Matrix Computations on GPU with ArrayFire - Python and ArrayFire - C/C++

Andrzej Chrzęszczyk

Jan Kochanowski University

Version 2017

Foreword

In recent years the Graphics Processing Units (GPUs) designed to efficiently manipulate computer graphics are more and more often used to General Purpose computing on GPU (GPGPU). Nvidia's CUDA and OpenCL platforms allow for general purpose parallel programming on modern graphics devices. Unfortunately many owners of powerful graphic cards are not experienced programmers and can find these platforms quite difficult. The purpose of this document is to make the first steps in using modern graphics cards to general purpose computations simpler.
We want to present the ArrayFire software library which in our opinion allows to start computations on GPU in the easiest way. The necessary software can be downloaded from:

`https://arrayfire.com/` .

`https://github.com/arrayfire/arrayfire` .

`https://github.com/arrayfire/arrayfire-python` .

In the present text we describe the ArrayFire 3.5, first in `Python` and next in `C++` version. Our purpose is to demonstrate how Arrayfire can be used in efficient dense matrix computations but let us mention that the library has also some support for sparse matrices.

What's new in this version.

- All code samples in this version were checked with ArrayFire 3.5. Most of examples from our previous text with similar title: `http://arrayfire.com/wp-content/uploads/2014/03/arrayfire_matrix_computations_chrzeszczyk_cover.pdf` do not work in ArrayFire 3.5, due to changes in the syntax, so extensive changes in our older text were necessary.

- In the first part, where Python language is used, we compare ArrayFire with Intel Python Distribution: `https://software.intel.com/en-us/distribution-for-python`, which seems to be the best choice in Python scientific computations on CPU. The comparison was not possible in the previous version since Intel Python Distribution started in 2016.

- We have checked the efficiency of ArrayFire on the system with Nvidia GTX 1080 Graphics card and Intel i7-6700K processor. Such a system seems to be sufficient for training and gives a good performance in single precision, but the double precision capabilities of GTX cards are restricted and in professional applications Tesla cards are preferred.

A word of warning. When copy-pasting from a pdf file, some systems replace the single quotation mark by a similar but slightly different symbol, and this leads (in Python) to the message: `SyntaxError:invalid syntax`. When you encounter a similar message, check if the proper quotations marks were used in the pasted code.

Contents

Chapter 1

ArrayFire-Python

1.1 Introductory remarks

In the present chapter we assume that the user has an access to a system with ArrayFire and Python installed.
This approach allows for executing our examples step by step and for observing the obtained results. Of course, it is also possible to create python scripts and execute all the commands at once. Note that in the interactive session the `print` commands can be omitted. We use these commands to allow for both options.

New users should probably begin with the commands:

```
$python                       # Start an interactive Python session
import arrayfire as af        # Import ArrayFire module
print(af.info())              # Print some details of
                              # installed software and hardware
```

```
ArrayFire v3.5.0 (CUDA, 64-bit Linux, build 05999f3)
Platform: CUDA Toolkit 8, Driver: 378.13
[0] GeForce GTX 1080, 8111 MB, CUDA Compute 6.1
```

Using the command `help(af)` one can list arrayfire sub-packages

```
.................
PACKAGE CONTENTS
    algorithm
    arith
    array
```

```
    base
    bcast
    blas
    cuda
    data
    device
    features
    graphics
    image
    index
    interop
    lapack
    library
    opencl
    random
    signal
    sparse
    statistics
    tests (package)
    timer
    util
    vision
..............
```

To see all possible ArrayFire-Python commands in Python interactive session it suffices to write from Python command line `af.[TAB]` (tabulation key), or alternatively, use the commands

```
l=dir(af)
for i in range(20,len(l)/2): print '%-30s  %-30s' % (l[2*i],l[2*i+1])
```

As the result one obtains four pages of output, but we show only a few lines.

```
...                            ...
abs                            accum
acos                           acosh
algorithm                      all_true
alloc_device                   alloc_host
alloc_pinned                   any_true
array                          asin
...                            ...
...                            ...
```

```
lower                          lu
lu_inplace                     match_template
math                           matmul
maxfilt                        maxof
mean                           mean_shift
...                            ...
...                            ...
qr                             qr_inplace
randn                          random
randu                          range
rank                           read_array
...                            ...
...                            ...
solve                          solve_lu
sort                           sort_by_key
sort_index                     sparse
stdev                          sum
susan                          svd
svd_inplace                    sync
...                            ...
...                            ...
```

To any element of the obtained list the help function can be used. This gives us the access to the information concerning the chosen function. For example:

```
help(af.solve)

Help on function solve in module arrayfire.lapack:

solve(A, B, options=<arrayfire.library.MATPROP object>)
    Solve a system of linear equations.

    Parameters
    ----------

    A: af.Array
       A 2 dimensional arrayfire array representing the coef-
       ficients of the system.

    B: af.Array
```

```
    A 1 or 2 dimensional arrayfire array representing the
    constants of the system.

options: optional: af.MATPROP. default: af.MATPROP.NONE.
    - Additional options to speed up computations.
    - Currently needs to be one of ‘af.MATPROP.NONE‘,
     ‘af.MATPROP.LOWER‘, ‘af.MATPROP.UPPER‘.

Returns
-------
X: af.Array
    A 1 or 2 dimensional arrayfire array representing the
    unknowns  in the system.
```

1.2 Defining arrays

ArrayFire Python interface allows for easy definitions of vectors and matrices. The command `help(af.Dtype)` shows that we have all necessary types at our disposal. We can compare them with corresponding Numpy types.

ArrayFire	**Numpy**	
af.Dtype.f32	np.float32	for float
af.Dtype.f64	np.float64	for double
af.Dtype.b8	np.bool_	for bool
af.Dtype.u8	np.uint8	for unsigned char
af.Dtype.s16	np.int16	for signed 16 bit integer
af.Dtype.u16	np.uint16	for unsigned 16 bit integer
af.Dtype.s32	np.int32	for signed 32 bit integer
af.Dtype.u32	np.uint32	for unsigned 32 bit integer
af.Dtype.s64	np.int64	for signed 64 bit integer
af.Dtype.u64	np.uint64	for unsigned 64 bit integer
af.Dtype.c32	np.complex64	for 32 bit complex number
af.Dtype.c64	np.complex128	for 64 bit complex number

A new ArrayFire arrays can be defined using python `array` module or lists, see `help(af.array)`. The default type in ArrayFire is `float32`, while in Numpy the default is `float64`.

```
##### Arrayfire #####                  ##### Numpy #####

import arrayfire as af                 import numpy as np
from array import array                from array import array
a=array('f',(1,2,3,4))                 a=array('f',(1,2,3,4))
b=af.Array(a)   # or                   a=np.array(a,np.foat32)  # or
b=af.Array([1,2,3,4])                  b=np.array([1,2,3,4],'float32')
af.display(b,0)                        print(b)
    1                                  [ 1.  2.  3.  4.]
    2
    3
    4
b=af.Array(a,(2,2))                b=np.array([[1,2],[3,4]],'float32')
af.display(b,0)                    print(b)
    1     3                        [[ 1.  2.]
    2     4                         [ 3.  4.]]
```

To obtain new ArrayFire arrays one can also use Numpy arrays and the `af.Array` function.

```
import numpy as np
x=np.array([[0,1,2],[3,4,5],[6,7,8]],'float32')
print(x)
[[ 0.,  1.,  2.],
 [ 3.,  4.,  5.],
 [ 6.,  7.,  8.]]                  # Numpy uses row major format

import arrayfire as af
y=af.Array(x.ctypes.data,x.shape,x.dtype.char)
af.display(y,0)
    0     3     6                  # af.Array() uses
    1     4     7                  # column major format
    2     5     8
# the same, shorter way
y=af.Array(range(9),(3,3))
af.display(y,0)
    0     3     6                  # af.Array() uses
    1     4     7                  # column major format
    2     5     8

z=af.np_to_af_array(x);
```

```
af.display(z,0)
    0     1     2                          # row major version
    3     4     5
    6     7     8
```

The simplest way to define an array in ArrayFire-Python is to use one of the commands `constant` or `identity`. In Numpy one can also use `zeros`, `ones`.

```
##### ArrayFire #####

import arrayfire as af
# 3x3 matrix of zeros
a=af.constant(0,3,3)
af.display(a,0)
0     0     0
0     0     0
0     0     0

# 3x3 matrix of ones
a=af.constant(1,3,3)
af.display(a,0)
1     1     1
1     1     1
1     1     1

# 3x3 identity matrix
a=af.identity(3,3)
af.display(a,0)
1     0     0
0     1     0
0     0     1
```

```
##### Numpy #####

import numpy as np
# 3x3 matrix of zeros
a=np.zeros((3,3),'float32')
print(a)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]

# 3x3 matrix of ones
np.ones((3,3),'float32')
print(a)
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

# 3x3 identity matrix
a=np.identity(3,'float32')
print(a)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

The `constant` function from ArrayFire one can replace in Numpy by `ones` multiplied by a constant, or by the pair `empty` and `fill`.

```
##### Arrayfire #####

# 3x3 constant matrix
import arrayfire as af
a=af.constant(2,3,3)
```

```
##### Numpy #####

# 3x3 constant matrix
import numpy as np
a=2*np.ones((3,3),'float32')
```

```
af.display(a,0)
2     2     2
2     2     2
2     2     2
```

```
a=np.empty((3,3),'float32')
a.fill(2)
print(a)
[[ 2.  2.  2.]
 [ 2.  2.  2.]
 [ 2.  2.  2.]
```

There are also ArrayFire `range` and Numpy `arange` functions.

```
##### ArrayFire #####

import arrayfire as af
# Range 0,1,...,8
a=af.range(9);
af.display(a,0) # as column
    0
    1
    2
    3
    4
    5
    6
    7
    8
```

```
##### Numpy #####

import numpy as np
# Range 0,1,...,8
a=np.arange(9.0).astype('float32')
print(a)      # as row
[ 0. 1. 2. 3. 4. 5. 6. 7. 8.]
```

The dimensions can be modified using `af.moddims` in Arrayfire and `reshape` in Numpy.

```
##### ArrayFire #####

# continuation, a defined above
A=af.moddims(a,3,3)
af.display(A,0)    # column-wise
0     3     6
1     4     7
2     5     8
```

```
##### Numpy #####

# continuation, a defined above
A=np.reshape(a,(3,3))
print(A)   # row-wise
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
```

Python lambda functions can be used to introduce matrices with elements given by a formula.

```
import numpy as np
import arrayfire as af
a=np.fromfunction(lambda i,j:i+j,(3,3),dtype='float32')
A=af.np_to_af_array(a)
af.display(A,0)
0     1     2                         # a[i,j]=i+j
1     2     3
2     3     4
```

Here is an example with complex entries:

```
#continuation
b=(a+1j*a).astype('complex64')      # Complex array in Numpy
B= af.np_to_af_array(b)             # ArrayFire complex array
af.display(B,0)
(0,0)      (1,1)      (2,2)
(1,1)      (2,2)      (3,3)
(2,2)      (3,3)      (4,4)
```

There are also `af.diag, np.diag` functions, which define diagonal matrices.

```
import arrayfire as af
a=af.diag(af.range(4)+3,num=0,extract=False)    # Diagonal matrix
af.display(a,0)                                 # in ArrayFire
3      0      0      0
0      4      0      0
0      0      5      0
0      0      0      6
```

```
import numpy as np
np.diag(np.arange(4)+3).astype('float32')       # Diagonal matrix
print(a)                                        # in Numpy
[[ 3.  0.  0.  0.]
 [ 0.  4.  0.  0.]
 [ 0.  0.  5.  0.]
 [ 0.  0.  0.  6.]]
```

1.3 Random arrays

Very often random arrays from uniform or normal distributions are used.

```
##### ArrayFire #####

import arrayfire as af
# Random array unif.distr
ru=af.randu(3,3)
af.display(ru)
0.5170     0.6734     0.1583
0.3335     0.1631     0.9808
0.1845     0.2072     0.4279

# Random array, norm.distr
# complex, double
rn=af.randn(3,2,1,1,af.Dtype.c64)
af.display(rn,3)
( 2.275,0.321)   (1.551, 1.353)
( 0.575,1.047)   (0.042,-1.065)
(-0.999,0.554)   (0.336,-1.306)
```

```
##### Numpy #####

import numpy as np
from numpy.random import rand
ru=rand(3,3).astype('float32')
np.set_printoptions(precision=4)
print(ru)
[[ 0.4447  0.642   0.5338]
 [ 0.0527  0.8069  0.81  ]
 [ 0.3728  0.6747  0.5486]]

from numpy.random import randn
rn=randn(3,2)+1j*randn(3,2)
np.set_printoptions(precision=3)
print(rn)
[[-0.701-0.907j -1.319-0.657j]
 [-1.200+0.049j  0.427+0.041j]
 [-1.682-1.098j  0.918+0.085j]]
```

In Intel Python distribution there is also possibility to use `random_intel` package, which closely mirrors the design of `numpy.random` and uses MKL vector statistics library to achieve better performance. A more detailed description of `random_intel` can be found using the commands

```
import numpy.random_intel
help(numpy.random_intel)
```

Below we give `random_intel` (double precision) versions of previous random arrays.

```
import numpy as np
import numpy.random_intel as rnd
ru=rnd.rand(3,3)           # uniform random array from random_intel
np.set_printoptions(precision=4)
print(ru)
[[ 0.297   0.2542  0.2081]
 [ 0.295   0.5661  0.732 ]
 [ 0.6     0.8514  0.6889]]
```

```
rn=rnd.randn(3,3)          # normal random array from random_intel
print(rn)
[[ 0.4529 -1.4122  0.4357]
 [ 1.0728  0.1446  0.0862]
 [-1.2764  1.5734 -0.8745]]
```

Using the functions `ceil, floor` and `round`, `around` one can obtain random arrays with integer entries. We demonstrate how to use these functions (but let us mention that there is also `randint` function which generates random integer arrays).

```
##### ArrayFire #####

import arrayfire as af
A=af.randn(3,3)*5

af.display(A)
 4.7901    -1.7087    -2.5217
 5.3208    -6.0836    -0.2033
-2.1504    -0.8603     2.3320

# Rounding array elements
a=af.round(A)
af.display(a,0)
 5     -2     -3
 5     -6     -0
-2     -1      2

# Applying floor function
a=af.floor(A)
af.display(a,0)
 4     -2     -3
 5     -7     -1
-3     -1      2

# Applying ceiling function
a=af.ceil(A)
af.display(a,0)
 5     -1     -2
 6     -6     -0
-2     -0      3
```

```
##### Numpy #####

import numpy as np
from numpy.random import randn
A=5*randn(3,3).astype('float32')
np.set_printoptions(precision=4)
print(A)
[[ 1.4424   2.7763   5.7765]
 [11.9378  -0.1201   3.5055]
 [ 8.6238   1.7922   4.2064]]

a=np.around(A)
print(a)
[[  1.   3.   6.]
 [ 12.  -0.   4.]
 [  9.   2.   4.]]

a=np.floor(A)
print(a)
[[  1.   2.   5.]
 [ 11.  -1.   3.]
 [  8.   1.   4.]]

a=np.ceil(A)
print(a)
[[  2.   3.   6.]
 [ 12.  -0.   4.]
 [  9.   2.   5.]]
```

Generation of random arrays gives us the opportunity to check the efficiency of ArrayFire. Let us check the time needed for generation of a 8000x8000 single precision random array on GTX 1080 card.

```
import arrayfire as af
from time import time
N=8000
ru=af.randu(N,N)                    # Warm up
t0=time();ru=af.randu(N,N);af.eval(ru);af.sync();t=time()-t0
print "time: %.4f sec." % t
time: 0.0012 sec.                   # uniform matr. generation time

rn=af.randn(N,N)                    # Warm up
t0=time();rn=af.randn(N,N);af.eval(rn);af.sync();t=time()-t0
print "time: %.4f sec." % t
time: 0.0014 sec.                   # normal matr. generation time
```

Similar computations in Numpy take much more time but the single and double precision versions have similar performance. Below we present uniform, single precision versions in three ways.

```
import numpy as np
from time import time
import numpy.random_intel as rndi
N=8000
# uniform, random matrix  in random_intel, single precision  (1)
t0=time();ru=rndi.ranf(N*N);t=time()-t0
print t
0.168909788132

# uniform, random matrix  in random_intel, single precision  (2)
t0=time();ru=rndi.random_sample((N,N));t=time()-t0
print(t)
0.161665201187

# uniform, random matrix  in random_intel, single precision  (3)
t0=time();ru=rndi.rand(N,N).astype(np.float);t=time()-t0
print t
0.261273860931
```

And now the uniform, double precision version.

```
# uniform, random matrix  in random_intel, double precision
t0=time();ru=rndi.rand(N,N);t=time()-t0
print t
0.167246103287
print(ru.dtype)
float64
```

The corresponding computations with normal distribution:

```
# normal random matrix  in random_intel, single precision
rn=rndi.randn(N,N).astype(np.float32)
t0=time();rn=rndi.randn(N,N).astype(np.float32);t=time()-t0
print t
0.31102602005

# normal random matrix  in random_intel, double precision
t0=time();rn=rndi.randn(N,N);t=time()-t0
print t
0.243793010712
```

1.4 Rearranging arrays

The transpose, conjugate and conjugate transpose operations are particularly easy in ArrayFire

```
##### ArrayFire #####

import arrayfire as af
# 3x3 matrix A in ArrayFire
A=af.moddims(af.range(9),3,3)
af.display(A,0) # column-wise
     0      3      6
     1      4      7
     2      5      8

# Transposition of A
AT=A.T
af.display(AT,0)
```

```
##### Numpy #####

import numpy as np
# 3x3 matrix a in numpy
a=np.arange(9.0).reshape(3,3)
print(a)  # row-wise
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]


aT=a.T
print(aT)
```

```
    0      1      2                    [[ 0.  3.  6.]
    3      4      5                     [ 1.  4.  7.]
    6      7      8                     [ 2.  5.  8.]]
```

Consider a complex example with conjugation and conjugate transposition operations.

```
##### ArrayFire #####                  ##### Numpy #####
import arrayfire as af
import numpy as np                     import numpy as np
# 3x3 complex matrix in Arrayfire      # 3x3 complex matrix in Numpy
a=np.arange(9.0).reshape(3,3)          a=np.arange(9.0).reshape(3,3)
B=af.np_to_af_array(a+2j*a)            b=a+2j*a
af.display(B,0)                        print(b)
(0,0)      (1,2)      (2,4)           [[ 0. +0.j  1. +2.j  2. +4.j]
(3,6)      (4,8)      (5,10)           [ 3. +6.j  4. +8.j  5.+10.j]
(6,12)     (7,14)     (8,16)           [ 6.+12.j  7.+14.j  8.+16.j]]

# Conjugation
BC=af.conjg(B)                         bc=np.conj(b) # bc=b.conj( )
af.display(BC,0)                       print(bc)
(0,-0)     (1,-2)     (2,-4)          [[ 0. -0.j  1. -2.j  2. -4.j]
(3,-6)     (4,-8)     (5,-10)          [ 3. -6.j  4. -8.j  5.-10.j]
(6,-12)    (7,-14)    (8,-16)          [ 6.-12.j  7.-14.j  8.-16.j]]

# Conjugate transposition
BH=B.H                                 bh=b.conj().T
af.display(BH,0)                       print(bh)
(0,-0)     (3,-6)     (6,-12)         [[ 0. -0.j  3. -6.j  6.-12.j]
(1,-2)     (4,-8)     (7,-14)          [ 1. -2.j  4. -8.j  7.-14.j]
 (2,-4)    (5,-10)    (8,-16)          [ 2. -4.j  5.-10.j  8.-16.j]]
```

One can also flip the array horizontally or vertically:

```
##### ArrayFire #####                  ##### Numpy #####

A=af.moddims(af.range(9),3,3)          a=np.arange(9.0).reshape(3,3)
af.display(A,0) # column-wise          print(a)  # row-wise
    0      3      6                   [[ 0.  1.  2.]
    1      4      7                    [ 3.  4.  5.]
    2      5      8                    [ 6.  7.  8.]]
```

```
# Flip horizontally
AFH=af.flip(A)
af.display(AFH,0)
     2      5      8
     1      4      7
     0      3      6
```

```
afh=np.flipud(a)
print(afh)
[[ 6.  7.  8.]
 [ 3.  4.  5.]
 [ 0.  1.  2.]]
```

```
# Flip vertically
AFV=af.flip(A,1)
af.display(AFV,0)
     6      3      0
     7      4      1
     8      5      2
```

```
afv=np.fliplr(a)
print(afv)
[[ 2.  1.  0.]
 [ 5.  4.  3.]
 [ 8.  7.  6.]]
```

The array can be flattened and upper or lower triangular parts can be extracted.

```
##### ArrayFire #####

A=af.moddims(af.range(9),3,3)
af.display(A,0) # column-wise
     0      3      6
     1      4      7
     2      5      8
```

```
##### Numpy #####

a=np.arange(9.0).reshape(3,3)
print(a)  # row-wise
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
```

```
# Flattened A
AF=af.flat(A)
af.display(AF,0)
     0
     1
     2
     3
     4
     5
     6
     7
     8
```

```
af=a.flatten()
print(af)
[ 0. 1. 2. 3. 4. 5. 6. 7. 8.]
```

```
# Lower triangular part
AL=af.lower(A)
af.display(AL,0)
```

```
al=np.tril(a)
print(al)
```

```
     0      0      0                  [[ 0.  0.  0.]
     1      4      0                   [ 3.  4.  0.]
     2      5      8                   [ 6.  7.  8.]]

# Upper triangular part
AU=af.upper(A)                        au=np.triu(a)
af.display(AU,0)                      print(au)
     0      3      6                  [[ 0.  1.  2.]
     0      4      7                   [ 0.  4.  5.]
     0      0      8                   [ 0.  0.  8.]]
```

There is also shift operation.

```
##### ArrayFire #####                 ##### Numpy #####

AF=af.range(9)                        a=np.arange(9.0)
af.display(AF,0)                      print(a)
     0                                [0. 1. 2. 3. 4. 5. 6. 7. 8.]
     1
     2
     3
     4
     5
     6
     7
     8
# Shift
S=af.shift(AF,1)                      s=np.roll(a,1)
af.display(S,0)                       print(s)
     8                                [8. 0. 1. 2. 3. 4. 5. 6. 7.]
     0
     1
     2
     3
     4
     5
     6
     7
```

1.5 Matrix addition multiplication and powering

To obtain the sum, the difference and the product of two matrices one can use the operations +, – and `matmul`.

```
##### ArrayFire #####

import arrayfire as af
A=af.moddims(af.range(9),3,3)
af.display(A,0)
     0      3      6
     1      4      7
     2      5      8

# Matrix addition in ArrayFire
I=af.identity(3,3)
B=A+I
af.display(B,0)
     1      3      6
     1      5      7
     2      5      9

# Matrix multiplication,ArrayFire
B=af.matmul(A,I)
af.display(B,0)
     0      3      6
     1      4      7
     2      5      8
```

```
##### Numpy #####

import numpy as np
a=np.arange(9.0).reshape(3,3)
print(a)
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]

# Matrix addition in Numpy
I=np.identity(3)
B=a+I
print(B)
[[ 1.  1.  2.]
 [ 3.  5.  5.]
 [ 6.  7.  9.]]

# Matrix multiplication,Numpy
B=np.dot(a,I)
print(B)
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
```

There is also the element-wise version of multiplication.

```
##### ArrayFire #####

import arrayfire as af
A=af.moddims(af.range(9),3,3)
# Element-wise multiplication A by A
B=A*A
af.display(B,0)
     0      9     36
     1     16     49
     4     25     64
```

```
##### Numpy #####

import numpy as np
a=np.arange(9.0).reshape(3,3)

B=a*a
print(B)
[[  0.   1.   4.]
 [  9.  16.  25.]
 [ 36.  49.  64.]]
```

The element-wise power can be obtained using the `pow` function:

```
##### ArrayFire #####

import arrayfire as af
A=af.moddims(af.range(9),3,3)
# Element-wise power
B=af.pow(A,2)
af.display(B,0)
     0      9     36
     1     16     49
     4     25     64
```

```
##### Numpy #####

import numpy as np
a=np.arange(9.0).reshape(3,3)

B=np.power(a,2)
print(B)
[[  0.   1.   4.]
 [  9.  16.  25.]
 [ 36.  49.  64.]]
```

The matrix product operation gives us a next opportunity to check the ArrayFire performance and compare it to Numpy/Scipy performance.

```
from time import time
import arrayfire as af
N=8000
a=af.randu(N,N)
t0=time();b=af.matmul(a,a);af.sync();t=time()-t0
                                    # 8000x8000 matr. multipl.
print t                             # on GPU, using ArrayFire
0.1390209198                        # Multiplication time
                                    # on GTX 1080
import numpy as np
import numpy.random_intel as rndi
N=8000
a=np.random.rand(8000,8000).astype('float32')
t0=time();b=np.dot(a,a);t=time()-t0
print t
4.43243217468                       # CPU i7-6700, Numpy with MKL

import scipy.linalg as la
t0=time();b=la.blas.sgemm(1.0,a,a);t=time()-t0
print t
2.72001791                          # CPU i7-6700, Scipy with MKL
```

1.6 Sums and products of elements

Of course we have at our disposal the possibility of summing or multiplying elements of an array. It is possible to sum/multiply columns, rows or all

elements.
Let us begin with simple examples.

```
##### ArrayFire #####

import arrayfire as af
# a: 3x3 matrix of ones
a=af.constant(1,3,3)
af.display(a,0)
     1      1      1
     1      1      1
     1      1      1

# Sums of columns
ac=af.sum(a,0)
af.display(ac,0)
     3      3      3

# Sums of rows
ar=af.sum(a,1)
af.display(ar,0)
     3
     3
     3

# Sum of all elements
asum=af.sum(a)
print(asum)
9.0

# 2*a
a=a*2
af.display(a,0)
     2      2      2
     2      2      2
     2      2      2

# Product of columns
ac=af.product(a,0)
af.display(ac,0)
     8      8      8
```

```
##### Numpy #####

import numpy as np
# a: 3x3 matrix of ones
a=np.ones((3,3),'f')
print(a)
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

# Sums of columns
ac=np.sum(a,0)
print(ac)
[ 3.  3.  3.]

# Sums of rows
ar=np.sum(a,1)
print(ar)
[ 3.  3.  3.]

# Sum of all elements
asum=np.sum(a)
print(asum)
9.0

# 2*a
a=a*2
print(a)
[[ 2.  2.  2.]
 [ 2.  2.  2.]
 [ 2.  2.  2.]]

# Product of columns
ac=np.prod(a,0)
print(ac)
[ 8.  8.  8.]
```

```
# Products of rows
ar=af.product(a,1)
af.display(ar,0)
     8
     8
     8

# Product of all elements
ap=af.product(a)
print(ap)
512.0
```

```
# Products of rows
ar=np.prod(a,1)
print(ar)
[ 8.  8.  8.]

# Product of all elements
ap=np.prod(a)
print(ap)
512.0
```

Now let us use ArrayFire to check the Euler's formula:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}.$$

```
##### ArrayFire #####

import arrayfire as af
# a=[1,2,...,10^8]
a=af.range(8000)+1
# sum_1^{10^8} 1/k^2
s=af.sum(1./(a*a))
print(s)
1.64480900764
```

```
##### Numpy #####

import numpy as np
# a=[1,2,...,10^8]
a=np.arange(8000)+1
# sum_1^{10^8} 1/k^2
s=np.sum(1./(a*a))
print(s)
1.64480907466

# pi^2/6 in numpy
print(np.pi**2/6)
1.64493406685
```

1.7 Mean, variance, standard deviation and histograms

ArrayFire-Python function `mean` allows for an easy computation of the average of elements of an array. As in the case of `sum` or `prod` one can compute the mean of rows, columns or all elements.

```
##### ArrayFire #####

import arrayfire as af
# 3x3 matrix
A=af.moddims(af.range(9),3,3)
af.display(A,0)
     0      3      6
     1      4      7
     2      5      8

# Averages of columns
ac=af.mean(A,dim=0)
af.display(ac,0)
     1      4      7

# Averages of rows
ar=af.mean(A,dim=1)
af.display(ar,0)
     3
     4
     5

# Average of elements
am=af.mean(A)
print(am)
4.0
```

```
##### Numpy #####

import numpy as np
# 3x3 matrix
a=np.arange(9.0).reshape(3,3)
print(a)
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]

# Averages of columns
ac=np.mean(a,0)
print(ac)
[ 3.  4.  5.]

# Averages of rows
ar=np.mean(a,1)
print(ar)
[ 1.  4.  7.]

# Average of elements
am=np.mean(a)
print(am)
4.0
```

The `var` and `stdev` functions in ArrayFire-Python compute the variance and the standard deviation respectively. Consider the uniform distribution first.

```
##### ArrayFire #####

import arrayfire as af
N=10000000;x=af.randu(N)
m=af.mean(x);
# var(x)=mean(x^2)-mean(x)^2
v=af.mean(x*x)-m*m
print(m)
# Theoretical mean: 1/2=0.5
0.499969303608
```

```
##### Numpy #####
import numpy as np
import numpy.random_intel as rndi
N=10000000;x=rndi.rand(N)
m=np.mean(x)
# var(x)=mean(x^2)-mean(x)^2
v=np.mean(x*x)-m*m
print(m)
# Theoretical mean: 1/2=0.5
0.499889897311
```

```
# Variance with the help of mean
print(v)
0.0833306899185
# Theoretical variance:
# 1/12=0.08333333...

# ArrayFire variance
av=af.var(x)
print(av)
0.0833304673433

# Standard deviation, Arrayfire
sd=af.stdev(x)
print(sd)
0.288670152426
# Theoretical standard dev:
# sqrt(1/12)= 0.28867513...
```

```
# Variance with the help of mean
print(v)
0.0833273097913
    # Theoretical variance:
    # 1/12=0.08333333...

    # Numpy variance
    av=np.var(x)
    print(av)
    0.0833273097913

    # Standard deviation, Numpy
    sd=np.std(x)
    print(sd)
    0.288664701325
    # Theoretical standard dev:
    # sqrt(1/12)= 0.28867513...
```

In the case of normal distribution we obtain:

```
##### ArrayFire #####

N=10000000;x=af.randn(N)
m=af.mean(x);v=af.mean(x*x)-m*m
print(m)
-0.000223015042138
# Theoretical mean: 0

# Variance with the help of mean
print(v)
0.999524424408
# Theoretical variance: 1

# ArrayFire variance
av=af.var(x)
print(av)
0.999524891376

# Standard deviation, ArrayFire
sd=af.stdev(x)
print(sd)
```

```
##### Numpy #####

N=10000000;x=rndi.randn(N)
m=np.mean(x);v=np.mean(x*x)-m*m
print(m)
0.00023204327067
# Theoretical mean: 0

# Variance with the help of mean
print(v)
0.999263876548
# Theoretical variance: 1

# Numpy variance
av=np.var(x)
print(av)
0.999263876548

# Standard deviation, Numpy
sd=np.std(x)
print(sd)
```

```
0.999762356281                          0.999631870514
# Theoretical standard dev              # Theoretical standard dev
# sqrt(1)=1                             # sqrt(1)=1
```

ArrayFire-Python interface contains also a fast histogram function.

```
import arrayfire as af
N=8000
ru=af.randu(N,N)                          # 8000x8000 uniformly
from time import time                     #  distributed array
t0=time();h=af.histogram(ru,100);af.sync();t=time()-t0
print "time:",t
time: 0.00322985649109

rn=af.randn(N,N)                          # 8000x8000 normally
                                          # distributed array
t0=time();h=af.histogram(rn,100);af.sync();t=time()-t0
print "time:",t
time: 0.00327897071838

# Write the next 7 lines to the file hist.py
import arrayfire as af
hist_win = af.Window(1024, 512, "Histogram")
N=1000
rn=af.randn(N,N)
hist = af.histogram(rn,100)
while (not hist_win.close()):
    hist_win.hist(hist, 0, 100)
# python hist.py
```

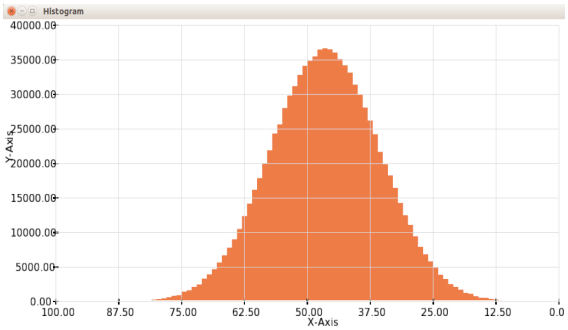

Figure 1.0: Histogram in ArrayFire

The same computations in Numpy:

```
import numpy as np
import numpy.random_intel as rndi
from time import time
N=8000                                  # 8000x8000 uniformly
ru=rndi.rand(N,N).astype('f')           #  distributed array
t0=time();h=np.histogram(ru,100);t=time()-t0
print "time:",t
time: 0.363550901413
                                        # 8000x8000 normally
rn=rndi.randn(N,N).astype('f')          # distributed array
t0=time();h=np.histogram(rn,100);t=time()-t0
print "time:",t
time: 0.448377132416ru=rndi.rand(N,N).astype('f')
t0=time();h=np.histogram(ru,100);t=time()-t0
print "time:",t
time: 0.448377132416
```

1.8 Solving linear systems

ArrayFire-Python and Numpy allow for efficient numerical solving of linear systems

$$Ax = B,$$

where A, B are ArrayFire or Numpy arrays.

```
##### ArrayFire #####

import arrayfire as af

# Random coefficients matrix
A=af.randu(3,3)
# Random RHS
# RHS will be equal to A*b
b=af.randu(3,1)
af.display(b)
    0.1570
    0.3725
    0.9624
```

```
##### Numpy/Scipy #####

import numpy as np
import numpy.random_intel as rndi
import scipy.linalg as la
a=rndi.rand(3,3).astype('f')
# Random RHS
# RHS will be equal to a*b
b=rndi.rand(3,1).astype('f')
print(b)
[[ 0.32301673]
 [ 0.30273142]
 [ 0.83872432]]
```

```
# RHS                                  # RHS
B=af.matmul(A,b)                       B=np.dot(a,b)
# Solution X should be equal to b          # Sol. X should be equal to b
X=af.solve(A,B)                            x=la.solve(a,B)
af.display(X)
    0.1570                                 [[ 0.32301685]
    0.3725                                  [ 0.30273178]
    0.9624                                  [ 0.83872396]]
# Max error                            # Max error
print(af.max(af.abs(X-b)))             print(np.max(np.abs(x-b))
2.83122062683e-07                      3.57628e-07
print(A.is_single())                   print(a.dtype)
True                                   float32
```

On GTX 1080 card we solved an 8000x8000 system using single and double precision.

```
import arrayfire as af
from time import time
N=8000;A=af.randu(N,N)           # 8000x8000 single precision
b=af.randu(N,1);B=af.matmul(A,b)
t0=time();X=af.solve(A,B);af.eval(X);af.sync();t=time()-t0
print "time:",t
time: 0.194478988647             # Solving time in single prec.
                                 # on GTX 1080
print(af.max(af.abs(X-b)))       # Max error
0.00577610623837

print(X.is_single())             # solution in single precision
True

                                 # 8000x8000 double precision
N=8000;A=af.randu(N,N,dtype=af.Dtype.f64)
b=af.randu(N,1,dtype=af.Dtype.f64);B=af.matmul(A,b)
t0=time();X=af.solve(A,B);af.eval(X);af.sync();t=time()-t0
print t                          # Solving time in double prec.
1.29586195946                    # on GTX 1080

print(af.max(af.abs(X-b)))       # Max error
2.83069734319e-10
```

```
print(X.is_double())             # solution in double precision
True
```

For comparison, let us perform analogous computations in intel python with single and double precision.

```
import numpy as np
from time import time
import numpy.random_intel as rndi
import scipy.linalg as la
N=8000                           # 8000x8000 single precision
A=rndi.rand(N,N).astype('float32')
b=rndi.rand(N).astype('float32')
B=np.dot(A,b)
X=la.solve(A,B)
t0=time();X=la.solve(A,B);t=time()-t0
print "time:",t                  # Solving time in single prec.
time: 1.39543216705              # using MKL on i7-6700

print(np.max(np.abs(X-b)))       # Max error
0.0048679709
print(X.dtype)
float32
                                 # direct lapack sgesv usage
t0=time();xxx=la.lapack.sgesv(A,B);t=time()-t0
print "time:",t
time: 1.3490459919

# continuation
N=8000                           # 8000x8000 double precision
A=rndi.rand(N,N)
b=rndi.rand(N)
B=np.dot(A,b)
X=la.solve(A,B)
t0=time();X=la.solve(A,B);t=time()-t0
print "time:",t                  # Solving time in double prec.
time:  2.51149082184             # using MKL on i7-6700

print(np.max(np.abs(X-b)))       # Max error
2.4676816146040e-11
```

```
print(X.dtype)
float64
                                  # direct lapack dgesv usage
t0=time();xxx=la.lapack.dgesv(A,B);t=time()-t0
print "time:",t
time: 2.46150708199

N=8000
A=rndi.rand(N,N)
b=rndi.rand(N)
B=np.dot(A,B)
lu,piv,info=la.lapack.dgetrf(A)
x,info=la.lapack.dgetrs(lu,piv,B)  # dgetrf+dgetrs
np.allclose(np.dot(A,x),B)
True
t0=time();lu,piv,info=la.lapack.dgetrf(A);  # new line for editing
          x,info=la.lapack.dgetrs(lu,piv,B);t=time()-t0 # purposes
print "time:",t
time: 2.446714077
```

1.9 Matrix inverse

ArrayFire and Numpy/Scipy allow for inverting of nonsingular, square matrices, i.e. for finding a matrix A^{-1} such that

$$A \cdot A^{-1} = I,$$

where I denotes the identity matrix. Below we show the functions `inverse` and `inv` in action.

```
##### ArrayFire #####

import arrayfire as af
A=af.randu(3,3)
# Inverse matrix
IA=af.inverse(A)
af.display(IA)

 3.6001     6.8885   -17.1199
-1.0316    -5.1806    12.2552
-1.0525    -0.4610     3.7828
```

```
##### Numpy/Scipy #####

import numpy as np
import scipy.linalg as la
a=np.random.rand(3,3)
ia=la.inv(a)
np.set_printoptions(precision=4)
print(ia)
[[-1.9555  0.817   2.1825]
 [-0.3387  1.3266 -0.6596]
 [ 2.8325 -1.1949 -1.2179]]
```

```
# Check if A*A^-1=I                # Check if a*a^-1=I
AIA=af.round(af.matmul(A,IA))      print(np.round(np.dot(a,ia)))
af.display(AIA)

 1.0000    -0.0000    -0.0000      [[ 1.  0. -0.]
-0.0000     1.0000    -0.0000       [ 0.  1.  0.]
 0.0000     0.0000     1.0000       [ 0.  0.  1.]]
```

In the next code example we give a comparison of Numpy/Scipy and ArrayFire speed in matrix inversion.

```
import arrayfire as af
from time import time
N=8000;A=af.randu(N,N)                  # 8000x8000 sing.prec.matrix
t0=time();IA=af.inverse(A);af.eval(IA);af.sync();t=time()-t0
print(t)                                # Inversion time in ArrayFire
0.203871011734                          # on GTX 1080

AIA=af.matmul(A,IA)                     # A*A^-1
I=af.identity(N,N)                      # I - identity
print af.max(af.abs(AIA-I))             # A*A^-1 -I
0.00331087806262                        # Max error

import numpy as np
from time import time
import scipy.linalg as la
import numpy.random_intel as rndi
a=rndi.rand(N,N).astype('float32')      # 8000x8000 sing.prec.matrix
t0=time();ia=la.inv(a);t=time()-t0
print(t)                                # Inversion time in Scipy
3.45366692543                           # on i7-6700 CPU

I=np.identity(N,'f')                    # I - identity
aia=np.dot(a,ia)                        # A*A^-1
print np.max(np.abs(aia-I))             # A*A^-1 -I
0.00164378                              # Max error
```

1.10 LU decomposition

The LU decomposition allows for representing matrix A as a product

$$A = PLU,$$

where P is a permutation matrix (square matrix obtained by a permutation of rows or columns of the identity matrix), L is a lower triangular matrix, and U is an upper triangular matrix. Using this decomposition one can replace one general linear system of equations by two easy to solve triangular systems.
Let us compare the ArrayFire and Scipy versions.

```
##### ArrayFire #####

import arrayfire as af
A=af.randu(3,3)
# LU factorization
L,U,P=af.lu(A)
# Lower triangular part L
af.display(L)
1.0000     0.0000     0.0000
0.6497     1.0000     0.0000
0.0992     0.9323     1.0000
# Upper triangular part U
af.display(U)
0.6051     0.2655     0.2200
0.0000     0.6088     0.2995
0.0000     0.0000     0.5520
# Permutation of rows
af.display(P)
1
0
2
```

```
##### Numpy/Scipy #####

import numpy as np
import scipy.linalg as la
a=np.random.rand(3,3)
p,l,u=la.lu(a)
# Lower triangular part l
print(l)
[[ 1.      0.      0.    ]
 [ 0.8988  1.      0.    ]
 [ 0.7187  0.0733  1.    ]]
# Upper triangular part U
print(u)
[[ 0.559   0.5673  0.3036]
 [ 0.      0.2112  0.5731]
 [ 0.      0.      0.5712]]
# Permutation matrix
print(p)
[[ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]]
```

Let us check the equality $A = PLU$.

```
##### ArrayFire #####

# LU = L*U
LU=af.matmul(L,U)
# P*A - LU
```

```
##### Numpy/Scipy #####

# lu=l*u
lu=np.dot(l,u)
plu=np.dot(p,lu)
```

```
print(af.max(af.abs(A[P]-LU)))          # p*l*u=a
0.0                                     np.allclose(plu,a)
                                        True
```

There are also `lu_inplace` and `lu_factor` versions of LU decomposition. In this version lower and upper triangles of A are replaced by L and U respectively.

```
##### ArrayFire #####                   ##### Numpy/Scipy #####

import arrayfire as af                  import numpy as np
A=af.randu(3,3)                         import scipy.linalg as la
A1=A.copy()                             a=np.random.rand(3,3)
# LU in ArrayFire, packed output        #
P=af.lu_inplace(A1,"full")              lu,piv=la.lu_factor(a)
# lower triangle                        # lower triangle
L=af.lower(A1,is_unit_diag=True)        l0=la.tril(lu)
# upper triangle                        l0[range(3),range(3)]=\
U=af.upper(A1)                          np.ones(3)
# LU=L*U                                u0=la.triu(lu) #upp.triang
LU=af.matmul(L,U)                       # lu0=l0*u0
# check if P*A=L*U                      lu0=np.dot(l0,u0)
print(af.max(af.abs(A[P]-LU)))          # P*A
5.960464477539e-08                      pa=la.lapack.slaswp(a,piv)
                                        print(np.allclose(lu0,pa))
                                        True
```

To check the ArrayFire efficiency in LU factorization one can use for example the following script.

```
import arrayfire as af
from time import time
N=8000
A=af.randu(N,N)                         # 8000x8000 random matrix
t0=time();LU=af.lu(A);af.eval(LU[0]);af.sync();t=time()-t0
print(t)
0.223619937897                          # LU decomp time on GTX 1080

# double precision version
A=af.randu(N,N,dtype=af.Dtype.f64)
t0=time();LU=af.lu(A);af.eval(LU[0]);af.sync();t=time()-t0
print(t)
1.43387985229
```

Compare the Scipy version in single precision:

```
from time import time
import numpy as np
import numpy.random_intel as rndi
import scipy.linalg as la
N=8000
a=rndi.rand(N,N).astype('f')
t0=time();lup=la.lu_factor(a);t=time()-t0
print(t)
1.4020049572
                                    # direct use of lapack sgetrf
t0=time();lupi=la.lapack.sgetrf(a);t=time()-t0
print(t)                            # time for single precision
1.35158014297                       # sgetrf on i7-6700 CPU
```

and in double precision:

```
from time import time
import numpy as np
import numpy.random_intel as rndi
import scipy.linalg as la
N=8000                              # double precision
a=rndi.rand(N,N)                    # 8000x8000 random matrix
t0=time();lup=la.lu_factor(a);t=time()-t0  #LU factorization
print(t)                            # time of LU factorization
2.48830580711                       # on i7-6700
print(lup[0].dtype)
float64                             # double precision
                                    # direct use of lapack dgetrf
t0=time();lupi=la.lapack.dgetrf(a);t=time()-t0
print(t)                            # time for lapack dgetrf
2.44885993004                       # on i7-6700 CPU
```

1.11 Cholesky decomposition

If a square matrix A is symmetric ($A^T = A$ or $A^H = A$) and positive definite ($x \cdot A \cdot x > 0$ for $x \neq 0$) then one can use a faster triangular decomposition

$$A = L \cdot L^T \text{ or } A = L^T \cdot L,$$

where L is a lower triangular matrix in the first formula and upper triangular in the second one.
In ArrayFire one can use `cholesky` function, which gives a factorization in the form $A = R^T \cdot R$ with upper triangular matrix R (if the lower triangular version is preferred then the command `af.cholesky(A,is_upper=False)` should be used). In Scipy we also use the version with upper triangular matrix R. If the lower triangular version is preferred, the command `cholesky(A,lower=True` or `cholesky(A,lower)` should be used.

```
##### ArayFire #####

import arrayfire as af
A=af.randu(3,3)
# make A symmetric
A=af.matmul(A.T,A)
# ArrayFire cholesky decomp.
R,info=af.cholesky(A)

af.display(R)
0.6423     0.6862     0.7595
0.0000     0.2282    -0.7275
0.0000     0.0000     0.2529
R1=af.matmul(R.T,R)
# check if R^T*R = A
print(af.max(af.abs(R1-A)))
0.0
```

```
##### Numpy/Scipy #####

import numpy as np
import scipy.linalg as la
A=np.random.rand(3,3)
A=np.dot(A.T,A)
# Scipy cholesky decomp.
R=la.cholesky(A)
np.set_printoptions(precision=4)
print(R)
[[ 1.2674  1.3576  1.0975]
 [ 0.      0.2956 -0.3014]
 [ 0.      0.      0.6136]]
# check if R^T*R = A
np.allclose(A,np.dot(R.T,R))
True
```

In Arrayfire there is also `cholesky_inplace` function, which replaces the upper triangle of A by R. In Scipy one can also use the `cho_factor` function which can overwrite A but by default the matrix is not overwritten. The full syntax is as follows: `cho_factor(a, lower=False, overwrite_a=False, check_finite=True)`.

```
##### ArrayFire #####

# continuation
A1=A.copy()
af.cholesky_inplace(A1)
R=af.upper(A1)
af.display(R)
```

```
##### Numpy/Scipy #####

# continuation
A1=A.copy()
B,lower=la.cho_factor(A1)
R=np.triu(B)
print(R)
```

```
0.6423     0.6862     0.7595     [[ 1.2674  1.3576  1.0975]
0.0000     0.2282    -0.7275      [ 0.      0.2956 -0.3014]
0.0000     0.0000     0.2529      [ 0.      0.      0.6136]]
R1=af.matmul(R.T,R)              print(np.allclose(A,np.dot(R.T,R)))
print(af.max(af.abs(R1-A)))      True
0.0
```

In Scipy one can also use lapack functions dpotrf, spotrf directly.

```
import numpy as np
from numpy.random import rand
import scipy.linalg as la
A=rand(3,3)
A=np.dot(A.T,A)
R,info=la.lapack.dpotrf(A)
# check if A=R^T*R
print(np.allclose(A,np.dot(R.T,R)))
True
```

Now let us check the efficiency of ArrayFire `cholesky` function.

```
import arrayfire as af
from time import time
N=10000;A=af.randu(N,N)                  # Rand. 10000x10000 matrix
A=af.matmul(A.T,A)                       # make A symmetric,
A=af.identity(N,N)*100+A                 # positive definite
t0=time();R,info=af.cholesky(A);af.eval(R);af.sync();t=time()-t0
                                         # Cholesky decomposition
print(t)                                 # in ArrayFire
0.164012908936                           # Decomp.time in ArrayFire
                                         # on GTX 1080
```

Compare it to Scipy version in double precision:

```
import numpy as np
from time import time
import numpy.random_intel as rndi
import scipy.linalg as la
N=10000;A=rndi.rand(N,N)                 # Rand. 10000x10000 matrix
A=np.dot(A.T,A)                          # make A symmetric
A=np.identity(N)*100+A                   # positive definite
t0=time(); R=la.cholesky(A);t=time()-t0
```

```
                                        # Cholesky decomposition
print(t)                                # in Scipy
2.65089011192                           # Decomp. time in Scipy
                                        # on i7-6700 CPU
print(L.dtype)                          # Double precision
float64

t0=time();res=la.lapack.dpotrf(A);t=time()-t0
print(t)                              # direct use of lapack dpotrf
2.36906909943
```

and next to Scipy single precision functions `cholesky` and `spotrf`:

```
import numpy as np
from time import time
import scipy.linalg as la
N=10000
A=rndi.rand(N,N).astype('float32')      # Random 10000x10000 matrix
A=np.dot(A.T,A)
A=A+np.eye(N).astype('float32')*30      # Symmetric, positive def.
t0=time(); R=la.cholesky(A);t=time()-t0  # Cholesky decomposition
print(t)                                # in single precision
1.22565197945                           # on i7-6700 CPU

t0=time();res=la.lapack.spotrf(A);t=time()-t0
print(t)                                # direct use lapack spotrf
1.14207792282
print(A.dtype)
float32
```

1.12 QR decomposition

A QR decomposition allows for representing matrix A as a product

$$A = Q \cdot R,$$

where Q is an orthogonal matrix (i.e. $Q^T \cdot Q = I$) and R is upper triangular matrix. The decomposition can be used to solve the linear least squares problem. It is also the basis for QR algorithm used in eigenvalues computations.
Here are the ArrayFire and Scipy versions of QR decomposition.

```
##### ArrayFire #####

import arrayfire as af
A=af.randu(3,3)
# QR decomposition
Q,R,T=af.qr(A)
# check if A =Q*R
QR=af.matmul(Q,R)
e=af.max(af.abs(QR-A))
print(e)
5.96046447754e-07

# check if Q is orthogonal
af.display(af.matmul(Q.T,Q),1)
    1.0    -0.0    -0.0
   -0.0     1.0     0.0
   -0.0     0.0     1.0
# check if R is triangular
af.display(R)
-0.6423    -0.6862    -0.7595
 0.0000     0.2282    -0.7275
 0.0000     0.0000     0.2529

A1=A.copy()
# in place QR (if Q is not essential)
T=af.qr_inplace(A1)
# compare with the previous
R1=af.upper(A1)
Q,R,T=af.qr(A)
af.max(af.abs(R1-R))
0.0
```

```
##### Numpy/Scipy #####

import numpy as np
import scipy.linalg as la
from numpy.random import rand
A=rand(3,3).astype('f')
Q,R=la.qr(A)
# check if A =Q*R
QR=np.dot(Q,R)
print(np.allclose(A,QR))
True

# check if Q is orthogonal
print(np.round(np.dot(Q.T,Q)))
[[ 1.  0.  0.]
 [ 0.  1. -0.]
 [ 0. -0.  1.]]
np.set_printoptions(precision=4)
print(R)
[[-1.0441, -1.1920, -0.2727],
 [ 0.    , -0.6809, -0.2753],
 [ 0.    ,  0.    ,  0.2630]]

A1=A.copy()
R1=la.qr(A,mode='r')
# compare with the previous
print(np.allclose(R,R1))
True
```

In Scipy we have an additional versions of QR decomposition with pivoting.

```
# continuation
Q,R,P=la.qr(A,pivoting=True)         # QR with pivoting: A*P=Q*R
np.allclose(A[:,P],np.dot(Q,R))      # chck if P*A=Q*R
True

qr,tau,work,info=la.lapack.sgeqrf(A) # direct use of lapack sgeqrf
qqr=np.empty((3,3))
qqr[:,:3]=qr
```

```
q,work,info=la.lapack.sorgqr(qqr,tau)# obtain q from qr
r=np.triu(qr)                        # r is in upper triangle
np.allclose(A,np.dot(q,r))           # check if A=q*r
True
```

Let us compare the efficiency of Scipy and ArrayFire in QR decomposition. The following script shows the ArrayFire version.

```
import arrayfire as af
from time import time                # single precision
N=5000;A=af.randu(N,N)               # 5000x5000 random array
t0=time();Q,R,T=af.qr(A);af.eval(R);af.sync();t=time()-t0
print(t)
3.54815292358                        # QR decomp. time on GTX 1080
```

Here is the Scipy version for comparison.

```
import numpy as np
from time import time
import numpy.random_intel as rndi
import scipy.linalg as la            # single precision
N=5000;A=rndi.rand(N,N).astype('f')  # 5000x5000 random array
t0=time();Q,R=la.qr(A);t=time()-t0   # QR decomposition
print(t)
1.35702681541                        # QR decomp. time on i7-6700
                                     # Direct use of lapack sgeqrf
t0=time();qr,tau,work,info=la.lapack.sgeqrf(A);t=time()-t0
print(t)
0.648733854294                       # lapack sgeqrf time on i7-6700
```

1.13 Singular Value Decomposition

Singular value decomposition (SVD) is the most general matrix decomposition which works even for non-square and singular matrices. For $m \times n$ matrix A it has the form

$$A = U \cdot S \cdot V,$$

where U, V are orthogonal matrices of dimension $m \times m$ and $n \times n$ respectively. The $m \times n$ matrix S is diagonal and has only positive or zero elements on its diagonal (the singular values of A).

Let us show how this decomposition works in ArrayFire and Scipy.

```
##### ArrayFire #####

import arrayfire as af
A=af.randu(4,3)
# SVD in ArrayFire
U,s,Vt = af.svd(A)

# vector of sing. vals
af.display(s)
    1.5198
    0.8578
    0.2329
# matrix with sing. vals
S=af.diag(s,0,False)
S=af.join(0,S,af.constant(0,1,3))
af.display(S)
1.5198     0.0000     0.0000
0.0000     0.8578     0.0000
0.0000     0.0000     0.2329
0.0000     0.0000     0.0000

# check if A=U*S*Vt
USVt=af.matmul(af.matmul(U,S),Vt)
print(af.max(af.abs(USVt-A)))
1.19209289551e-07

# check if U is orthogonal
af.display(af.matmul(U.T,U),0)
 1     -0      0      0
-0      1     -0      0
 0     -0      1      0
 0      0      0      1
# check if Vt is orthogonal
af.display(af.matmul(Vt,Vt.T),0)
 1      0      0
 0      1     -0
 0     -0      1
```

```
##### Numpy/Scipy #####

import numpy as np
from numpy.random import rand
import scipy.linalg as la
A=rand(4,3).astype('f')
U,s,Vt=la.svd(A)
np.set_printoptions(4)
print(s)
[ 1.8963  0.8393  0.1755]



S=la.diagsvd(s,4,3)

print(S)
[[ 1.8963  0.      0.    ]
 [ 0.      0.8393  0.    ]
 [ 0.      0.      0.1755]
 [ 0.      0.      0.    ]]

SVt=np.dot(S,Vt)
q=np.allclose(A,np.dot(U,SVt))
print(q)
True


print(np.round(np.dot(U,U.T)))
[[ 1., -0.,  0.,  0.],
 [-0.,  1.,  0.,  0.],
 [ 0.,  0.,  1.,  0.],
 [ 0.,  0.,  0.,  1.]]

print(np.round(np.dot(Vt,Vt.T)))
[[ 1., -0.,  0.],
 [-0.,  1., -0.],
 [ 0., -0.,  1.]]
```

There are also `inplace` versions of SVD, where the matrix A may be overwritten.

```
##### ArrayFire #####

# Arrayfire in place version,
# A overwritten
import arrayfire as af
A=af.randu(3,3)
A1=A.copy()
U,s,Vt=af.svd_inplace(A)
S=af.diag(s,0,False)
# check the sing.vals
af.display(s)
    2.5247
    0.5302
    0.3873
# compute U*S*Vt
A0=af.matmul(af.matmul(U,S),Vt)
# check svd error
print(af.max(af.abs(A0-A1)))
3.57627868652e-07
```

```
##### Numpy/Scipy #####

import numpy as np
import scipy.linalg as la
from numpy.random import rand
np.set_printoptions(4)
A=rand(3,4).astype('f')
U,s,Vt=la.svd(A,overwrite_a=True)
U1,s1,Vt1=la.svd(A)
# check the sing.vals
print(s)
[ 1.8828  0.2793  0.1628]

print(s1)
[ 1.8828  0.2793  0.1628]
```

The efficiency of ArrayFire SVD can be checked using the script:

```
import arrayfire as af
from time import time
A=af.randu(2000,2000)              # 2000x2000 ArrayFire matrix
t0=time();U,S,Vt=af.svd(A);af.eval(S);af.sync();t=time()-t0
print(t)                           # Time for SVD decomp.
11.8229048252                      # on GTX 1080, AF v3.5
#3.80988192558                     # GTX 750, AF, v3.3.2
```

Below we present a corresponding Scipy test.

```
import numpy as np
from time import time
import numpy.random_intel as rndi
import scipy.linalg as la
A=rndi.rand(2000,2000).astype('f') # 2000x2000 single prec. matr.
t0=time();U,S,Vt=la.svd(A);t=time()-t0
print(t)                           # Scipy SVD decomp. on i7-6700
1.44229793549
```

```
# direct use of lapack
lwork=\
la.lapack._compute_lwork(la.lapack.sgesdd_lwork,A.shape[0],A.shape[1])
t0=time();U,S,Vt,info=la.lapack.sgesdd(A);t=time()-t0
print(t)
1.323127985                          # Scipy lapack SVD decomp.
```

1.14 Plotting with ArrayFire

1.14.1 Two-dimensional plot

To obtain a two-dimensional plot in ArrrayFire-Python one can use the function `plot`. For example the sine function can be plotted using the following python script:

```
import arrayfire as af
import math
N = 1000
X = math.pi*(2*(af.range(N+1)/N)-1) # X=pi*(-1,-1+2/N,-1+4/N,...,1)
win = af.Window(1024, 512, "Sine function")
while not win.close():
    Y = af.sin(X)                    # Y=sin(X)
    win.plot(X, Y)                   # Two-dimensional plot
```

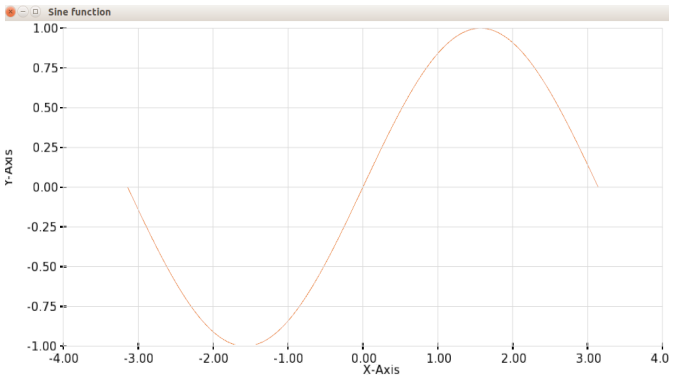

Figure 1.1: Plot of sine function in ArrayFire

1.14.2 Surface plot

ArrayFire-Python allows also for three-dimensional plots. The corresponding function is `surface`. For example let us show how to plot

$$f(x, y) = \cos(\sqrt{x^2 + y^2}).$$

```
import arrayfire as af
import math
N2 = 60
N = 2 * N2
# grid x,y=2pi*(-1,-1+2/N,-1+4/N,...,1-2/N,1)
x =2*math.pi*(af.iota(d0=N,d1=1,tile_dims=(1,N))-N2)/N2
y =2*math.pi*(af.iota(d0=1,d1=N,tile_dims=(N,1))-N2)/N2
win = af.Window(1200, 600, "3D Surface")
while not win.close():
    z=af.cos(af.sqrt(x*x+y*y))
    win.surface(x, y, z)              # Three-dimensional plot
```

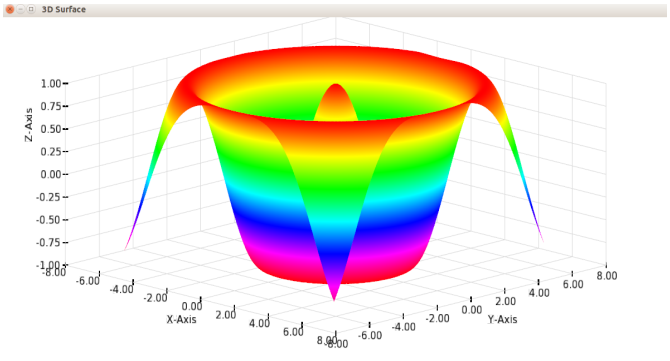

Figure 1.2: Three-dimensional plot in ArrayFire

1.14.3 Image-plot

There is also `image` function in ArrayFire. Let us show how to plot the function from the previous example using `image`.

```
import arrayfire as af
import math
N2 = 120
```

```
N = 2 * N2
# grid x,y=2pi*(-1,-1+2/N,-1+4/N,...,1-2/N,1)
x =2*math.pi*(af.iota(d0=N+1,d1=1,tile_dims=(1,N+1))-N2)/N2
y =2*math.pi*(af.iota(d0=1,d1=N+1,tile_dims=(N+1,1))-N2)/N2
win = af.Window(600, 600, "image")
win.set_colormap(af.COLORMAP.SPECTRUM)
while not win.close():
    z=af.cos(af.sqrt(x*x+y*y))
    win.image(z)                    # image plot
```

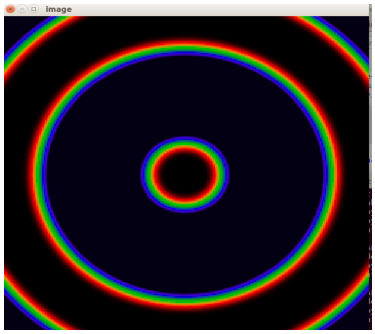

Figure 1.3: Image-plot in ArrayFire

Chapter 2

ArrayFire-C/C++

2.1 Introductory remarks

As we have mentioned in the foreword, the Nvidia CUDA and OpenCL platforms allow for solving many computational problems in an efficient, parallel manner. Both platforms can be considered as extensions of C/C++ language, therefore they need a significant experience in low-level C/C++ programming. The purpose of ArrayFire is to prepare the corresponding high-level programming environment. The aim of ArrayFire C++ interface is to allow the users to solve specific computational problems in few lines of C/C++ code achieving good computational efficiency.

Remark. The most complete description of ArrayFire C/C++ interface can be found on `arrayfire.org/docs/index.htm` or in the arrayfire documentation directory `arrayfire-3/share/ArrayFire/doc/html/index.htm`.

Let us begin with a short program showing some details concerning the ArrayFire version and hardware installed.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  info();                              // Information on installed
  return 0;                            // software and hardware
}
```

```
/*
ArrayFire v3.5.0 (CUDA, 64-bit Linux, build 05999f3)
Platform: CUDA Toolkit 8, Driver: 378.13
[0] GeForce GTX 1080, 8111 MB, CUDA Compute 6.1
*/
```

The pages

- `arrayfire.org/docs/using_on_linux.htm`,
- `arrayfire.org/docs/using_on_windows.htm`,
- `arrayfire.org/docs/using_on_osx.htm`

contain instructions how to compile the arrayfire code in preferred operating systems.

2.2 Defining arrays

It seems that the simplest way to define an ArrayFire array is to use `constant` or `identity` functions.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  array in0=constant(0,3,3);        // float zeros
  af_print(in0);
  array in1=constant(1,3,3);        // float ones
  af_print(in1);
  array in2=identity(3,3);          // float identity
  af_print(in2);
  af_print(constant(1,3,3,c32),1);  // complex ones
  af_print(constant(1,3,3,u32));    // integer ones
  return 0;
}
/*
[3 3 1 1]
    0.0000     0.0000     0.0000
    0.0000     0.0000     0.0000
    0.0000     0.0000     0.0000
```

```
in1
[3 3 1 1]
    1.0000     1.0000     1.0000
    1.0000     1.0000     1.0000
    1.0000     1.0000     1.0000

in2
[3 3 1 1]
    1.0000     0.0000     0.0000
    0.0000     1.0000     0.0000
    0.0000     0.0000     1.0000

constant(1,3,3,c32)
[3 3 1 1]
      (1.0,0.0)       (1.0,0.0)       (1.0,0.0)
      (1.0,0.0)       (1.0,0.0)       (1.0,0.0)
      (1.0,0.0)       (1.0,0.0)       (1.0,0.0)

constant(1,3,3,u32)
[3 3 1 1]
         1          1          1
         1          1          1
         1          1          1
*/
```

The default type is `f32` i.e. `float`. The basic types have the abbreviations:

- `f32` –real single-precision (float)
- `c32` –complex single-precision (cfloat)
- `f64` –real double-precision (double)
- `c64` –complex double-precision (cdouble)
- `b8` –8-bit boolean values (bool)
- `s32` –32-bit signed integer (int)
- `u32` –32-bit unsigned integer (unsigned)
- `u8` –8-bit unsigned values (unsigned char)

- `s64` –64-bit signed integer (intl)
- `u64` –64-bit unsigned integer (uintl)
- `s16` –16-bit signed integer (short)
- `u16` –16-bit unsigned integer (unsigned short)

Very often an array is defined on the host. Sometimes we want to create its copy on the device. In this case we can use the `cudaMalloc` and `cudaMemcpy` functions.

```
#include <stdio.h>
#include <arrayfire.h>
#include <af/cuda.h>
using namespace af;
int main(void){
  float host_ptr[] = {0,1,2,3,4,5,6,7,8};
  array a(3, 3, host_ptr); // 3x3 f32 matrix  from host pointer
  af_print(a);
  float *device_ptr;
  cudaMalloc((void**)&device_ptr, 9*sizeof(float));
  cudaMemcpy(device_ptr, host_ptr, 9*sizeof(float),
                           cudaMemcpyHostToDevice);
  array b( 3,3,device_ptr, afDevice); // 3x3 f32 matrix
  af_print(b);                        // from device pointer
  return 0;
}
/*
a
[3 3 1 1]                             // from host
    0.0     3.0     6.0
    1.0     4.0     7.0
    2.0     5.0     8.0

b                                     // from device
[3 3 1 1]
    0.0     3.0     6.0
    1.0     4.0     7.0
    2.0     5.0     8.0
*/
```

The last array can be also defined on the device using the sequence operation `seq` and the `moddims` command.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  array in=seq(0,8);
  af_print(in,1);
  array b=moddims(in,3,3);           // b as 3x3 matrix
  af_print(b,1);
  return 0;
}
/*
in                                   // in as a column
    0.0
    1.0
    2.0
    3.0
    4.0
    5.0
    6.0
    7.0
    8.0
b                                    // b as a 3x3 matrix
    0.0     3.0     6.0
    1.0     4.0     7.0
    2.0     5.0     8.0
*/
```

If the array elements are given by a formula we can use the following modification.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  float *host_ptr=new float[3*3];
  for(int i=0;i<3;i++)
    for(int j=0;j<3;j++)
      host_ptr[3*i+j]=i-j;
```

```
  array a(3, 3, host_ptr); // 3x3 f32 matrix  from host pointer
  af_print(a,a);
  delete [] host_ptr;
  return 0;
}
/*
a                                        // From host pointer
    0.0     1.0     2.0
   -1.0     0.0     1.0
   -2.0    -1.0     0.0
*/
```

Here is an example with complex entries.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  array a=moddims(seq(0,8),3,3);
  array b=moddims(seq(9,17),3,3);
  array c=complex(a,b);                  // c=a+b*i
  af_print(c,1);
  float hA[]={0,1,2,3,4,5};
  array dB(3,1,(cfloat*) hA);            // 0,2,4 - real parts
  af_print(dB,1);                        // 1,3,5 - imaginary parts
  return 0;
}
/*
c
[3 3 1 1]
      (0.0,9.0)        (3.0,12.0)       (6.0,15.0)
      (1.0,10.0)       (4.0,13.0)       (7.0,16.0)
      (2.0,11.0)       (5.0,14.0)       (8.0,17.0)


dB
[3 1 1 1]
      (0.0,1.0)
      (2.0,3.0)
      (4.0,5.0)
*/
```

2.3 Random arrays

ArrayFire allows for an easy and efficient generation of random arrays from uniform and normal distributions.
Let us begin with small matrices.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  int n = 3;
  array A = randu(n, n);              // uniformly distributed
  printf("uniform:\n");               // f32 3x3 random matrix
  af_print(A);
  A = randn(n, n, c32);
  printf("normal,complex:\n");        // normally distributed
  af_print(A);                        // compl. (c32) 3x3 matr.
  return 0;
}
/*
uniform:
A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
normal,complex:
A
[3 3 1 1]
   (-0.1370,-1.2768)   (-1.0515,-0.5493)    (1.1401,0.7409)
   (1.0782,-0.6237)    (2.0714,1.2586)      (-0.0625,0.5143)
   (1.5255,0.6660)     (1.4206,0.3912)      (0.5286,-1.0861)
*/
```

Using `floor, ceil` or `round` functions one can obtain random matrices with integer entries.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  array A = 5*randn(3, 3);        // random f32 3x3 matrix
```

```
  af_print(A)                     // A
  af_print(floor(A),0);           // apply floor
  af_print(ceil(A),0);            // apply ceil
  af_print(round(A),0);           // apply round
  return 0;
}
/*
A
[3 3 1 1]
   -4.6233     8.2411     5.9010
   -3.1699     2.3200     1.4000
   -0.9123     5.2804     1.0730

floor(A)
[3 3 1 1]
    -5      8      5
    -4      2      1
    -1      5      1

ceil(A)
[3 3 1 1]
    -4      9      6
    -3      3      2
    -0      6      2

round(A)
[3 3 1 1]
    -5      8      6
    -3      2      1
    -1      5      1
*/
```

Using 8000x8000 random matrices we can check the efficiency of GPU random generators.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  int n = 8e3;
  array A = randu(n, n);      // Warm-up
```

```
  for(int k=0;k<3;k++){
    timer::start();
    A = randu(n, n);          // 8000x8000 uniform random matrix
    af::sync();
    printf("uniform: %g\n", timer::stop());
  }
  A = randn(n, n);            // Warm-up
  for(int k=0;k<3;k++){
    timer::start();
    A = randn(n, n);          // 8000x8000 normal random matrix
    af::sync();
    printf("normal: %g\n", timer::stop());
  }
  return 0;
}
/*
uniform: 0.002012
uniform: 0.001005
uniform: 0.001007
normal: 0.002256
normal: 0.001135
normal: 0.001137
*/
```

2.4 Rearranging arrays

The operation of transposition, conjugation and conjugate transposition have the following realizations.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3, 3);
  af_print(A);
  af_print(A.T());                        // Transposition
  A=randu(3,3,c64);
  af_print(A);
  af_print(conjg(A));                     // Conjugation
  af_print(A.H());                        // Conjugate transp.
  return 0;
```

```
}
/*
A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702

A.T()
[3 3 1 1]
    0.7402     0.9210     0.0390
    0.9690     0.9251     0.4464
    0.6673     0.1099     0.4702

A
[3 3 1 1]
   (0.5170,0.0714)    (0.6734,0.1202)    (0.1583,0.7056)
   (0.3335,0.4896)    (0.1631,0.5666)    (0.9808,0.1623)
   (0.1845,0.5206)    (0.2072,0.4028)    (0.4279,0.7804)

conjg(A)
[3 3 1 1]
   (0.5170,-0.0714)    (0.6734,-0.1202)    (0.1583,-0.7056)
   (0.3335,-0.4896)    (0.1631,-0.5666)    (0.9808,-0.1623)
   (0.1845,-0.5206)    (0.2072,-0.4028)    (0.4279,-0.7804)

A.H()
[3 3 1 1]
   (0.5170,-0.0714)    (0.3335,-0.4896)    (0.1845,-0.5206)
   (0.6734,-0.1202)    (0.1631,-0.5666)    (0.2072,-0.4028)
   (0.1583,-0.7056)    (0.9808,-0.1623)    (0.4279,-0.7804)
*/
```

One can flip the array horizontally or vertically.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3, 3);
  af_print(A);
  af_print(flip(A,0));                      // Flip horizontally
```

```
  af_print(flip(A,1));                        // Flip vertically
  return 0;
}

/*
A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
flip(A,0)
[3 3 1 1]
    0.0390     0.4464     0.4702
    0.9210     0.9251     0.1099
    0.7402     0.9690     0.6673
flip(A,1)
[3 3 1 1]
    0.6673     0.9690     0.7402
    0.1099     0.9251     0.9210
    0.4702     0.4464     0.0390
*/
```

The array can be also flattened and upper and lower triangular part can be extracted.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3, 3);
  af_print(A);
  af_print(flat(A));                          // Flattened A
  af_print(lower(A));                         // Lower triang. part
  af_print(upper(A));                         // Upper triang. part
  return 0;
}

/*
A
[3 3 1 1]
```

```
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702

flat(A)
[9 1 1 1]
    0.7402
    0.9210
    0.0390
    0.9690
    0.9251
    0.4464
    0.6673
    0.1099
    0.4702

lower(A)
[3 3 1 1]
    0.7402     0.0000     0.0000
    0.9210     0.9251     0.0000
    0.0390     0.4464     0.4702

upper(A)
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.0000     0.9251     0.1099
    0.0000     0.0000     0.4702
*/
```

There are also `shift` and `rotate` operations.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=seq(0,8);                          // A: 0,1,...,8
  af_print(A.T(),1);
  af_print(shift(A,1).T(),1);                // Shift operation
  af_print(rotate(A,3).T(),1);               // Rotate operation

  return 0;
}
```

```
/*
A.T()
0      1      2      3      4      5      6      7      8

shift(A,1).T()
8      0      1      2      3      4      5      6      7


rotate(A,3).T()
0      7      6      5      4      3      2      1      0
*/
```

It is possible to join two matrices into one larger matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A1=seq(1,3);                  // A1: [1,2,3]^T
  array A2=seq(4,6);                  // A2: [4,5,6]^T
  af_print(join(0,A1,A2),0);          // Join vertically
  af_print(join(1,A1,A2),0);          // Join horizontally
  A1=constant(1,3,3);                 // A1: 3x3 matrix of ones
  A2=constant(0,3,3);                 // A2: 3x3 matrix of zeros
  af_print(join(0,A1,A2),0);          // Join vertically
  af_print(join(1,A1,A2),0);          // Join horizontally
  return 0;
}
/*
join(0,A1,A2)                         // Join vertically
           1
           2
           3
           4
           5
           6

join(1,A1,A2)                         // Join horizontally
           1        4
           2        5
           3        6
```

```
join(0,A1,A2)                          // Join vertically
           1        1        1
           1        1        1
           1        1        1
           0        0        0
           0        0        0
           0        0        0

join(1,A1,A2)                           // Join horizontally
     1   1   1   0   0   0
     1   1   1   0   0   0
     1   1   1   0   0   0
*/
```

2.5 Determinant, norm and rank

The ArrayFire function `det` allows for computing determinants.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 3;
  array A=randu(n,n);
  float d = det<float>(A);                  // Determinant of A
  printf("determinant: %f\n",d);
  return 0;
}
//determinant: 0.120453                     // The value of det(A)
```

In the following example we check the `det` function in the case of the upper triangular 5000x5000 matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 5e3;
  array A=upper(constant(1,n,n,f32));    // A: 5000x5000 upper
                                         // Triang. matrix of ones
```

```
  float d = det<float>(A);
  timer::start();
  d = det<float>(A);                    // Determinant of A
  af::sync();
  printf("det time: %g\n", timer::stop());
  printf("%f\n",d);
  return 0;
}
//det time: 0.06443
//1.000000                              // Value of det
```

In the `norm` function we can use an additional parameter to specify what kind of norm we have in mind:

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=seq(1,4);
  af_print(A);
  printf("%lf\n",norm(A));                      //euclid
  printf("%g\n",norm(A,AF_NORM_EUCLID));        //euclid
  printf("%g\n",norm(A,AF_NORM_VECTOR_1));      //L_1
  printf("%g\n",norm(A,AF_NORM_VECTOR_2));      //L_2
  printf("%g\n",norm(A,AF_NORM_VECTOR_P,4));    //L_4
  printf("%g\n",norm(A,AF_NORM_VECTOR_INF));    //L_inf
return 0;
}
/*
A
[4 1 1 1]
    1.0000
    2.0000
    3.0000
    4.0000

5.477226                                //euclid. norm of A
5.47723                                 //euclid. norm of A
10                                      //L_1 norm of A
5.47723                                 //L_2 norm of A
4.33761                                 //L_4 norm of A
```

```
4                                       //L_inf norm of A
*/
```

In ArrayFire one can also find the function `rank` which computes the number of linearly independent rows or columns of an array.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3,4);                   // A: random 3x4 matrix
  unsigned r=rank(A);                   // Rank of A
  printf("%d\n",r);
  return 0;
}
//3                                     // Rank of A
```

In some single precision calculations the default tolerance 1e-5 (which means that only the singular values greater than this number are considered) should be changed.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(600,400);             // A: 600x400 random matrix
  A(seq(300),span)=0;                 // First 300 rows set to 0
  unsigned r=rank(A,1e-3);            // Rank of A, tolerance 1e-3
  printf("%d\n",r);
  return 0;
}
//300                                 // Rank of A
```

2.6 Elementary arithmetic operations on matrices

In ArrayFire the sum, difference and the product of two matrices one can obtain using the +, – and `matmul` operators.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
```

```
  array A=moddims(seq(0,8),3,3);
  array B=array(seq(1,9),3,3);
  af_print(A,0);
  af_print(B,0);
  af_print(A+B,0);
  af_print(A-B,0);
  af_print(matmul(A,B),0);
  return 0;
}
/*
A
[3 3 1 1]
     0      3      6
     1      4      7
     2      5      8
B
[3 3 1 1]
     1      4      7
     2      5      8
     3      6      9
A+B
[3 3 1 1]
     1      7     13
     3      9     15
     5     11     17
A-B
[3 3 1 1]
    -1     -1     -1
    -1     -1     -1
    -1     -1     -1
matmul(A,B)
[3 3 1 1]
    24     51     78
    30     66    102
    36     81    126
*/
```

The * operator gives the element-wise product.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
```

```
int main(){
  array A=seq(0,8);                    // A: 0,...,8
  array B=moddims(A,3,3);              // B: A as 3x3 matrix
  af_print(B,0);
  af_print(B*B,0);                     // Element-wise product
  return 0;
}
/*
B                                      // B
     0      3      6
     1      4      7
     2      5      8

B*B                                    // Element-wise prod. B times B
     0      9     36
     1     16     49
     4     25     64
*/
```

The elementwise matrix power $A * \ldots * A$ (n-times) can be obtained using `pow` function

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=moddims(seq(0,8),3,3);       // A: 0,...,8 as 3x3 matrix
  af_print(A,0);
  af_print(A*A,0);                     // A*A (for comparison)
  af_print(af::pow(A,2),0);            // elementwise A^2
  af_print(af::pow(A,4),0);            // elementwise A^4
  return 0;
}
/*
A
[3 3 1 1]
     0      3      6
     1      4      7
     2      5      8
```

```
A*A
[3 3 1 1]
     0      9     36
     1     16     49
     4     25     64

af::pow(A,2)
[3 3 1 1]
     0      9     36
     1     16     49
     4     25     64

af::pow(A,4)
[3 3 1 1]
     0     81   1296
     1    256   2401
    16    625   4096
*/
```

The inputs for `pow` function can be two arrays or an array and a scalar.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=array(seq(0,8),3,3);
  af_print(A);
  af_print(pow(A,2.0f),0);               // scalar exponent
  af_print(pow(A,constant(2,3,3)),0);   // matrix exponent
  return 0;
}
/*
A
[3 3 1 1]
     0      3      6
     1      4      7
     2      5      8
pow(A,2.0f)                             // scalar exponent
[3 3 1 1]
     0      9     36
     1     16     49
     4     25     64
```

```
pow(A,constant(2,3,3))                  // matrix exponent
[3 3 1 1]
     0      9     36
     1     16     49
     4     25     64
*/
```

The efficiency of the matrix multiplication in ArrayFire C/C++ interface can be checked using the following code.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  int n = 8e3;
  array A = randu(n,n);                 // A,B 8000x8000 matrices
  array B = randu(n,n);
  for(int k=0;k<3;k++)
  {
    timer::start();
    array C = matmul(A,B);              // Matrix multiplication
    af::sync();
    printf("multiplication time: %g\n", timer::stop());
  }
  return 0;
}
/*
multiplication time: 0.2459
multiplication time: 0.13868
multiplication time: 0.138798
*/
```

2.7 Sums and products of elements

Using the functions `sum` and `prod` we can sum or multiply all entries of an array, or elements of some subsets, for example of rows or columns.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
 array A=constant(2,3,3);
  float su,pro;
  af_print(A,0);                                  // A
  af_print(sum(A),0);                             // Sums of columns
  af_print(sum(A,1),0);                           // Sums of rows
  su = sum<float>(A);                             // Sum of all elem.
  printf("sum= %f\n",su);
  af_print(product(A),0);                         // Products of columns
  af_print(product(A,1),0);                       // Products of rows
  pro=product<float>(A);                          // Product of all elem.
  printf("product= %f\n",pro);
  return 0;
}
/*
A                                                 // A
[3 3 1 1]
     2      2      2
     2      2      2
     2      2      2

sum(A)                                            // Sums of columns
[1 3 1 1]
     6      6      6

sum(A,1)                                          // Sums of rows
[3 1 1 1]
     6
     6
     6

sum= 18.000000                                    // Sum of all elements

product(A)                                        // Products of columns
[1 3 1 1]
     8      8      8

product(A,1)                                      // Products of rows
```

```
[3 1 1 1]
     8
     8
     8

product= 512.000000                    // Product of all elements
*/
```

To check the Euler's formula

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

one can perform the following calculations.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  long N=1e7;
  float su;
  array A=seq(1,N);               // A: 1,2,...,10^7
  A=A*A;                          // A: 1^2,2^2,...,(10^7)^2
  su = sum<float>(1.0/A);         // 1/1^2+1/2^2+...+1/(10^7)^2
  printf("sum= %f\n",su);
  return 0;
}
//sum= 1.644934                   // pi^2/6  approx.: 1.6449340668
```

2.8 Dot product

In ArrayFire there is a specialized `dot` function computing the dot product of two arrays.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  unsigned m = 3;
  array A1 = constant(1,m);              // A1: 1,1,1
```

```
  array A2 = seq(m);                    // A2: 0,1,2
  array inn = dot(A1,A2);               // Dot product of A1,A2
  af_print(A1,0);
  af_print(A2,0);
  printf("dot product:\n");
  af_print(inn,0);
  return 0;
}

/*
A1                                      // A1
[3 1 1 1]
     1
     1
     1

A2                                      // A2
[3 1 1 1]
     0
     1
     2

dot product:
inn
[1 1 1 1]
     3                                  // Inner product of A1,A2
*/
```

Now let us try to compute the dot product of A times A for 5000x1 matrix of ones.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  unsigned m = 5e3;
  array A1 = constant(1,m,1);           // 5000x1 matrix of ones
  array inn = dot(A1,A1);               // warm-up
  timer::start();
  inn = dot(A1,A1);                     // dot product A1 and A1
  printf("dot prod. time: %g\n", timer::stop());
```

```
  printf("dot prod. value: \n");
  af_print(inn);
  return 0;
}
/*
dot prod. time: 4.1e-05
dot prod. value:
inn
[1 1 1 1]
 5000.0000
*/
```

2.9 Mean, variance and standard deviation

The C/C++ interface to ArrayFire allows for efficient computations of average, variance and standard deviation for arrays.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=array(seq(0,8),3,3);      // A: 0,...,8 as 3x3 matrix
  af_print(A);
  af_print(mean(A));                    // Averages of columns
  af_print(mean(A,1));                  // Averages of rows
  printf("mean: %f\n",mean<float>(A));   // Average of A
  af_print(var(A));                     // Variances of columns
  af_print(var(A,0,1));                 // Variances of rows
  printf("var: %f\n",var<float>(A));     // Variance of A
  af_print(stdev(A));               // Stand. deviations of columns
  af_print(stdev(A,1));                // Stand. deviations of rows
  printf("stdev: %f\n",stdev<float>(A));  // Stand. deviation of A
  return 0;
}
/*
A                                          // A
[3 3 1 1]
    0.0000     3.0000     6.0000
    1.0000     4.0000     7.0000
    2.0000     5.0000     8.0000
```

```
mean(A)
[1 3 1 1]
    1.0000     4.0000     7.0000          // Averages of columns

mean(A,1)
[3 1 1 1]
    3.0000
    4.0000                                // Averages of row
    5.0000

mean: 4.000000                            // Average of A
var(A)
[1 3 1 1]
    1.0000     1.0000     1.0000          // Variances of columns

var(A,0,1)
[3 1 1 1]
    9.0000
    9.0000                                // Variances of rows
    9.0000

var: 7.500000                             // Variance of A

stdev(A)
[1 3 1 1]
    0.8165     0.8165     0.8165          // Std. dev. of columns

stdev(A,1)
[3 1 1 1]
    2.4495
    2.4495                                // Std. dev. of rows
    2.4495

stdev: 2.581989                           // Std. dev. of A
*/
```

Using larger arrays we can check the efficiency of ArrayFire.

```
#include <stdio.h>
#include <arrayfire.h>
```

```
using namespace af;
int main(void){
  int n = 8e3;
  float a,v;
  array A = randn(n, n,f32);               // A: 8000x8000 matrix
  for(int k=0;k<3;k++){
    timer::start();
    a=mean<float>(A);                      // Mean of A
    v=var<float>(A);                       // Variance of A
    array hi=histogram(A,100);             // Histogram of A
    af::sync();
    printf(" time: %g\n", timer::stop());
    printf(" mean(A): %g\n", a);
    printf(" var(A): %g\n", v);
  }
//  af_print(hi);             // Redirect to a file and use gnuplot
  return 0;     // or use the ArrayFire graphics (next code sample)
}
/*
time: 0.134017                          // Time for mean, var, hist.
mean(A): -0.000159369                   // Theoretical mean: 0
var(A): 0.999838                        // Theoretical variance: 1
time: 0.008637
mean(A): -0.000325331
var(A): 1.00042
time: 0.00914
mean(A): -0.000325331
var(A): 1.00042
*/
```

ArrayFire includes also graphical tools allowing to visualize histograms.

```
#include <arrayfire.h>
using namespace af;
int main(int argc, char *argv[])
{
    af::Window myWindow(1024, 512, "Histogram");
    int n=1000;
    array hi=histogram(A,100);
    while (!myWindow.close()) {
        myWindow.hist(hi, 0, 100);
    }
```

```
return 0;
}
```

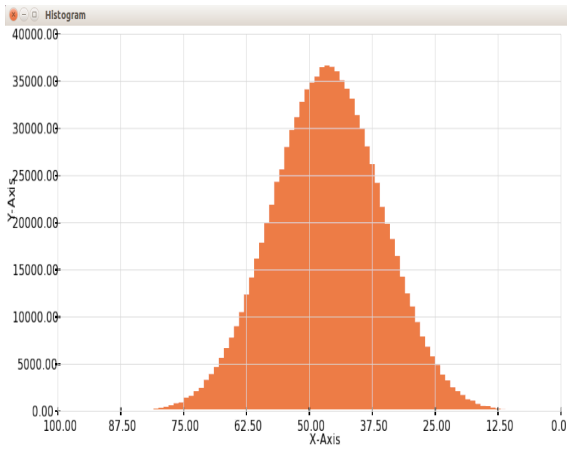

Figure 2.0: Histogram in ArrayFire

2.10 Solving linear systems

Systems of the form

$$Ax = B,$$

where A, B are ArrayFire arrays can be efficiently solved, using the `solve` function.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
    int n = 3;
    array A = randu(n,n);                // A: 3x3 random matrix
    array B = randu(n,1);                // B: 3x1 random vector
    array X = solve(A, B);               // Solving A*X=B
    printf("%g\n",norm(matmul(A,X) - B,AF_NORM_VECTOR_INF)); //L_inf
    return 0;                            // norm of the error
}

//5.96046e-08                           // L_inf error
```

Using ArrayFire C/C++ we can solve systems with thousands of equations in a fraction of a second.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
   int n = 8e3;
   array A = randu(n, n);              // A: 8000x8000 matrix
   array B = randu(n,1);               // B: 8000x1 vector
   for(int k=0;k<3;k++){
      timer::start();
      array X = solve(A, B);           // Solving A*X=B
      af::sync();
      printf("solving time: %g\n", timer::stop());
      printf("%g\n",norm(matmul(A,X)-B,AF_NORM_VECTOR_INF));//error
  }
  return 0;
}
solving time: 0.29151                 // Solving time
0.0141194                    // Inf norm of A*X-B (single precision)
solving time: 0.192112
0.0141194
solving time: 0.193896
0.0141194
```

2.11 Matrix inverse

The function `inv` gives the inverse matrix of A, i.e. a matrix A^{-1} such that

$$A \cdot A^{-1} = I,$$

where I denotes the identity matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3,3);                 // A : random 3x3 matrix
  array IA=inverse(A);                // IA: inverse of A
  af_print(matmul(A,IA),0);           // Check if A*IA=I
  return 0;
}
```

```
/*
matmul(A,IA)                            // A*IA (should be I)
[3 3 1 1]
     1      0      0
     0      1      0
     0      0      1
*/
```

To check the efficiency of ArrayFire `inv` function let us try to invert a 8000x8000 random matrix

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
    int n = 8e3;
    array A = randu(n, n);              // 8000x8000 random matrix
    array I=identity(n,n);
    array IA = inverse(A);              // Warm up
    for(int k=0;k<3;k++){
        timer::start();
        IA = inverse(A);                // Inverse matrix
        af::sync();
        printf("inverting time: %g\n", timer::stop());
        printf("%g\n",norm(matmul(A,IA)-I,AF_NORM_VECTOR_INF));
    }                                                   //error
    return 0;
}
/*
inverting time: 0.578662                // Inverting time
0.00256269                  // L_inf norm of error (single prec.)
inverting time: 0.211771
0.00256269
inverting time: 0.21056
0.00256269
*/
```

2.12 LU decomposition

The LU decomposition represents permuted matrix A as a product:

$$p(A) = L \cdot U,$$

where p is a permutation of rows, L is a lower triangular matrix, and U is an upper triangular matrix.
ArrayFire has two version of LU decomposition: unpacked `lu` and packed `luInPlace`, which should be used when memory is a concern.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main()
{
  int m = 3, n = 3;
  array L, U, p;
  array A = randu(m, n);
  lu(L, U, p, A);                       // LU, unpacked output
                                        // with pivoting
  af_print(A);                          // A is not overwritten
  af_print(L);
  af_print(U);
  array LU=matmul(L,U);                 // LU = L*U
  af_print(LU);
  af_print(p);                          // p - permutation of rows
  af_print(A(p,span));                  // p(A) - A permuted
                                        // should be equal to L*U
  array p1;
  array A1=A.copy();
  luInPlace(p1,A1,false);               // LU, packed  output
  af_print(A1);                         // A is overwritten
  array l1=lower(A1,true);              // l1 is in lower triangle
  array u1=upper(A1,false);             // u1 is in upper triangle
  af_print(matmul(l1,u1));              // print l1*l2
  af_print(A(p1,span));                 // p1(A1) should be eq. l1*l2
  af_print(p1);                         // p1 - permutation of rows
  return 0;
}
/*
A                                       // A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
```

```
L                                          // L
[3 3 1 1]
    1.0000     0.0000     0.0000
    0.0424     1.0000     0.0000
    0.8037     0.5536     1.0000

U                                          // U
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0000     0.4072     0.4656
    0.0000     0.0000     0.3212

LU                                         // L*U
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
    0.7402     0.9690     0.6673


p                                          // permutation of rows
[3 1 1 1]
         1
         2
         0

A(p,span)                                  // p(A) should be eq. L*U
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
    0.7402     0.9690     0.6673

A1                                         // copy of A
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0424     0.4072     0.4656
    0.8037     0.5536     0.3212
matmul(l1,u1)                              // l1*l2
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
    0.7402     0.9690     0.6673
```

```
A(p1,span)                                  // p1(A1) should be eq. l1*l2
[3 3 1 1]
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
    0.7402     0.9690     0.6673

p1                                          // permutation of rows
[3 1 1 1]
         1
         2
         0
*/
```

Let us consider a larger random matrix and check the efficiency of ArrayFire `lu` function.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 8e3;
  array L, U, p;
  array A = randu(n, n);                  // 8000x8000 matrix
  array A1=A.copy();
  for(int k=0;k<3;k++){
    timer::start();
    luInPlace(p,A1,false);        // in place LU decomposition
    af::sync();
    printf("LU time: %g\n", timer::stop());
  }
  return 0;
}
LU time: 0.357864
LU time: 0.237496
LU time: 0.225606
```

2.13 Cholesky decomposition

If A is square, symmetric ($A^T = A$ or $A^H = A$) and positive definite ($x \cdot A \cdot x > 0$ for $x \neq 0$) then faster decomposition

$$A = L \cdot L^T \text{ or } A = L^T \cdot L$$

is possible, where L is a lower triangular matrix in the first formula and upper triangular in the second one. As in the LU case there are two versions: unpacked – `cholesky` and packed – `choleskyInPlace`, used when memory is a concern.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3,3);               // A 3x3 matrix
  A=A+A.T();                        // A symmetric and
  A=A+4*identity(3,3);              // positive definite
  af_print(A);
  array Aupper=A.copy();            // using the upper triangle
  array Alower=A.copy();            // using the lower triangle
  choleskyInPlace(Aupper,true);     // cholesky in place - upp.tr.
  choleskyInPlace(Alower,false);    // cholesky in place - low.tr.
  af_print(Aupper);                 // print the res.in upper tr.
  af_print(Alower);                 // print the res.in lower tr.
  array U=upper(Aupper);            // U is the upper triangle
  af_print(U);                      // print U
  af_print(matmul(U.T(),U));        // U.T*U should be equal to A

  array Uout;
  array Lout;                       // unpacked version
  cholesky(Uout,A,true);            // cholesky decomp. - upper tr.
  cholesky(Lout,A,false);           // cholesky decomp. - lower tr.
  af_print(Uout);                   // Uout is equal to Aupper
  af_print(Lout);                   // Lout is equal to Alower
  return 0;
}
/*
A                                   // A
[3 3 1 1]
```

```
    5.4804     1.8900     0.7063
    1.8900     5.8503     0.5563
    0.7063     0.5563     4.9404

Aupper                              // Aupper after choleskyInPlace
[3 3 1 1]
    2.3410     0.8073     0.3017
    1.8900     2.2800     0.1371
    0.7063     0.5563     2.1979

Alower                              // Alower after choleskyInPlace
[3 3 1 1]
    2.3410     1.8900     0.7063
    0.8073     2.2800     0.5563
    0.3017     0.1371     2.1979

U                                   // upper triangle of Aupper
[3 3 1 1]
    2.3410     0.8073     0.3017
    0.0000     2.2800     0.1371
    0.0000     0.0000     2.1979


matmul(U.T(),U)                     // U.T*U should be equal to A
[3 3 1 1]
    5.4804     1.8900     0.7063
    1.8900     5.8503     0.5563
    0.7063     0.5563     4.9404

Uout                                // Uout after cholesky (unpack.)
[3 3 1 1]                           // (is equal to Aupper upp. tr.)
    2.3410     0.8073     0.3017
    0.0000     2.2800     0.1371
    0.0000     0.0000     2.1979

Lout                                // Lout after cholesky (unpack.)
[3 3 1 1]                           // (is equal to Alower low.tr.)
    2.3410     0.0000     0.0000
    0.8073     2.2800     0.0000
    0.3017     0.1371     2.1979
*/
```

Now let us try a larger matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 8e3;
  array A = randu(n, n);
  A=A+A.T();                          // A: 8000x8000 symmetric
  A=A+100*identity(n,n);              // positive definite matrix
  array Uout;
  for(int k=0;k<3;k++){
    timer::start();
    cholesky(Uout,A,true);            // Cholesky decomp
    af::sync();
    printf("cholesky time: %g\n", timer::stop());
  }
  return 0;
}
/*
cholesky time: 0.252116
cholesky time: 0.092311
cholesky time: 0.08885  */
```

2.14 QR decomposition

Let us recall, that QR decomposition allows for representing matrix A as a product

$$A = Q \cdot R,$$

where Q is an orthogonal matrix (i.e. $Q \cdot Q^T = I$) and R is upper triangular matrix.
As in the previous sections, one can use the unpacked version `qr` and and the packed one `qrInPlace`.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main()
{
  int n = 3;
```

```
  array Q, R;
  array tau;
  array A = randu(n, n);           // A: random 3x3 matrix
  array Ain=A.copy();              // a copy of A

  qrInPlace(tau,Ain);              // Packed output
  qr(Q, R, tau, A);                // unpacked output
  af_print(A);
  af_print(Q);                     // print the orthogonal factor
  af_print(matmul(Q,Q.T()));       // Q*Q^T (should be equal to I)
  af_print(R);                     // print the triangular factor
  af_print(matmul(Q,R));           // Q*R (should be equal to A)
  return 0;
}
/*
A                                  // A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
Q                                  // the orthogonal factor Q
[3 3 1 1]
   -0.6261     0.2930    -0.7226
   -0.7790    -0.2743     0.5638
   -0.0330     0.9159     0.4000
matmul(Q,Q.T())                    // check if Q*Q^T = I
[3 3 1 1]
    1.0000    -0.0000     0.0000
   -0.0000     1.0000    -0.0000
    0.0000    -0.0000     1.0000
R                                  // the triangular factor R
[3 3 1 1]
   -1.1822    -1.3421    -0.5190
    0.0000     0.4390     0.5961
    0.0000     0.0000    -0.2321
matmul(Q,R)                        // check if Q*R=A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702
*/
```

The efficiency of ArrayFire `qr` function can be checked using the following code.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 2e3;
  array Q, R;
  array tau;
  array A = randu(n, n);          // Random 2000x2000 matrix
  for(int k=0;k<3;k++){
    timer::start();
    qr(Q,R,tau,A);                // QR decomposition
    af::sync();
    printf("qr time: %g\n", timer::stop());
  }
  return 0;
}
/*
qr time: 1.22483
qr time: 0.287861
qr time: 0.292128
*/
```

2.15 Singular Value Decomposition

For $m \times n$ matrix A the singular value decomposition (SVD) has the form

$$A = U \cdot S \cdot V,$$

where U, V are orthogonal matrices of dimension $m \times m$ and $n \times n$ respectively. The $m \times n$ matrix S is diagonal and has only positive or zero elements on its diagonal (the singular values of A).

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int m = 3, n = 3;
```

```
  array U,S,Vt,s;
  array A = randu(m, n);                // Random 3x3 matrix
  svd(U,s,Vt, A);                       // SVD decomposition
  af_print(A);
  af_print(s);
  S=diag(s,0,false);                    // Diagonal matrix S
  af_print(S);
  af_print(U);
  af_print(Vt);
  af_print(matmul(U,matmul(S,Vt)));     // Check if U*S*Vt=A
  af_print(matmul(U,U.T()));            // Check orthogonality of U
  af_print(matmul(Vt,Vt.T()));          // Check orthogonality of V
  return 0;
}
/*
A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702

s                                   // Vector of singular values
[3 1 1 1]
    1.9311
    0.5739
    0.1087

S                               // Diagonal matrix with diagonal s
[3 3 1 1]
    1.9311     0.0000     0.0000
    0.0000     0.5739     0.0000
    0.0000     0.0000     0.1087


U
[3 3 1 1]
   -0.7120     0.3365    -0.6163
   -0.6502    -0.6475     0.3975
   -0.2653     0.6837     0.6798
```

```
Vt
[3 3 1 1]
   -0.5884    -0.7301    -0.3476
   -0.5586     0.0561     0.8275
   -0.5846     0.6811    -0.4409

matmul(U,matmul(S,Vt))                  // U*S*Vt = A
[3 3 1 1]
    0.7402     0.9690     0.6673
    0.9210     0.9251     0.1099
    0.0390     0.4464     0.4702

matmul(U,U.T())                         // U*U^T=I
[3 3 1 1]
    1.0000     0.0000    -0.0000
    0.0000     1.0000     0.0000
   -0.0000     0.0000     1.0000

matmul(Vt,Vt.T())                       // Vt*Vt^T=I
[3 3 1 1]
    1.0000    -0.0000     0.0000
   -0.0000     1.0000     0.0000
    0.0000     0.0000     1.0000
*/
```

Now let us try to apply the `svd` function to larger matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 2e3;
  array U,S,Vt,s;                       // U,V - orthogonal matrices
                                        // S -matrix with singul.val.
                                        // s - vector of singul. val.
  array A = randu(n, n);                // 2000x2000 random matrix
  timer::start();
  svd(U,s,Vt, A);                       // SVD decomp. A=U*S*Vt
  af::sync();
  printf("SVD time: %g\n", timer::stop());
  return 0;
```

```
}
/*
SVD time: 17.3208
SVD time: 11.8411
SVD time: 11.9405
*/
```

2.16 Plotting with ArrayFire

2.16.1 Two-dimensional plot

To obtain a two-dimensional plot in ArrrayFire-C/C++ one can use the function `plot`. For example the sine function can be plotted as follows:

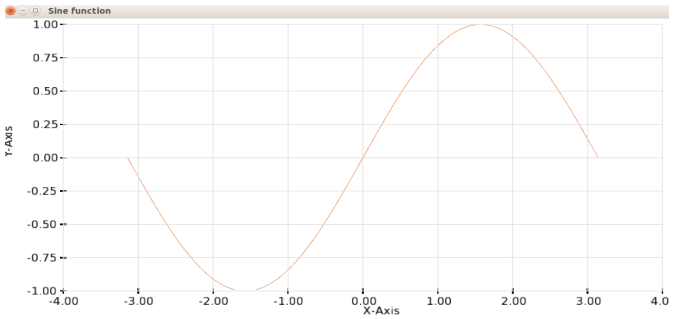

Figure 2.1: Plot of sine function in ArrayFire

```
#include <arrayfire.h>
#include <cstdio>
#include <math.h>
using namespace af;
static const float PRECISION = 1.0f/50.0f;
int main(int argc, char *argv[])
{
  af::Window myWindow(800, 600, "Sine function");
  array X = seq(-af::Pi, af::Pi, PRECISION);   // [-Pi,Pi] interval
  array Y = sin(X);                            // y=sin(x)
  do{
      myWindow.plot(X, Y);                     // plot
  }while(!myWindow.close());                   // close window
  return 0;                                    // to finish
}
```

2.16.2 Three-dimensional plot

ArrayFire-C/C++ allows also for three-dimensional plots. The graph of a function of two variables can be obtained using `surface`. For example, let us show how to plot

$$f(x, y) = \cos(\sqrt{x^2 + y^2}).$$

```
#include <arrayfire.h>
#include <cstdio>
#include <math.h>
using namespace af;
static const int M = 30;                // MxN  xy grid
static const int N =  M;
int main(int argc, char *argv[])
{
  af::Window myWindow(800, 600, "3D Surface");
  const array x=3*af::Pi*iota(dim4(N,1),dim4(1,N))/M-1.5*af::Pi;
  const array y=3*af::Pi*iota(dim4(1,N),dim4(N,1))/M-1.5*af::Pi;
  array z =  cos(sqrt((x*x)+(y*y)));
  while(!myWindow.close()) {          // close window to finish
    myWindow.surface(x, y, z);        // surface plot
  }
  return 0;
}
```

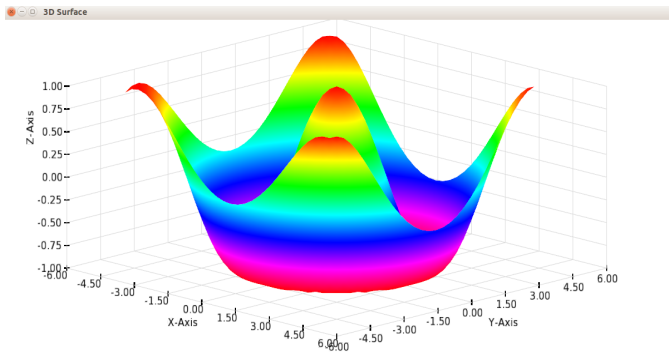

Figure 2.2: Three-dimensional plot in ArrayFire

2.16.3 Image-plot

There is also `image` function in ArrayFire which allows to plot the two dimensional map of the function of two variables. Let us show how to plot the function from the previous example using `image`.

```
#include <arrayfire.h>
#include <cstdio>
#include <math.h>
using namespace af;
static const int M = 100;              // MxN xy grid
static const int N =  M;
int main(int argc, char *argv[])
{
  af::Window myWindow(800, 600, "Image");
  const array x=3*af::Pi*iota(dim4(N,1),dim4(1,N))/M-1.5*af::Pi;
  const array y=3*af::Pi*iota(dim4(1,N),dim4(N,1))/M-1.5*af::Pi;
  array z =  cos(sqrt((x*x)+(y*y)));
  while(!myWindow.close()) {
    myWindow.image(z);                 // image plot of
  }                                    // z=cos(sqrt(x^2+y^2))
  return 0;
}
```

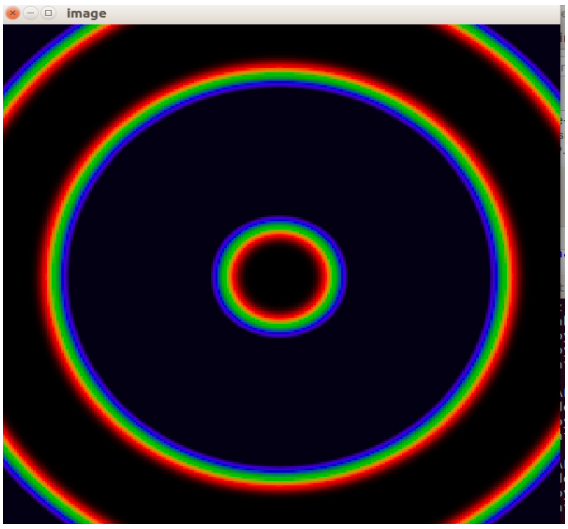

Figure 2.3: Image-plot in ArrayFire