matrix A using:

   inverseOfA = 1/A;

   Alternatively, we could have used the library function **inv** to find the inverse:

   inverseOfA = inv(A);

5. The proposed language provides an intuitive way to iterate over all elements of a matrix.

   Keyword **ele** is used in the following fashion in order to iterate over all elements of matrix A. If you had the following matrix: A = {1,2; 3,4};

   Then the following code adds 1 to each element in the matrix:

   ```
   ele e:A{
     e = e + 1;
   }
   ```

   After the execution of above loop, the matrix A would look like:

```
A = {2,3; 4,5};
```

Inside the loop, element **e** contains attributes rowNum and colNum, that can be used to find the position of the element in the matrix. For example, if the current element in the loop corresponds to (3,2) in the matrix, then print(e.rowNum) will print 3 to the console.

There are two more variations to the above loop. You can add var in front of the variable name, using which you can change the value of the variable but the change will not be reflected in the matrix:

```
A = {1, 2, 3; 4, 5, 6};
ele var e:A{
    e = e + 3;
    print(e);
}
```

The above example prints 4, 5, 6...9 but the matrix A will still remain {1, 2, 3; 4, 5, 6}.

The const keyword can be used instead of the var keyword, which will throw an compilation error when **e** is changed in the loop. This makes sure that the user does not change the matrix unintentionally:

```
A = {1, 2, 3; 4, 5, 6};
ele var e:A{
    e = e + 3;
}
```

6. We can also iterate through the rows and columns easily. Keyword **row** is used in the following fashion in order to iterate over all rows of matrix. This example negates all the odd rows of A:

```
row r:A{
    if(r.rowNum % 2 ==1) {
        r = r * (-1);
    }
}
```

We can similarly use the keyword **column** to access the columns of the matrix.

7. The iterators also have attributes, rowNum and columnNum which output the row and column number at which the iterator is currently operating on.

4 Demo Code

The following code performs bi-cubic interpolation to zoom images in our language. Consider a third degree polynomial:

$$f(x) = ax^3 + bx^2 + cx + d$$

Suppose you have values v_0, v_1, v_2 and v_3 at x=-1, x=0, x=1 and x=2 respectively, we can estimate the value of a, b, c and d using:

$$\begin{gathered}
a = -\tfrac{1}{2}v_0 + \tfrac{3}{2}v_1 - \tfrac{3}{2}v_2 + \tfrac{1}{2}v_3 \\
b = v_0 - \tfrac{5}{2}v_1 + 2p_2 - \tfrac{1}{2}v_3 \\
c = -\tfrac{1}{2}v_0 + \tfrac{1}{2}v_1 \\
d = v_1
\end{gathered}$$

The above estimates of a, b c and d can be used to perform cubic interpolation, given by:

$$f(v_0, v_1, v_2, v_3, x) = (-\tfrac{1}{2}v_0 + \tfrac{3}{2}v_1 - \tfrac{3}{2}v_2 + \tfrac{1}{2}v_3)x^3 + (v_0 - \tfrac{5}{2}v_1 + 2v_2 - \tfrac{1}{2}v_3)x^2 + (-\tfrac{1}{2}v_0 + \tfrac{1}{2}v_1)x + v_1$$

In case of images, we use bi-cubic interpolation, which is essentially equivalent to performing cubic interpolation in two dimensions. If we consider a 4X4 grid of pixels, with each pixel having value v_{ij}, then we can perform bicubic interpolation using:

$$g(x, y) = f(f(v_{00}, v_{01}, v_{02}, v_{03}, y), f(v_{10}, v_{11}, v_{12}, v_{13}, y), f(v_{20}, v_{21}, v_{22}, v_{23}, y), f(v_{30}, v_{31}, v_{32}, v_{33}, y), x)$$

Demo code

```
void doCubicInterpolation(x, rowOfPixels)
{
    pixelVal = {};

    pixelVal = rowOfPixels[1] + 0.5 * x *(rowOfPixels[2] - rowOfPixels[0] +
   x * (2.0 * rowOfPixels[0] - 5.0 * rowOfPixels[1] +
   4.0 * rowOfPixels[2] - rowOfPixels[3] +
   x * (3.0 * (rowOfPixels[1] - rowOfPixels[2]) +
   rowOfPixels[3] - rowOfPixels[0])));


    return pixelVal;
}


void doBicubicInterpolation(pixelSquare, double x, double y)
{
    pixelAfterXInterPolation = {}; /*empty matrix*/

    row const r:pixelSquare
    {
        pixelAfterInterPolation = {pixelAfterXInterpolation,
        doCubicInterpolation(x, r)}; /*Append result to matrix*/
    }

    return doCubicInterpolation(y, pixelAfterXInterPolation);
}

function retrievePixel(in, x, y)
{

        if (x < 0)
        {
            x = 0;
        }
        if (y < 0)
        {
            y = 0;
        }
        if (x >= in.width)
        {
            x = in.width - 1;
        }
        if (y >= in.height)
```

```
        {
            y = in.height - 1;
        }

        return in[x][y]; /* return the pixel at location x, y */
}

func performBicubicInterpolation(in, xScale, yScale)
{
        width = scalingX * in.width;
        height = scalingY * in.height;

        out  = whiteImage(width, height);
        /* All pixel values will be 255,255,255 and
           by default all images have 3 channels. */

        xScale = 1 / xScale;
        yScale = 1 / yScale;

        pixel p:out /* for each pixel p in image out */
        {
            x = xScale * p.rowNum;
            y = yScale * p.columNum;

            pixelSquare =
            {   retrievePixel(in, x - 1, y - 1), retrievePixel(in, x, y - 1),
                retrievePixel(in, x + 1, y - 1), retrievePixel(in, x + 2, y - 1);

                retrievePixel(in, x - 1, y), retrievePixel(in, x, y),
                retrievePixel(in, x + 1, y), retrievePixel(in, x + 2, y);

                retrievePixel(in, x - 1, y + 1), retrievePixel(in, x, y + 1),
                retrievePixel(in, x + 1, y + 1), retrievePixel(in, x + 2, y + 1);

                retrievePixel(in, x - 1, y + 2), retrievePixel(in, x, y + 2),
                retrievePixel(in, x + 1, y + 2), retrievePixel(in, x + 2, y + 2)
            };
            /*pixelSquare is a 4X4 matrix of 'pixels'*/


            p = doBicubicInterpolation(pixelSquare,
                (((xScale * w) - x), ((yScale * h) - y));
        }

        return out;
}
```

```
func main()
{
    inputImage = readImage("path_to_image/img.jpg");
    /* Reads image from path*/

        scalingX = 2;
        scalingY = 1.5;

        showImage("Input Image", inputImage);

        outImage = performBicubicInterpolation(inputImage, scalingX, scalingY);

        showImage("Output Image", outImage);
}

main();
```