Matrix computations on the GPU with ArrayFire for Python and C/C++

Andrzej Chrzeszczyk
Jan Kochanowski University

Matrix Computations on the GPU with ArrayFire for Python and C/C++

by Andrzej Chrzęszczyk of Jan Kochanowski University

Phone: +1.800.570.1941 • sales@accelereyes.com • www.accelereyes.com

Foreward by John Melonakos of AccelerEyes

One of the biggest pleasures we experience at AccelerEyes is watching programmers develop awesome stuff with ArrayFire. Oftentimes, ArrayFire programmers contribute back to the community in the form of code, examples, or help on the community forums.

This document is an example of an extraordinary contribution by an ArrayFire programmer, written entirely by Andrzej Chrzęszczyk of Jan Kochanowski University. Readers of this document will find it to be a great resource in learning the ins-and-outs of ArrayFire.

On behalf of the rest of the community, we thank you Andrzej for this marvelous contribution.

Foreward by Andrzej Chrzęszczyk of Jan Kochanowski University

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.

In the first two chapters 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:

- http://www.accelereyes.com/products/arrayfire

In the present text we describe the ArrayFire 1.1 free version. It allows for efficient dense matrix computations in single precision on single GPU. The readers interested in double precision linear algebra, multiple GPUs, or sparse matrices should consider ArrayFire Pro.

Contents

Foreword 1

1 ArrayFire-Python **4**

1.1 Introductory remarks 4
1.2 Defining arrays 6
1.3 Random arrays 8
1.4 Rearranging arrays 10
1.5 Matrix addition multiplication and powering 13
1.6 Sums and products of elements 16
1.7 Mean, variance, standard deviation and histograms 17
1.8 Solving linear systems 20
1.9 Matrix inverse 23
1.10 LU decomposition 24
1.11 Cholesky decomposition 27
1.12 QR decomposition 30
1.13 Singular Value Decomposition 32
1.14 Pseudo-inverse 36
1.15 Hessenberg decomposition 38
1.16 Plotting with ArrayFire 39
1.16.1 Two-dimensional plot 39
1.16.2 Three-dimensional plot 40
1.16.3 Image-plot 40

2 ArrayFire-C/C++ **42**

2.1 Introductory remarks 42
2.2 How to compile the examples 43
2.3 Defining arrays 43
2.4 Random arrays 47
2.5 Rearranging arrays 50
2.6 Determinant, norm and rank 54

2.7 Elementary arithmetic operations on matrices 57
2.8 Sums and products of elements 61
2.9 Dot product 63
2.10 Mean, variance, standard deviation and histograms 64
2.11 Solving linear systems 66
2.12 Matrix inverse 67
2.13 LU decomposition 68
2.14 Cholesky decomposition 70
2.15 QR decomposition 72
2.16 Singular Value Decomposition 74
2.17 Pseudo-inverse 76
2.18 Hessenberg decomposition 77
2.19 Plotting with ArrayFire 79
2.19.1 Two-dimensional plot 79
2.19.2 Three-dimensional plot 80
2.19.3 Image-plot 81

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 1.1 and Python installed. We shall use `ipython` interactive shell. 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.

Remark. The users of free version of ArrayFire should have a working Internet connection.

New users should probably begin with the commands:

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

```
ArrayFire v1.1 (build c67f168) by AccelerEyes (64-bit Linux)
License Type: Concurrent Network (27000@server.accelereyes.com)
Addons: none
CUDA toolkit 4.1, driver 290.10
GPU0 GeForce GTX 580, 1536 MB, Compute 2.0 (single,double)
Display Device: GPU0 GeForce GTX 580
Memory Usage: 1437 MB free (1536 MB total)
None
```

To see all possible ArrayFire-Python commands in Python interactive session it suffices to write `af.` and press `[Tab]` button:

```
af.[TAB]

af.RTE          af.diff2          af.join       af.resize
af.abs          af.dilate         af.log        af.rotate
af.acos         af.eigen          af.lower      af.round
af.acosh        af.eigen_values   af.lu         af.rtypes
af.all          af.erf            af.matmul     af.select
af.any          af.erode          af.matpow     af.shift
af.areas        af.eval           af.max        af.sin
af.array        af.exp            af.mean       af.singular_values
af.arrows       af.fft            af.medfilt    af.sinh
af.asin         af.fft2           af.median     af.solve
af.asinh        af.filter         af.min        af.sort
af.atan         af.fir            af.np         af.sqrt
af.atanh        af.flat           af.ones       af.std
af.ceil         af.fliph          af.pinv       af.sum
af.centroids    af.flipv          af.plot       af.svd
af.cholesky     af.floor          af.plot3d     af.sync
af.complex      af.grid           af.points     af.tan
af.conj         af.hessenberg     af.pow        af.tanh
af.convolve     af.histogram      af.prod       af.tic
af.convolve2    af.identity       af.qr         af.tile
af.cos          af.ifft           af.randn      af.toc
af.cosh         af.ifft2          af.randu      af.types
af.current      af.iir            af.range      af.upper
af.det          af.imag           af.rank       af.var
af.devices      af.imgplot        af.real       af.zeros
af.diag         af.info           af.regions
af.diff1        af.inv            af.reshape
```

Remark. Some of the listed functions (for example `eigen`) work only in ArrayFire-Pro version.
Any element of the obtained list can be used with the question mark at the end. This gives us the access to the help concerning the choosen function. For example:

```
af.solve?
Type:       builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
```

```
String Form:<built-in function solve>
Namespace:  Interactive
Docstring:
Solve a system of equations AX=B

Parameters:
----------
A: Supports{array}
State matrix for the set of equations

B: Supports{array}
Observed results for the set of equations

Usage:
------
A = af.randu(5,5)
X0 = af.randu(5,1)
B = A * X0
X = solve(A, B)
```

Remark. Recall that the most complete description of ArrayFire-Python can be found on `http://www.accelereyes.com/arrayfire/python/`.

1.2 Defining arrays

ArrayFire Python interface allows for easy definitions of vectors and matrices. The simplest way to define an array in ArrayFire-Python is to use one of the commands `zeros`, `ones` or `identity`.

```
import arrayfire as af

a=af.zeros(3,3)                      # Zeros
print(a)

[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]

a=af.ones(3,3)                       # Ones
print(a)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

a=af.identity(3,3)                # Identity
print(a)

[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

To obtain new ArrayFire arrays one can use numpy arrays and the `array` function.

```
import numpy as np
#x=np.array([[0,1,2],[3,4,5],[6,7,8]])   # alternative definitions
#x=np.array([[0,1,2],[3,4,5],[6,7,8]],dtype='float32')
x=np.array([[0,1,2],[3,4,5],[6,7,8]]).astype('float32')
import arrayfire as af
a=af.array(x)                     # ArrayFire array definition based
print(a)                          # on numpy arrays

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

Possible ArrayFire types are:

```
af.types
{'bool':4,'complex128':3,'complex64':1,'float32':0,'float64':2}
```

The last array can be also obtained without numpy.

```
import arrayfire as af
a=af.range(9)                     # Range 0,1,...,8
print(a)
[[ 0.]
 [ 1.]
 [ 2.]
 [ 3.]
 [ 4.]
```

```
 [ 5.]
 [ 6.]
 [ 7.]
 [ 8.]]

A=af.reshape(a,3,3)
print(A)

[[ 0.  3.  6.]                     # Range 0,...,8 as 3x3 matrix.
 [ 1.  4.  7.]                     # Column wise order!
 [ 2.  5.  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.array(a)
print(A)

[[ 0.  1.  2.]                     # Array with a_{ij}=i+j
 [ 1.  2.  3.]
 [ 2.  3.  4.]]
```

Here is an example with complex entries:

```
#continuation
b=a+1j*a                           # Complex array in numpy
B=af.array(b)                      # ArrayFire complex array
print(B)

[[ 0.+0.j  1.+1.j  2.+2.j]
 [ 1.+1.j  2.+2.j  3.+3.j]
 [ 2.+2.j  3.+3.j  4.+4.j]]
```

1.3 Random arrays

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

```
import arrayfire as af
```

```
ru=af.randu(3,3)                    # Random array, uniform distr.
print(ru)

[[ 0.58755869  0.37504062  0.24048613]
 [ 0.41475514  0.09369452  0.63255239]
 [ 0.28493077  0.77932     0.21329425]]

rn=af.randn(3,2,dtype='complex64')  # Random array, normal distr.
print(rn)

[[ 0.30717012+1.11348569j -0.03761575-0.71087211j]
 [-1.29028428-0.88222182j -0.06493776-0.81823027j]
 [ 1.42657781-0.12852676j  0.28100717+0.82957321j]]
```

Using the functions `ceil`, `floor` and `round` one can obtain random arrays with integer entries.

```
A=af.randn(3,3)*5
a=af.round(A)                       # Rounding array elements
print(a)

[[  1.  -4.   0.]
 [ -2.  13.  -0.]
 [  0.  -1.   1.]]

a=af.floor(A)                       # Applying floor function
print(a)

[[  1.  -4.   0.]
 [ -2.  12.  -1.]
 [  0.  -2.   0.]]

a=af.ceil(A)                        # Applying ceiling function
print(a)

[[  2.  -3.   1.]
 [ -1.  13.  -0.]
 [  1.  -1.   1.]]
```

Generation of random arrays gives us the opportunity to check the efficiency of ArrayFire. Let us check the time needed for generation of a 10000x10000 random array on GTX 580 card.

```
import arrayfire as af
N=1e4
ru=af.randu(N,N)                    # Warm up
af.tic();ru=af.randu(N,N);af.sync();print("uniform:",af.toc())
rn=af.randn(N,N)                    # Warm up
af.tic();rn=af.randn(N,N);af.sync();print("normal:",af.toc())
#uniform: 0.009668                  # Uniform distr. generation time
#normal: 0.013811                   # Normal distr. generation time
                                    # on GTX 580
```

1.4 Rearranging arrays

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

```
import arrayfire as af
A=af.reshape(af.range(9),3,3)
print(A)                            # A

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

AT=A.T()                            # Transposition of A
print(AT)

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

import numpy as np
a=np.arange(9.0).reshape(3,3)
B=af.array(a+2j*a)                  # Complex matrix example
print(B)

[[ 0. +0.j  1. +2.j  2. +4.j]
 [ 3. +6.j  4. +8.j  5.+10.j]
 [ 6.+12.j  7.+14.j  8.+16.j]]
```

```
BC=af.conj(B)                           # Conjugation
print(BC)

[[ 0. -0.j  1. -2.j  2. -4.j]
 [ 3. -6.j  4. -8.j  5.-10.j]
 [ 6.-12.j  7.-14.j  8.-16.j]]

BH=B.H()                                # Conjugate transposition
print(BH)

[[ 0. -0.j  3. -6.j  6.-12.j]
 [ 1. -2.j  4. -8.j  7.-14.j]
 [ 2. -4.j  5.-10.j  8.-16.j]]
```

One can also flip the array horizontally or vertically:

```
#continuation
print(A)                                # A

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


AFH=af.fliph(A)                         # Flip horizontally
print(AFH)

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

AFV=af.flipv(A)                         # Flip vertically
print(AFV)

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

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

```
#continuation
```

```
print(A)                                  # A

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

AF=af.flat(A)                             # Flattened A
print(AF)

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

AL=af.lower(A)                            # Lower triangular part
print(AL)

[[ 0.  0.  0.]
 [ 1.  4.  0.]
 [ 2.  5.  8.]]

AU=af.upper(A)                            # Upper triangular part
print(AU)

[[ 0.  3.  6.]
 [ 0.  4.  7.]
 [ 0.  0.  8.]]
```

There are also shift and rotate operations.

```
F=af.flat(A)                              # Flat form
print(F)

[[ 0.]
 [ 1.]
 [ 2.]
 [ 3.]
```

```
 [ 4.]
 [ 5.]
 [ 6.]
 [ 7.]
 [ 8.]]

R=af.rotate(F,3)                          # Rotation
print(R)

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

S=af.shift(F,1)                           # Shift
print(S)

[[ 8.]
 [ 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`.

```
import arrayfire as af
A=af.reshape(af.range(9),3,3)
print(A)                                  # A
```

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

I=af.identity(3,3)
B=A+I                                # Matrix addition
print(B)

[[ 1.  3.  6.]
 [ 1.  5.  7.]
 [ 2.  5.  9.]]

B=af.matmul(A,I)                     # Matrix multiplication
print(B)

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

There is also the element-wise version of multiplication.

```
#continuation
B=A*A                               # Element-wise multiplication;
print(B)                            # A by A

[[  0.   9.  36.]
 [  1.  16.  49.]
 [  4.  25.  64.]]


B=A*I                               # Element-wise multiplication;
print(B)                            # A by I

[[ 0.  0.  0.]
 [ 0.  4.  0.]
 [ 0.  0.  8.]]
```

The matrix power ie. $A^n = A \cdot \ldots \cdot A$ (n-times) can be obtained using `matpow` function:

```
#continuation
B=af.matpow(A,2)                    # Matrix power
```

```
print(B)

[[  15.   42.   69.]
 [  18.   54.   90.]
 [  21.   66.  111.]]
```

and the element-wise power, using the `pow` function:

```
B=af.pow(A,2)                        # Element-wise power
print(B)

[[  0.   9.  36.]
 [  1.  16.  49.]
 [  4.  25.  64.]]
```

The matrix product operation gives us a next opportunity to check the graphics card performance (and compare it to CPU performance).

```
import numpy as np
import time
a=np.random.rand(10000,10000).astype('float32')
t=time.time();b=np.dot(a,a);print time.time()-t
                                     # 10000x10000 matr. multipl.
                                     # on CPU, using numpy
# 8.11502814293                      # Multiplication time
                                     # on i7 2600k CPU

import arrayfire as af
A=af.array(a)
af.tic();B=matmul(A,A);af.sync();print af.toc()
                                     # 10000x10000 matr. multipl.
                                     # on GPU, using ArrayFire
#1.965441                            # Multiplication time
                                     # on GTX 580
```

(in scipy one can find also `sgemm` function but it does not outperform the `dot` function).
To check the `matpow` function performance on GTX 580 card we were forced to lower the dimensions.

```
import arrayfire as af
```

```
N=9000;A=af.randu(N,N)              # 9000x9000 random matrix
af.tic();B=af.matpow(A,2.0);af.sync();print af.toc()
#1.64776801                         # Squaring time on GTX 580
```

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.

```
import arrayfire as af
a=af.ones(3,3);print(a)             # a: 3x3 matrix of ones

[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

print(af.sum(a,0))                  # Sums of columns

[[ 3.  3.  3.]]

print(af.sum(a,1))                  # Sums of rows

[[ 3.]
 [ 3.]
 [ 3.]]

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

[[ 9.]]

a=a*2;print(a)                      # 2*a

[[ 2.  2.  2.]
 [ 2.  2.  2.]
 [ 2.  2.  2.]]

print(af.prod(a,0))                 # Product of columns

[[ 8.  8.  8.]]
```

```
print(af.prod(a,1))                # Products of rows

[[ 8.]
 [ 8.]
 [ 8.]]

print(af.prod(a))                  # Product of all elements

[[ 512.]]
```

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

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

```
import numpy as np
import arrayfire as af
N=1e8;a=af.range(1,N)             # 1,2,...,10^8-1
print(af.sum(1./(a*a)))           # sum_1^{10^8} 1/k^2

[[ 1.6449343]]

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

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.

```
import arrayfire as af
A=af.reshape(af.range(9),3,3)
print(A)                          # A

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

```
print(af.mean(A,0))                  # Averages of columns

[[ 1.  4.  7.]]

print(af.mean(A,1))                  # Averages of rows

[[ 3.]
 [ 4.]
 [ 5.]]

print(af.mean(A))                    # Average of all elements

[[ 4.]]
```

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

```
import arrayfire as af
N=1e8;x=af.randu(N)
m=af.mean(x);v=af.mean(x*x)-m*m  # var(x)=mean(x^2)-mean(x)^2
print(m)                         # Mean

[[ 0.49995065]]                  # Theoretical mean: 1/2=0.5

print(v)                         # Variance with the help of mean

[[ 0.08333674]]                  # Theoretical variance:
                                 # 1/12=0.08333333...
print(af.var(x))                 # ArrayFire variance

[[ 0.0833351]]

print(af.std(x))                 # Standard deviation

[[ 0.28867564]]                  # Theoretical standard dev:
                                 # sqrt(1/12)= 0.28867513...
```

In the case of normal distribution we obtain:

```
N=1e8;x=af.randn(N)
m=af.mean(x);v=af.mean(x*x)-m*m
```

```
print(m)

[[  4.49110848e-06]]            # Theoretical mean: 0

print(v)                        # Variance with the help of mean

[[ 0.99990606]]                 # Theoretical variance: 1

print(af.var(x))                # ArrayFire variance

[[ 1.00017381]]

print(af.std(x))                # Standard deviation

[[ 1.00008702]]                 # Theoretical standard dev:
                                # sqrt(1)=1
```

ArrayFire-Python interface contains also a fast histogram function.

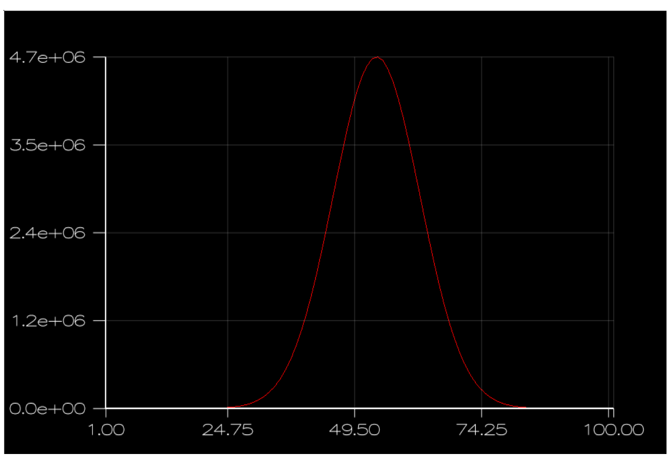

Figure 1.1: Histogram for normally distributed random array

```
import arrayfire as af
n=10000
ru=af.randu(n,n)                  # 10000x10000 uniformly
                                  #  distributed array
af.tic();ru=af.randu(n,n);af.sync();t=af.toc();
```

```
print "generation uniform: ",t

#generation uniform:  0.009646    # Generation time on GTX 580

af.tic();h=af.histogram(ru,100);af.sync();t=af.toc();
print "histogram uniform:",t

#histogram uniform: 0.01316       # Histogram time on GTX 580

rn=af.randn(n,n)                  # 10000x10000 normally
                                  # distributed array
af.tic();rn=af.randn(n,n);af.sync();t=af.toc();
print "generation normal: ",t

#generation normal:  0.017937     # Generation time on GTX 580

af.tic();h=af.histogram(rn,100);af.sync();t=af.toc();
print "histogram normal: ",t

#histogram normal:  0.016088      # Histogram time on GTX 580
```

1.8 Solving linear systems

ArrayFire-Python allows for efficient numerical solving of linear systems

$$Ax = B,$$

where A, B are ArrayFire arrays.

```
import arrayfire as af
A=af.randu(3,3);print(A)          # Random coefficients matrix

[[ 0.26425067  0.00801698  0.54594052]
 [ 0.25824451  0.21248953  0.16285282]
 [ 0.79723495  0.98772854  0.53500992]]

b=af.randu(3,1);print(b)          # RHS will be A*b

[[ 0.18229593]
 [ 0.04821385]
 [ 0.31683788]]
```

```
B=af.matmul(A,b)                 # RHS

X=af.solve(A,B);print(X)         # Solution X should be equal to b

[[ 0.18229593]
 [ 0.04821387]
 [ 0.31683788]]

print(af.max(af.abs(X-b)))       # Error

[[  1.11758709e-08]]
```

On GTX 580 card we were able to solve 9000x9000 system.

```
import arrayfire as af
N=9000;A=af.randu(N,N)           # 9000x9000 system
b=af.randu(N,1);B=A*b
af.tic();X=af.solve(A,B);af.eval(X);af.sync();print(af.toc())

#0.890319                         # Solving time in ArrayFire
                                  # on GTX 580
print(af.max(af.abs(X-b)))       # Error

[ 0.0332551]]
```

For comparison let us perform a similar computation in numpy or scipy on CPU. Let us begin with `solve` function from numpy.

```
import numpy as np
import time
N=9000;a=np.random.rand(N,N).astype('float32')
                                  # 9000x9000 system in numpy on CPU
b=np.random.rand(N,1).astype('float32');B=np.dot(a,b)
t=time.time();x=np.linalg.solve(a,B);print time.time()-t

#5.701667                         # Solving time in numpy
                                  # on i7 2600k CPU
print(np.max(np.abs(x-b)))       # Error

#0.014217019
```

```
print(x.dtype)

#float32                              # Precision
```

Note however than numpy can give in the same time a more accurate solution using double precision arithmetic.

```
N=9000;a=np.random.rand(N,N)
b=np.random.rand(N,1);B=np.dot(a,b)
t=time.time();x=np.linalg.solve(a,B);print time.time()-t

#5.409778                             # Solving time in numpy
                                      # on i7 2600 CPU
print(np.max(np.abs(x-b)))

#8.8492747851720566e-11               # Error in double precision

print(x.dtype)

#float64                              # Precision
```

The analogous double precision calculations on GPU require ArrayFire-Pro (which is non-free).
To obtain the single precision solution on CPU one can use `sgesv` function from scipy.

```
import numpy as np
import scipy as sc
import time
from scipy import linalg
N=9000;a=np.random.rand(N,N).astype('float32')
                                      # 9000x9000 random matrix
b=np.random.rand(N,1).astype('float32')
                                      # 9000x1 column vector
t=time.time();lu,piv,x,info=sc.linalg.lapack.flapack.sgesv(a,b);
print time.time()-t
#3.051084041                          # Time for single precision
                                      # solution on i7 2600 CPU
                                      # with scipy
```

1.9 Matrix inverse

ArrayFire allows 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 check that ArrayFire function `inv` gives the same result as numpy.

```
import numpy as np
a=np.random.rand(3,3)
ia=np.linalg.inv(a)                  # Inverse matrix in numpy
print(ia)

[[ 5.40068179 -2.06390562 -2.83870761]
 [-1.12050388 -0.49476792  2.06648821]
 [-1.45042153  2.86095935  0.12135636]]

print(np.round(np.dot(a,ia)))        # Check if a*a^-1=I

[[ 1.  0.  0.]
 [-0.  1.  0.]
 [-0. -0.  1.]]

import arrayfire as af
a=a.astype('float32')
A=af.array(a)
IA=af.inv(A)                         # Inverse matrix in ArrayFire
print(IA)

[[ 5.40068245 -2.06390572 -2.83870792]
 [-1.12050438 -0.49476787  2.0664885 ]
 [-1.45042133  2.86095905  0.12135629]]

print(af.round(af.matmul(A,IA)))     # Check if A*A^-1=I

[[ 1. -0.  0.]
 [ 0.  1. -0.]
 [-0. -0.  1.]]
```

In the next script we give a comparison of numpy and ArrayFire speed in matrix inversion.

```
import numpy as np
a=np.random.rand(7000,7000).astype('float32')
import time                           # 7000x7000 random matrix
t=time.time()                         # in numpy
ia=np.linalg.inv(a)                   # Inversion in numpy
print time.time()-t
#9.74611902237                        # Inversion time in numpy
                                      # on i7 2600k CPU
import arrayfire as af
A=af.array(a)                         # ArrayFire 7000x7000 matrix
af.tic()
IA=af.inv(A)                          # Inversion in ArrayFire
af.eval(IA)
t=af.toc()
print(str(t))
#1.504929                             # Inversion time in ArrayFire
                                      # on GTX 580
```

Remark. Scipy contains a single precision `sgetri` function for inverting matrices but we were unable to obtain a good performance using it.

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 begin with scipy version of LU.

```
import numpy as np
import scipy as sc
from scipy import linalg
a=np.random.rand(3,3)
```

```
sc.linalg.lu_factor(a)              # LU factorization in scipy,
                                    # packed output
(array([[ 0.93282818,  0.76701732,  0.749577  ],
       [ 0.66334124,  0.43847395, -0.0494742 ],
       [ 0.68396042,  0.13095973,  0.45653624]]),
array([2, 1, 2], dtype=int32))      # Permutation


p,l,u=sc.linalg.lu(a)               # a=p*l*u
print(p)                            # Permutation matrix

[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [ 1.  0.  0.]]

print(l)                            # Lower triangular part l

[[ 1.          0.          0.        ]
 [ 0.49393723  1.          0.        ]
 [ 0.72628946  0.38392691  1.        ]]

print(u)                            # Upper triangular part u

[[ 0.96273811  0.65556923  0.68396042]
 [ 0.          0.62345813  0.32550773]
 [ 0.          0.          0.31110375]]
```

Now let us consider the ArrayFire version.

```
import arrayfire as af
a=a.astype('float32')
A=af.array(a)
L,U,P=af.lu(A)                      # LU factorization
                                    # in ArrayFire
print(L)                            # Lower triangular part L

[[ 1.          0.          0.        ]
 [ 0.49393722  1.          0.        ]
 [ 0.72628945  0.38392681  1.        ]]

 print(U)                           # Upper triangular part U
```

```
[[ 0.9627381   0.65556926  0.68396044]
 [ 0.         0.62345815  0.3255077 ]
 [ 0.         0.          0.31110382]]

print(P)                          # Permutation matrix P

[[ 0.  0.  1.]
 [ 0.  1.  0.]
 [ 1.  0.  0.]]
```

Let us check the equality $A = PLU$.

```
print(af.matmul(P,af.matmul(L,U))) # Check if P*L*U=A

[[ 0.6992265   0.71549535  0.93282819]
 [ 0.47553217  0.94726825  0.66334122]
 [ 0.9627381   0.65556926  0.68396044]]

print(A)                          # A

[[ 0.69922656  0.71549535  0.93282819]
 [ 0.4755322   0.94726819  0.66334122]
 [ 0.9627381   0.65556926  0.68396044]]
```

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

```
import arrayfire as af
A=af.randu(9000,9000)             # 9000x9000 random matrix
af.tic()
LU=af.lu(A)                       # LU factorization
af.eval(LU)
af.sync()
t=af.toc()
print(str(t))
#0.777192                         # Decomp. time on GTX 580
```

Compare the scipy CPU version, first in double precision:

```
import numpy as np
import scipy as sc
```

```
import time
from scipy import linalg
a=np.random.rand(9000,9000)          # 9000x9000 random matrix
t=time.time();lu,piv=sc.linalg.lu_factor(a);print time.time()-t
                                     # LU factorization
#5.36308979988                       # time in scipy on
                                     # i7 2600k CPU
print(lu.dtype)                      # using double precision

#float64
```

and next in single precision:

```
import numpy as np
import scipy as sc
import time
from scipy import linalg
N=9000;a=np.random.rand(N,N).astype('float32')
t=time.time();lu,piv,info=sc.linalg.lapack.flapack.sgetrf(a);
print time.time()-t
                                     # LU decomposition
#3.02882385254                       # time for single precision
                                     # on i7 2600k CPU with scipy
```

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. Let us begin with numpy version of the decomposition.

```
import numpy as np
a=np.random.rand(3,3)                # Random 3x3 matrix
a=a+np.transpose(a)+np.eye(3)*2      # Symmetric, positive definite
L=np.linalg.cholesky(a)              # Cholesky decomposition of a
print(L)                             # L

[[ 1.75574823,  0.        ,  0.        ],
 [ 0.70661975,  1.78719686,  0.        ],
```

```
 [ 0.36085409,  0.54213928,  1.52620185]]

print(np.dot(L,np.transpose(L)))         # L*L^T

[[ 3.08265186,  1.24064638,  0.63356893],
 [ 1.24064638,  3.69338408,  1.22389624],
 [ 0.63356893,  1.22389624,  2.75342275]]

print(a)                                 # a (=L*L^T)

[[ 3.08265186,  1.24064638,  0.63356893],
 [ 1.24064638,  3.69338408,  1.22389624],
 [ 0.63356893,  1.22389624,  2.75342275]]

print np.allclose(np.dot(L,np.transpose(L)),a)
                                         # Check if a=L*L^T
True
```

In ArrayFire one can use `cholesky` function, which gives a factorization in the form $A = L^T \cdot L$ with upper triangular matrix L.

```
import arrayfire as af
a=a.astype('float32')
A=af.array(a)                            # ArrayFire version of a
L=af.cholesky(A)                         # ArrayFire cholesky decomp.
print(L)                                 # L

[[ 1.75574827  0.70661974  0.36085409]
 [ 0.          1.78719687  0.54213929]
 [ 0.          0.          1.52620184]]

print(af.max(af.abs(A-af.matmul(L.T(),L))))
                                         # Check if |A-L^T*L|
[[  2.38418579e-07]]                     # is small
```

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

```
import arrayfire as af
N=8000;A=af.randu(N,N)                   # Random 8000x8000 matrix
B = A + A.T() + af.identity(N,N) * 100.0 # Symmetric, positive def.
af.tic();ch=af.cholesky(B);af.sync();print(af.toc())
                                         # Cholesky decomposition
```

```
                                          # in ArrayFire
#0.447913                                 # Decomp. time in ArrayFire
                                          # on GTX 580
```

Compare it to numpy version in double precision:

```
import numpy as np
import time
N=8000;A=np.random.rand(N,N)              # Random 8000x8000 matrix
B=A+np.transpose(A)+np.eye(N)*100         # Symmetric, positive def.
t=time.time(); L=np.linalg.cholesky(B);print time.time()-t
                                          # Cholesky decomposition
                                          # in numpy
#3.16157889366                            # Decomp. time in numpy
                                          # on i7 2600k CPU
print(L.dtype)                            # Double precision

#float64
```

and next to scipy single precision function `spotrf`:

```
import numpy as np
import scipy as sc
import time
from scipy import linalg as la
N=8000;A=np.random.rand(N,N).astype('float32')
                                          # Random 8000x8000 matrix
B=A+np.transpose(A)+np.eye(N).astype('float32')*100
                                          # Symmetric, positive def.
t=time.time();ch,info=la.lapack.flapack.spotrf(B);print time.time()-t
                                          # Cholesky decomposition
#1.50379610062                            # Decomp. time in scipy
                                          # on i7 2600k CPU
print(info)

#0                                        # Successful exit

print(ch.dtype)

#float32                                  # Single precision
```

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.
Numpy version of QR decomposition gives both matrices Q and R.

```
import numpy as np
a=np.random.rand(3,3)                        # a: 3x3 random matrix
Q,R=np.linalg.qr(a)
print(Q)                                     # Orthogonal part

[[-0.29624791  0.93160265 -0.21060314]
 [-0.64461043 -0.357727   -0.67565434]
 [-0.7047798  -0.06440421  0.70649666]]

print(R)                                     # Upper triangular part

[[-1.10554415 -0.16208647 -0.78529391]
 [ 0.          0.16167641 -0.10099383]
 [ 0.          0.          0.35246134]]

print(np.round(np.dot(Q.T,Q)))               # Orthogonality of Q
[[ 1.,  0.,  0.],
 [ 0.,  1.,  0.],
 [ 0.,  0.,  1.]]

print(np.allclose(a,np.dot(Q,R)))            # Check if a=Q*R

True
```

Here is the ArrayFire version of QR decomposition.

```
import arrayfire as af
a=a.astype('float32')                        # ArrayFire version
A=af.array(a)                                # of a array
Q,R=af.qr(A)                                 # QR decomposition
```

```
                                                  # in ArrayFire
print(Q)

[[-0.29624784  0.93160242 -0.21060318]      # Orthogonal part Q
 [-0.64461035 -0.35772702 -0.67565429]
 [-0.70477974 -0.06440431  0.70649666]]

print(R)
[[-1.10554421 -0.16208644 -0.78529382]      # Upper triangular part R
 [ 0.          0.16167639 -0.10099392]
 [ 0.          0.          0.35246137]]
```

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

```
import arrayfire as af
A=af.randu(8000,8000)                  # Random 8000x8000 matrix
af.tic()
QR=af.qr(A)                            # ArrayFire QR decomposition
af.eval(QR)
af.sync()
t=af.toc()
print(str(t))
#1.073498                              # QR decomp. time on GTX 580
```

Numpy is slower but since `np.linalg.qr` gives the interface to double precision LAPACK procedure, it allows for calculations on double precision matrices in the same time as on single precision ones.

```
import numpy as np
import time
N=8000;a=np.random.rand(N,N).astype('float32')
t=time.time(); r=np.linalg.qr(a,mode='r');print time.time()-t
                                       # Numpy QR on single prec. data
#8.52450108528                         # QR decomp. time for single
                                       # prec. data on i7 2600k CPU
print(r.dtype)

#float32                               # Single precision

N=8000;a=np.random.rand(N,N)           # Double precision matrix
```

```
t=time.time(); r=np.linalg.qr(a,mode='r'); print time.time()-t
                                    # Numpy QR on double prec. data
#8.27344202995                      # QR decomp. time for double
                                    # prec. data on i7 2600k CPU
print(r.dtype)

#float64                            # Double precision
```

In scipy we can choose (for example) a specialized single precision LAPACK function `sgeqrf`.

```
import numpy as np
import scipy as sc
import time
from scipy import linalg
N=8000;a=np.random.rand(N,N).astype('float32')
t=time.time(); qr,tau,work,info=sc.linalg.lapack.flapack.sgeqrf(a);
print time.time()-t                 # LAPACK sgeqrf from scipy

#3.80929803848                      # Time for single prec.
                                    # QR decomposition
print(info)                         # on i7 2600k CPU

#0                                  # Successful exit

print(qr.dtype)

#float32                            # Single precision
```

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 first show how this decomposition works in numpy.

```
import numpy as np
a=np.random.rand(4,3)
U,s,V=np.linalg.svd(a)                  # SVD in numpy
print(U)                                # Orthogonal matrix U

[[-0.49052701  0.38722803  0.38825728 -0.67726951]
 [-0.34457586 -0.6054373   0.65983033  0.28166838]
 [-0.65874124 -0.37555162 -0.64320464 -0.10634265]
 [-0.4546545   0.58520177  0.01296177  0.67131228]]

print(s)                                # Singular values

[ 2.05083035  0.64612715  0.35499201]

print(V)                                # Orthogonal matrix V

[[-0.69376461 -0.45634992 -0.55716731]
 [ 0.57329038 -0.81818595 -0.04370228]
 [-0.43592293 -0.34973776  0.82924948]]
```

Let us check the equality $a = U \cdot S \cdot V$ and the orthogonality of U, V.

```
#continuation
S=np.zeros((4,3))
S[:3,:3]=np.diag(s)
print(S)                                # Matrix with singular
                                        # values on the diagonal
[[ 2.05083035  0.          0.        ]
 [ 0.          0.64612715  0.        ]
 [ 0.          0.          0.35499201]
 [ 0.          0.          0.        ]]

print np.allclose(a, np.dot(U,np.dot(S,V)))
                                        # Check if U*S*V=a
#True

print(np.round(np.dot(U,U.T)))          # Check if U*U^T=I

[[ 1. -0. -0.  0.]
 [-0.  1. -0. -0.]
 [-0. -0.  1. -0.]
```

```
 [ 0. -0. -0.  1.]]

print(np.round(np.dot(V,V.T)))       # Check if V*V^T=I

[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [-0.  0.  1.]]
```

Now consider the same matrix in ArrayFire and compute its singular values.

```
#continuation
import arrayfire as af
a=a.astype('float32')
A=af.array(a)
U,S,V = af.svd(A)                    # SVD in ArrayFire

print(U)                             # Orthogonal matrix U

[[-0.49052709 -0.38722801  0.38825747 -0.67726952]
 [-0.34457594  0.60543764  0.65983015  0.28166839]
 [-0.65874141  0.37555152 -0.64320475 -0.10634266]
 [-0.45465451 -0.58520204  0.01296196  0.67131221]]

print(S)                             # Matrix with singular
                                     # values on the diagonal
[[ 2.05083013  0.          0.        ]
 [ 0.          0.64612722  0.        ]
 [ 0.          0.          0.35499203]
 [ 0.          0.          0.        ]]


print(af.singular_values(A))         # Singular values

[[ 2.05083036]
 [ 0.64612722]
 [ 0.35499203]]

print(V)                             # Orthogonal matrix V

[[-0.69376457 -0.45635003 -0.55716735]
 [-0.57329065  0.81818593  0.04370262]
 [-0.43592271 -0.34973809  0.8292495 ]]
```

```
print(af.max(af.abs(af.matmul(U,af.matmul(S,V))-A)))
                                      # Check if U*S*V=A
[[  2.98023224e-07]]

print(af.max(af.abs(af.matmul(U.T(),U)-af.identity(4,4))))
                                      # Check if U^T*U=I
[[  5.96046448e-07]]

print(af.max(af.abs(af.matmul(V.T(),V)-af.identity(3,3))))
                                      # Check if V^T*V=I
[[  3.57627869e-07]]
```

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

```
import arrayfire as af
A=af.randu(8000,6000)                 # 8000x6000 ArrayFire
af.tic()                              # matrix
SVD=af.svd(A)                         # SVD decomposition
af.eval(SVD)                          # in ArrayFire
af.sync()
t=af.toc()
print(str(t))
#12.183012                            # Time for SVD decomp.
                                      # on GTX 580
```

Below we present a corresponding numpy test.

```
import numpy as np
import time
a=np.random.rand(8000,6000)           # 8000x6000 double prec. matr.
t=time.time();ss=np.linalg.svd(a,compute_uv=0);print time.time()-t
#105.366261005                        # Time for numpy SVD decomp.
                                      # in double precision
                                      # on i7 2600k CPU
```

One can also use single precision function `sgesdd` from scipy.

```
import numpy as np
import time
a=np.random.rand(8000,6000).astype('float32') # Single precision
import scipy                          # 8000x6000 numpy matrix
```

```
from scipy import linalg
t=time.time();ss=scipy.linalg.lapack.flapack.sgesdd(a,compute_uv=0);
print time.time()-t
#67.9981679916                       # Time for scipy SVD decomp.
                                     # in single precision
                                     # on i7 2600k CPU
```

1.14 Pseudo-inverse

The pseudo-inverse is a generalization of the inverse matrix notion to general rectangular matrices. It is characterized as the unique matrix $B = A^+$, satisfying the following Moore–Penrose conditions:

$$A \cdot B \cdot A = A, \quad B \cdot A \cdot B = B, \quad (B \cdot A)^H = B \cdot A, \quad (A \cdot B)^H = A \cdot B,$$

If A has the SVD decomposition $A = U \cdot S \cdot V$ and S is diagonal with main diagonal $s = [s_0, s_1, \ldots, s_r, 0, \ldots, 0]$, $s_0, \ldots, s_r > 0$, then

$$A^+ = V^H \cdot S^+ \cdot U^H,$$

where S^+ is diagonal with main diagonal $s^+ = [1/s_0, 1/s_1, \ldots, 1/s_r, 0, \ldots, 0]$. Let us check that numpy and ArrayFire give "the same" (up to assumed precision) pseudo-inverses and the Moore–Penrose conditions are satisfied.

```
import numpy as np
a=np.random.rand(3,3)
pa=np.linalg.pinv(a)                 # Numpy pseudo-inverse
print(pa)

[[  0.32675503  -2.2330254    7.26878218]
 [ -9.15753474   5.91584956  -0.32785249]
 [ 10.05352215  -3.7262236   -5.06822522]]

import arrayfire as af
a=a.astype('float32')
A=af.array(a)
PA=af.pinv(A)                        # ArrayFire pseudo-inverse
print(PA)

[[  0.32675681  -2.23302579   7.26878166]
 [ -9.15753746   5.91585016  -0.32785228]
```

```
 [ 10.05352402  -3.72622395  -5.06822586]]

print(af.max(af.abs(af.matmul(A,af.matmul(PA,A))-A)))
[[  5.96046448e-08]]                  # Check if A*PA*A=A

print(af.max(af.abs(af.matmul(PA,af.matmul(A,PA))-PA)))
[[  1.90734863e-06]]                  # Check if PA*A*PA=PA

print(af.max(af.abs((af.matmul(A,PA)).H()-af.matmul(A,PA))))
[[  7.25220843e-08]]                  # Check if (A*PA)^H=A*PA

print(af.max(af.abs((af.matmul(PA,A)).H()-af.matmul(PA,A))))
[[  1.47361433e-07]]                  # Check if (PA*A)^H=PA*A
```

The following script compares the efficiency of pseudo-inverse operation in ArrayFire and numpy.

```
import numpy as np
import arrayfire as af
a=np.random.rand(6000,6000)           # Random 6000x6000 array
a=a.astype('float32')
A=af.array(a)
af.tic();PA=af.pinv(A);af.sync();print(af.toc())
                                      # ArrayFire pseudo-inverse
#0.99518                              # Pseudo-inverse time
                                      # in ArrayFire on GTX 580
import time
t1=time.time();pia=np.linalg.pinv(a);print time.time()-t1
                                      # Numpy pseudo-inverse
#91.9397661686                        # Pseudoinv. time in numpy
                                      # on i7 2600k CPU
```

1.15 Hessenberg decomposition

An upper Hessenberg matrix is a square matrix which has zero entries below the first subdiagonal:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1(n-2)} & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2(n-2)} & a_{2(n-1)} & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3(n-2)} & a_{3(n-1)} & a_{3n} \\ 0 & 0 & a_{43} & \cdots & a_{4(n-2)} & a_{4(n-1)} & a_{4n} \\ 0 & 0 & 0 & \cdots & a_{5(n-2)} & a_{5(n-1)} & a_{5n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

In Hessenberg decomposition the matrix A is represented in the form

$$A = U \cdot H \cdot U^H,$$

where U is unitary and H is an upper Hessenberg matrix.
Let us present the `hessenberg` functions from scipy and ArrayFire.

```
import arrayfire as af
import numpy as np
import scipy as sc
from scipy import linalg
a=np.random.rand(4,4)              # a: 4x4 random matrix
h,q=sc.linalg.hessenberg(a,calc_q=True)
                                   # Hessenberg decomp. in scipy
print np.allclose(a,np.dot(q,np.dot(h,q.T)))
                                   # Check if q*h*q^H=a
#True

a=a.astype('float32')
A=af.array(a)
H,U=af.hessenberg(A)               # Hessenberg dec. in ArrayFire
print(af.max(af.abs(af.matmul(U,af.matmul(H,U.H()))-A)))
                                   # Check if U*H*U^H=A
[[  1.78813934e-07]]

print(H)                           # The result from ArrayFire
```

```
[[ 0.28284356 -0.44367653  0.30509603  0.04981411]
 [-0.93112117  1.1645813  -0.78707689 -0.38981318]
 [ 0.         -0.86372834  0.45027709 -0.17600021]
 [ 0.          0.          0.51184952  0.42402357]]

print(h)                         # The result from scipy

[[ 0.28284355 -0.44367653  0.30509606  0.04981408]
 [-0.93112114  1.16458159 -0.78707698 -0.38981319]
 [ 0.         -0.8637285   0.45027715 -0.17600021]
 [ 0.          0.          0.51184942  0.42402348]]
```

Now let us compare the efficiency of these functions.

```
import arrayfire as af
import numpy as np
import scipy as sc
from scipy import linalg
import time
a=np.random.rand(6000,6000)      # 6000x6000 random matrix
a=a.astype('float32')
t1=time.time();h=sc.linalg.hessenberg(a,calc_q=False);
print time.time()-t1             # Hessenberg decomp. in scipy

#122.095136166                   # Decomp. time in scipy
                                 # on i7 2600k CPU
A=af.array(a)
af.tic();H=af.hessenberg(A);af.sync();print(af.toc())
                                 # Hessenberg decomp. in ArrayFire
#6.433709                        # Decomp. time in ArrayFire
                                 # on GTX 580
```

1.16 Plotting with ArrayFire

1.16.1 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 interactively as follows:

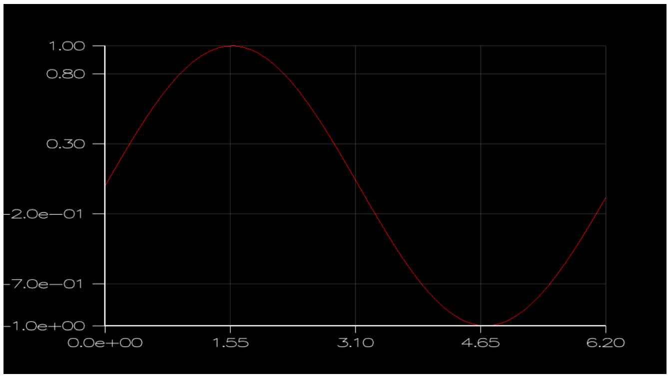

Figure 1.2: Plot of sine function in ArrayFire

```
import arrayfire as af
x=af.range(0,63)*0.1                # x=0,0.1,...,6.2
y=af.sin(x)                         # y=sin(x)
af.plot(y,x)                        # Two-dimensional plot
```

1.16.2 Three-dimensional plot

ArrayFire-Python allows also for three-dimensional plots. The corresponding function is `plot3d`. For example let us show how to plot

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

```
import arrayfire as af
x=af.range(-50,50)*0.1              # x=-5.0,-4.9,...,4.9
y=x
X,Y=af.grid(x,y)                    # 100x100 grid
f=af.cos(af.sqrt(af.pow(X,2)+af.pow(Y,2)))
af.plot3d(f)                        # Three-dimensional plot
```

1.16.3 Image-plot

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

```
import arrayfire as af
x=af.range(-50,50)*0.1              # x=-5.0,-4.9,...,4.9
y=x
```

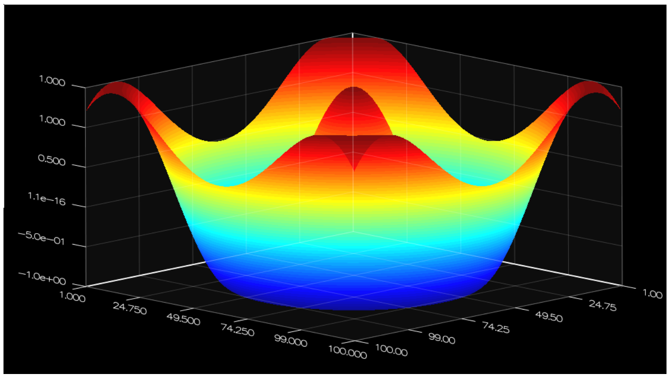

Figure 1.3: Three-dimensional plot in ArrayFire

```
X,Y=af.grid(x,y)                    # 100x100 grid
f=af.sqrt(af.pow(X,2)+af.pow(Y,2))
af.imgplot(f)                       # Image plot
```

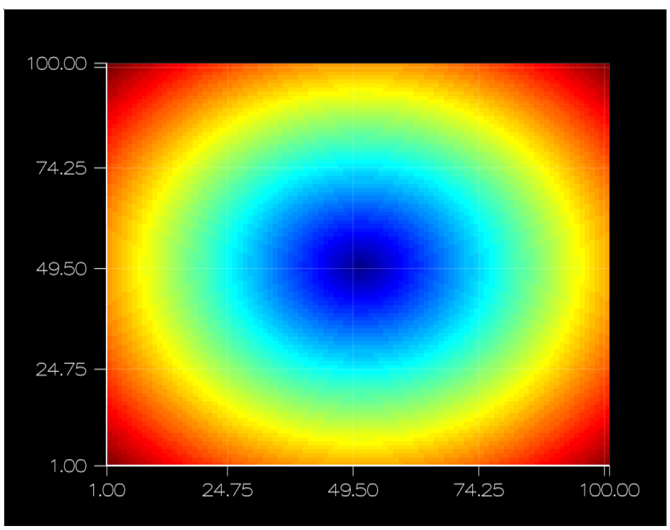

Figure 1.4: Image-plot in ArrayFire

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 `http://www.accelereyes.com/arrayfire/c/` and in the directory `.../arrayfire/doc/`.

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
                                       // software and hardware
  return 0;
}
/*
ArrayFire v1.1 (build c67f168) by AccelerEyes (64-bit Linux)
```

```
License Type: Concurrent Network (27000@server.accelereyes.com)
Addons: none
CUDA toolkit 4.1, driver 290.10
GPU0 GeForce GTX 580, 1536 MB, Compute 2.0 (single,double)
Display Device: GPU0 GeForce GTX 580
Memory Usage: 1445 MB free (1536 MB total)
*/
```

2.2 How to compile the examples

The examples from the present chapter can be easily compiled with the use of `Makefile` prepared by AccelerEyes developers. One can copy one of the directories from `../arrayfire/examples`, for example `template` into the same directory `../arrayfire/examples` using some new name, for example `matrices`. In the obtained directory one can create the file `info.cpp` containing the code from the previous section (or use `template.cpp`). In the file `../matrices/Makefile`, after `BIN :=` one can append the string `info` and save the `Makefile`. Assuming that the current working directory is `../arrayfire/examples/matrices` all we need to compile `info.cpp` is to issue `make` command:

```
$ make
g++  -m64 -Wall -Werror -I../../include -I/usr/local/cuda/include
-O2 -DNDEBUG -lafGFX -lrt -Wl,--no-as-needed -L../../lib64 -laf
-L/usr/local/cuda/lib64 -lcuda -lcudart -lpthread -lstdc++ -Wl,
-rpath,../../lib64,-rpath,/home/andy/arrayfire10may/arrayfire/lib64,
-rpath,/usr/local/cuda/lib64  info.cpp   -o info
```

To execute the obtained binary file `info` it suffices to issue the command

```
$ ./info
```

Remark. The users of the free version of ArrayFire need to have a working Internet connection to execute ArrayFire binaries.

2.3 Defining arrays

As in Python interface, the simplest way to define an array is to use `zeros`, `ones` or `identity` functions.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  array a0=zeros(3,3);                     // Zeros
  print(a0);
  array a1=ones(3,3);                      // Ones
  print(a1);
  array a2=identity(3,3);                  // Identity
  print(a2);
  print(ones(3,3,c32));                    // Complex version
  print(ones(3,3,u32));                    // Integer version
  return 0;}
/*
a0 =                                       // Zeros
        0.0000      0.0000      0.0000
        0.0000      0.0000      0.0000
        0.0000      0.0000      0.0000
a1 =                                       // Ones
        1.0000      1.0000      1.0000
        1.0000      1.0000      1.0000
        1.0000      1.0000      1.0000
a2 =                                       // Identity
        1.0000      0.0000      0.0000
        0.0000      1.0000      0.0000
        0.0000      0.0000      1.0000

ones(3,3,c32) =                            // Complex ones
        1.0000 + 0.0000i    1.0000 + 0.0000i    1.0000 + 0.0000i
        1.0000 + 0.0000i    1.0000 + 0.0000i    1.0000 + 0.0000i
        1.0000 + 0.0000i    1.0000 + 0.0000i    1.0000 + 0.0000i

ones(3,3,u32) =                            // Integer ones
        1     1     1
        1     1     1
        1     1     1
*/
```

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

```
f32 - float,  f64 - double,   c32 - cuComplex,
```

`c64 - cuDoubleComplex,   u32 - unsigned,   s32 int,   b8 - bool.`

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>
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
  //array a(3, 3, (float[]){0,1,2,3,4,5,6,7,8}); // shorter way
  print(a);                                  // from a host pointer
  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, afDevicePointer);
                                            // 3x3 f32 matrix
  print(b);                                 // from a device pointer
  return 0;
}
/*
a =                                         // from a host pointer
        0.0000      3.0000      6.0000
        1.0000      4.0000      7.0000
        2.0000      5.0000      8.0000
b =                                         // from a device pointer
        0.0000      3.0000      6.0000
        1.0000      4.0000      7.0000
        2.0000      5.0000      8.0000
*/
```

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

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

```
  array a=seq(0,8);                 // a=0,1,2,3,4,5,6,7,8
  print(a);
  array b=reshape(a,3,3);           // a as 3x3 array
  print(b);
  return 0;
}
/*
a =                                 // a as a column
        0.0000
        1.0000
        2.0000
        3.0000
        4.0000
        5.0000
        6.0000
        7.0000
        8.0000
b =                                 // a as a 3x3 matrix
           0.0000       3.0000       6.0000
           1.0000       4.0000       7.0000
           2.0000       5.0000       8.0000
*/
```

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[3*3];
//float *host_ptr=new float[3*3];
  for(int i=0;i<3;i++)              // Elements given by a formula
    for(int j=0;j<3;j++)
      host_ptr[3*i+j]=i-j;
  array a(3, 3, host_ptr);          // 3x3 f32 matrix
  print(a);                         // from host pointer
  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, afDevicePointer);
  print(b);                              // 3x3 f32 matrix
                                         // from device pointer
//delete [] host_ptr;
  return 0;
}
/*
a =                                      // From host pointer
        0.0000     1.0000     2.0000
       -1.0000     0.0000     1.0000
       -2.0000    -1.0000     0.0000

b =                                      // From device pointer
        0.0000     1.0000     2.0000
       -1.0000     0.0000     1.0000
       -2.0000    -1.0000     0.0000
*/
```

Here is an example with complex entries.

```
#include <stdio.h>
#include <arrayfire.h>
#include <complex.h>
#include "cuComplex.h"
using namespace af;
int main(void){
  cuComplex a[]={ {0.0f,1.0f},{2.0f,3.0f},{4.0f,5.0f},{6.0f,7.0f}};
  array b(2,2,a);                        // matrix with complex elem.
  print(b);
  return 0;
}
/*
b =                                      // complex matrix
        0.0000 + 1.0000i    4.0000 + 5.0000i
        2.0000 + 3.0000i    6.0000 + 7.0000i
*/
```

2.4 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
  print(A);
  A = randn(n, n, c32);               // Normally distr. 3x3 random
  printf("normal,complex:\n");        // matrix with complex entries
  print(A);
  return 0;
}
/*
uniform:                              // 3x3 random matrix,
A =                                   // uniform distribution
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

normal,complex:                       // 3x3 complex random matr.,
A =                                   // normally distributed
        0.2925-0.7184i    0.0083-0.2510i    0.5434-0.7168i
        0.1000-0.3932i    0.1290+0.3728i   -1.4913+1.4805i
        2.5470-0.0034i    1.0822-0.6650i    0.1374-1.2208i

*/
```

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
  print(floor(A));                    // Apply floor
  print(ceil(A));                     // Apply ceil
  print(round(A));                    // Apply round
  return 0;
```

```
}
/*
floor(A) =                                        // Floor
        1.0000    -2.0000     0.0000
       -4.0000    12.0000    -2.0000
        0.0000    -1.0000     0.0000

ceil(A) =                                         // Ceil
        2.0000    -1.0000     1.0000
       -3.0000    13.0000    -1.0000
        1.0000    -0.0000     1.0000

round(A) =                                        // Round
        1.0000    -2.0000     0.0000
       -4.0000    13.0000    -1.0000
        0.0000    -0.0000     1.0000
*/
```

Using 10000x10000 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 = 1e4;
  array A = randu(n, n);      // Warm-up
  timer::tic();
  A = randu(n, n);            // 10000x10000 uniform random matrix
  af::sync();                 // generation
  printf("uniform: %g\n", timer::toc());
  A = randn(n, n);            // Warm-up
  timer::tic();
  A = randn(n, n);            // 10000x10000 normal random matrix
  af::sync();                 // generation
  printf("normal: %g\n", timer::toc());
  return 0;
}
/*
uniform:  0.00967             // Time of uniform rand. matrix generation
                              // on GTX 580
```

```
normal:   0.017953         // Time of normal rand. matrix generation
                           // on GTX 580
*/
```

2.5 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);
  print(A);
  print(A.T());                           // Transposition
  A=randu(3,3,c64);
  print(A);
  print(conj(A));                         // Conjugation
  print(A.H());                           // Conjugate transposition
  return 0;
}
/*
A =                                       // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

A.T() =                                   // Transpose of A
        0.7402     0.9210     0.0390
        0.9690     0.9251     0.4464
        0.6673     0.1099     0.4702

A =                                       // Complex A
        0.0428+0.5084i    0.5790+0.3856i    0.0454+0.4146i
        0.6545+0.5126i    0.9082+0.6416i    0.0573+0.0816i
        0.2643+0.0520i    0.2834+0.6558i    0.1057+0.8006i

conj(A) =                                 // Conjugate
        0.0428-0.5084i    0.5790-0.3856i    0.0454-0.4146i
        0.6545-0.5126i    0.9082-0.6416i    0.0573-0.0816i
```

```
        0.2643-0.0520i    0.2834-0.6558i    0.1057-0.8006i

A.H() =                                     // Conjugate transpose
        0.0428-0.5084i    0.6545-0.5126i    0.2643-0.0520i
        0.5790-0.3856i    0.9082-0.6416i    0.2834-0.6558i
        0.0454-0.4146i    0.0573-0.0816i    0.1057-0.8006i
*/
```

One can flip the array horizontally or vertically.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  print(A);
  print(fliph(A));                          // Flip horizontally
  print(flipv(A));                          // Flip vertically
  return 0;
}
/*A =                                       // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

fliph(A) =                                  // Flip horizontally
        0.6673     0.9690     0.7402
        0.1099     0.9251     0.9210
        0.4702     0.4464     0.0390

flipv(A) =                                  // Flip vertically
        0.0390     0.4464     0.4702
        0.9210     0.9251     0.1099
        0.7402     0.9690     0.6673
*/
```

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);
  print(A);
  print(flat(A));                            // Flattened A
  print(lower(A));                           // Lower triang. part
  print(upper(A));                           // Upper triang. part
  return 0;
}
/*
A =                                          // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

flat(A) =                                    // A flattened
        0.7402
        0.9210
        0.0390
        0.9690
        0.9251
        0.4464
        0.6673
        0.1099
        0.4702

lower(A) =                                   // Lower triang. part
        0.7402     0.0000     0.0000
        0.9210     0.9251     0.0000
        0.0390     0.4464     0.4702

upper(A) =                                   // Upper triang. part
        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
```

```
  print(A.T());
  print(shift(A,1).T());                  // Shift operation
  print(rotate(A,3).T());                 // Rotate operation
  return 0;
}
/*
A.T() =
0.0000  1.0000  2.0000  3.0000  4.0000  5.0000  6.0000  7.0000  8.0000

shift(A,1).T() =
8.0000  0.0000  1.0000  2.0000  3.0000  4.0000  5.0000  6.0000  7.0000

rotate(A,3).T() =
0.0000  7.0000  6.0000  5.0000  4.0000  3.0000  2.0000  1.0000  0.0000
*/
```

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
  print(join(0,A1,A2));                  // Join vertically
  print(join(1,A1,A2));                  // Join horizontally
  A3=ones(3,3);                          // A3: 3x3 matrix of ones
  A4=zeros(3,3);                         // A4: 3x3 matrix of zeros
  print(join(0,A3,A4));                  // Join vertically
  print(join(1,A3,A4));                  // Join horizontally
  return 0;
}
/*
join(0,A1,A2) =                          // Join vertically
           1.0000
           2.0000
           3.0000
           4.0000
           5.0000
           6.0000

join(1,A1,A2) =                          // Join horizontally
```

```
          1.0000       4.0000
          2.0000       5.0000
          3.0000       6.0000

join(0,A1,A2) =                           // Join vertically
          1.0000       1.0000       1.0000
          1.0000       1.0000       1.0000
          1.0000       1.0000       1.0000
          0.0000       0.0000       0.0000
          0.0000       0.0000       0.0000
          0.0000       0.0000       0.0000

join(1,A1,A2) =                           // Join horizontally
     1.0000  1.0000  1.0000   0.0000   0.0000   0.0000
     1.0000  1.0000  1.0000   0.0000   0.0000   0.0000
     1.0000  1.0000  1.0000   0.0000   0.0000   0.0000
*/
```

2.6 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);                     // A: random 3x3 matrix
  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 10000x10000 matrix.

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

```
  int n = 1e4;
  array A=upper(ones(n,n,f32));          // A: 10000x10000 upper
                                         // Triang. matrix of ones
  float d = det<float>(A);               // Warm up
  timer::tic();
  d = det<float>(A);                     // Determinant of A
  af::sync();
  printf("det time: %g\n", timer::toc());
  printf("det value: %f\n",d);
  return 0;
}
//det time: 1.24086                      // det(A) computation time
                                         // on GTX 580
//det value: 1.000000                    // det(A) value
```

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

p-parameter	norm
`p=Inf`	$\max\lvert x_i\rvert$
`p=-Inf`	$\min\lvert x_i\rvert$
`p=nan (default)`	$(\sum\lvert x_i\rvert^2)^{1/2}$
`p` anything else	$(\sum\lvert x_i\rvert^p)^{1/p}$

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=seq(1,4);                      // A: [1,2,3,4]^T
  print(A);
  printf("Inf-norm,  max(abs(x_i)) : %f\n",norm<float>(A,Inf));
  printf("-Inf-norm,  min(abs(x_i)) : %f\n",norm<float>(A,-Inf));
  printf("1-norm, sum abs(x_i) : %f\n",norm<float>(A,1));
  printf("2-norm, (sum abs(x_i)^2)^(1/2) ): %f\n",norm<float>(A,2));
  printf("norm, 2-norm: %f\n",norm<float>(A));
  printf("5-norm, (sum abs(x_i)^5)^(1/5) : %f\n",norm<float>(A,5));
  return 0;
}
/*
A =                                      // A
        1.0000
```

```
        2.0000
        3.0000
        4.0000

Inf-norm,  max(abs(x_i)) : 4.000000          // Inf-norm
-Inf-norm,  min(abs(x_i)) : 1.000000         // -Inf-norm
1-norm, sum abs(x_i) : 10.000000             // 1-norm
2-norm, (sum abs(x_i)^2)^(1/2) ): 5.477226   // 2-norm
norm, 2-norm: 5.477226                       // norm
5-norm, (sum abs(x_i)^5)^(1/5) : 4.195548    // 5-norm
*/
```

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=randn(6,4);                    // A: random 6x4 matrix
  A(seq(3),span)=0;                      // First 3 rows set to 0
  unsigned r=rank(A);                    // Rank of A
  print(A);                              // A
  printf("rank: %d\n",r);                // print rank of A
  return 0;
}
/*
A =                                      // A
        0.0000     0.0000     0.0000     0.0000
        0.0000     0.0000     0.0000     0.0000
        0.0000     0.0000     0.0000     0.0000
       -0.3932     0.3728     1.4805    -0.7109
        2.5470     1.0822     0.1374    -1.2903
       -0.0034    -0.6650    -1.2208    -0.8822

rank: 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
```

The double precision calculations require ArrayFire-Pro.

2.7 Elementary arithmetic operations on matrices

In ArrayFire 1.1 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=seq(0,8);                     // A: 0,...,8
  array B=reshape(A,3,3);               // B: A as 3x3 matrix
  array I=identity(3,3);                // I: 3x3 identity matr.
  print(B);
  print(I);
  print(B+I);                           // Matrix addition
  print(B-I);                           // Matrix difference
  print(matmul(B,I));                   // Matrix multiplication
  return 0;
}
/*
B =                                     // B
        0.0000     3.0000     6.0000
        1.0000     4.0000     7.0000
        2.0000     5.0000     8.0000

I =                                     // I: identity matrix
        1.0000     0.0000     0.0000
```

```
        0.0000     1.0000     0.0000
        0.0000     0.0000     1.0000

B+I =                                   // B+I
        1.0000     3.0000     6.0000
        1.0000     5.0000     7.0000
        2.0000     5.0000     9.0000

B-I =                                   // B-I
       -1.0000     3.0000     6.0000
        1.0000     3.0000     7.0000
        2.0000     5.0000     7.0000


matmul(B,I) =                           // B*I
        0.0000     3.0000     6.0000
        1.0000     4.0000     7.0000
        2.0000     5.0000     8.0000
*/
```

The mul function 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=reshape(A,3,3);               // B: A as 3x3 matrix
  print(B);
  print(mul(B,B));                      // Element-wise product
  return 0;
}
/*
B =                                     // B
        0.0000     3.0000     6.0000
        1.0000     4.0000     7.0000
        2.0000     5.0000     8.0000

mul(B,B) =                              // Element-wise prod B times B
        0.0000     9.0000    36.0000
        1.0000    16.0000    49.0000
        4.0000    25.0000    64.0000
```

```
*/
```

The matrix power $A^n = A * \ldots * A$ (n-times) can be obtained using `matpow` function

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=reshape(seq(0,8),3,3);      // A: 0,...,8 as 3x3 matrix
  print(A);
  print(matmul(A,A));                 // A*A (for comparison)
  print(matpow(A,2.0f));              // Matrix power A^2
  return 0;
}
/*
A =
        0.0000     3.0000     6.0000
        1.0000     4.0000     7.0000
        2.0000     5.0000     8.0000

matmul(A,A) =                         // A*A
       15.0000    42.0000    69.0000
       18.0000    54.0000    90.0000
       21.0000    66.0000   111.0000

matpow(A,2.0f) =                      // A^2
       15.0000    42.0000    69.0000
       18.0000    54.0000    90.0000
       21.0000    66.0000   111.0000
*/
```

and the element-wise power, using the `pow` function.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=reshape(seq(0,8),3,3);      // A: 0,...,8 as 3x3 matrix
  print(A);
  print(pow(A,2.0f));                 // All elements are squared;
  print(pow(A,2*ones(3,3)));          // the second argument can
```

```
  return 0;                       // contain a matrix of exponents
}
/*
A =                                       // A
        0.0000     3.0000     6.0000
        1.0000     4.0000     7.0000
        2.0000     5.0000     8.0000

pow(A,2.0f) =                             // All elements are squared
        0.0000     9.0000    36.0000
        1.0000    16.0000    49.0000
        4.0000    25.0000    64.0000

pow(A,2*ones(3,3)) =                      // Exponents from 2*ones(3,3)
        0.0000     9.0000    36.0000
        1.0000    16.0000    49.0000
        4.0000    25.0000    64.0000
*/
```

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 = 1e4;
  array A = randu(n,n);                   // A,B 10000x10000 matrices
  array B = randu(n,n);
  timer::tic();
  array C = matmul(A,B);                  // Multiplication A*B
  af::sync();
  printf("multiplication time: %g\n", timer::toc());
  return 0;
}
//multiplication time: 1.98535          // Time on GTX 580
```

An analogous experiment with the powering function `matpow` gives:

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

```
int main(void){
  int n = 1e4;
  array A = randu(n,n);               // A: 10000x10000 matrix
  timer::tic();
  A = matpow(A,2.0f);                 // A^2
  af::sync();
  printf("powering time: %g\n", timer::toc());
  return 0;
}
//powering time: 1.9811              // Time on GTX 580
```

2.8 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=2*ones(3,3);
  float su,pro;
  print(A);                           // A
  print(sum(A));                      // Sums of columns
  print(sum(A,1));                    // Sums of rows
  su = sum<float>(A);                 // Sum of all elements
  printf("sum= %f\n",su);
  print(prod(A));                     // Products of columns
  print(prod(A,1));                   // Products of rows
  pro=prod<float>(A);                 // Product of all elements
  printf("product= %f\n",pro);
  return 0;
}
/*
A =                                   // A
        2.0000     2.0000     2.0000
        2.0000     2.0000     2.0000
        2.0000     2.0000     2.0000

sum(A) =
        6.0000     6.0000     6.0000  // Sums of columns
```

```
sum(A,1) =
        6.0000                        // Sums of rows
        6.0000
        6.0000

sum= 18.000000                        // Sum of all elements

prod(A) =
        8.0000     8.0000     8.0000  // Products of columns

prod(A,1) =
        8.0000                        // Products of rows
        8.0000
        8.0000

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=mul(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
```

2.9 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;
  float inn;
  array A1 = ones(m);                       // A1: 1,1,1
  array A2 = seq(m);                        // A2: 0,1,2
  inn = dot<float>(A1,A2);                  // Dot product of A1,A2
  print(A1);
  print(A2);
  printf("dot product: %f\n",inn);
  return 0;
}
/*
A1 =                                        // A1
        1.0000
        1.0000
        1.0000

A2 =                                        // A2
        0.0000
        1.0000
        2.0000

dot product: 3.000000                       // Inner product of A1,A2
*/
```

Now let us try to compute the dot product of A times A for 10000x10000 matrix of ones.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  unsigned m = 1e4;
  float inn;
```

```
  array A = ones(m,m);                // A: 10000x10000 matrix
                                      // of ones
  inn = dot<float>(A,A);              // Warm up
  timer::tic();
  inn = dot<float>(A,A);              // Dot product of A by A
  printf("dot prod. time: %g\n", timer::toc());
  printf("dot prod. value: %f\n",inn);
  return 0;
}
//dot prod. time:  0.005129            // Dot product comp. time
                                       // on GTX 580
//dot prod. value: 100000000.000000    // Dot product value
```

2.10 Mean, variance, standard deviation and histograms

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

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

```
mean(A) =
        1.0000     4.0000     7.0000          // Averages of columns

mean(A,1) =
        3.0000                                // Averages of rows
        4.0000
        5.0000

mean: 4.000000                                // Average of A

var(A) =
        1.0000     1.0000     1.0000          // Variances of columns

var(A,0,1) =
        9.0000                                // Variances of rows
        9.0000
        9.0000

var: 7.500000                                 // Variance of A

stdev(A) =
       1.0000      1.0000     1.0000          // Std. dev. of columns

stdev(A,0,1) =
           3.0000                             // Std. dev. of rows
           3.0000
           3.0000

stdev: 2.738613                               // Std. dev. of A

*/
```

Using larger arrays we can check the efficiency.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
  int n = 1e4;
  float a,v;
```

```
  array A = randn(n, n);                 // A: 10000x10000 matrix
  timer::tic();
  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::toc());
  printf("mean(A): %g\n", a);
  printf("var(A): %g\n", v);
//print(hi);                  // Redirect to a file and use gnuplot
  return 0;
}
//time: 0.039523                         // Time for mean, var, hist.
                                         // of A on GTX 580
//mean(A): 4.4911e-06                    // Theoretical mean: 0
//var(A): 1.00017                        // Theoretical variance: 1
```

2.11 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
    float er = norm<float>(matmul(A,X)-B,Inf);
    printf("max error: %g\n", er);       // Inf norm of A*X-B
    return 0;
}
// max error: 5.96046e-08
```

Using ArrayFire C/C++ we were able to solve 10000x10000 systems in one second in single precision on GTX 580 (the double precision computations require ArrayFire-Pro).

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
    int n = 1e4;
    array A = randu(n, n);            // A: 10000x10000 matrix
    array B = randu(n,1);             // B: 10000x1 vector
    timer::tic();
    array X = solve(A, B);            // Solving A*X=B
    af::sync();
    printf("solving time: %g\n", timer::toc());  // Time
    float er = norm<float>(matmul(A,X)-B,Inf);
    printf("max error: %g\n", er);    // Inf norm of A*X-B
    return 0;
//solving time: 1.11959               // Solving time on GTX 580
//max error: 0.0474624                // Inf norm of A*X-B
```

2.12 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=inv(A);                    // IA: inverse of A
  print(matmul(A,IA));                // Check if A*IA=I
  return 0;
}
/*
A*IA =                                // A*IA (should be I)
        1.0000    -0.0000    -0.0000
        0.0000     1.0000    -0.0000
       -0.0000     0.0000     1.0000
*/
```

To check the efficiency of ArrayFire `inv` let us try to invert a 7000x7000 random matrix

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(void){
    int n = 7e3;                        // 7000x7000 random matrix
    array A = randu(n, n);
    array I=identity(n,n);
    array IA = inv(A);                  // Warm up
    timer::tic();
    IA = inv(A);                        // Inverse matrix
    af::sync();
    printf("inverting time: %g\n", timer::toc());
    float er = max<float>(abs(matmul(A,IA) - I));
    printf("max error: %g\n", er);
    return 0;
}
/*
inverting time: 1.61757             // Inversion time on GTX 580
max error: 0.00194288               // Max error
*/
```

2.13 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.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main()
{
  int m = 3, n = 3,k;
  array L, U, p;
  array A = randu(m, n);              // A: random 3x3 matrix
```

```
  array LU = lu(A);             // Packed output
  lu(L, U, p, A);               // Unpacked output with permutation
  print(A);                     // A
  print(L);                     // L
  print(U);                     // U
  print(matmul(L,U));           // L*U
  print(LU);                    // L in lower, U in upper triangle
                                // Unit diagonal of L is not stored
  print(p);                     // Permutation
  array pA=zeros(m,n);
  for(k=0;k<m;k++)
    pA(k,span)=A(p(k),span);    // pA: permuted A
  print(pA);
  return 0;
}
/*
A =                                     // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

L =                                     // L
        1.0000     0.0000     0.0000
        0.0424     1.0000     0.0000
        0.8037     0.5536     1.0000

U =                                     // U
        0.9210     0.9251     0.1099
        0.0000     0.4072     0.4656
        0.0000     0.0000     0.3212

matmul(L,U) =                           // L*U
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702
        0.7402     0.9690     0.6673

LU =                                    // L in lower
        0.9210     0.9251     0.1099    // U in upper
        0.0424     0.4072     0.4656    // triangle
        0.8037     0.5536     0.3212
```

```
p =
        1.0000     2.0000     0.0000    // Permutation

pA =                                    // Permuted A
        0.9210     0.9251     0.1099    // equal to L*U
        0.0390     0.4464     0.4702
        0.7402     0.9690     0.6673
*/
```

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 = 1e4;
  array A = randu(n, n);                // 10000x10000 matrix
  timer::tic();
  array LU = lu(A);                     // LU, packed output
  af::sync();
  printf("LU decomp. time: %g\n", timer::toc());
  return 0;
}
//LU decomp. time: 0.951535             // LU decomposition time
                                        // on GTX 580
```

2.14 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.

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

```
int main(){
  unsigned info;
  array A=randu(3,3);
  A=A+A.T();                          // A symmetric and
  A=A+4*identity(3,3);                // positive definite
  array L=cholesky(info,A);           // Cholesky decomp. of A
  print(A);                           // A
  print(L);                           // L
  print(matmul(L.T(),L));             // A=L^T*L
  printf("%d\n",info);                // Check if decomposition
  return 0;                           // is successful
}
/*
A =                                   // A
        5.4804     1.8900     0.7063
        1.8900     5.8503     0.5563
        0.7063     0.5563     4.9404

L =                                   // L
        2.3410     0.8073     0.3017
        0.0000     2.2800     0.1371
        0.0000     0.0000     2.1979

matmul(L.T(),L) =                     // L^T*L (equal to A)
        5.4804     1.8900     0.7063
        1.8900     5.8503     0.5563
        0.7063     0.5563     4.9404

0                                     // Successful decomposition
*/
```

Now let us try a larger matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  unsigned info;
  int n = 1e4;
  array A = randu(n, n);
  A=A+A.T();                         // A: 10000x10000 symmetric
  A=A+100*identity(n,n);             // positive definite matrix
```

```
  timer::tic();
  array L = cholesky(info,A);           // Cholesky decomposition
  af::sync();
  printf("cholesky decomp. time: %g\n", timer::toc());
  printf("%d\n",info);
  return 0;
}
//cholesky decomp. time: 0.696417       // Decomposition time
                                        // on GTX 580
//0                                     // Successful decomposition
```

2.15 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.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 3;
  array Q, R;
  array A = randu(n, n);                // A: random 3x3 matrix
  array QR = qr(A);                     // QR, packed output
  qr(Q, R, A);                          // QR, unpacked output
  print(A);                             // A
  print(Q);                             // Q
  print(matmul(Q*Q.T()));               // Q*Q^T (should be equal to I)
  print(R);                             // R
  print(matmul(Q*R));                   // Q*R (should be equal to A)
  print(QR);                            // Packed output
  return 0;
}
/*
A =                                     // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
```

```
        0.0390      0.4464      0.4702

Q =                                       // Q
       -0.6261      0.2930     -0.7226
       -0.7790     -0.2743      0.5638
       -0.0330      0.9159      0.4000

matmul(Q,Q.T()) =                         // Q*Q^T  (=I)
        1.0000     -0.0000      0.0000
       -0.0000      1.0000     -0.0000
        0.0000     -0.0000      1.0000

R =                                       // R
       -1.1822     -1.3421     -0.5190
        0.0000      0.4390      0.5961
        0.0000      0.0000     -0.2321

matmul(Q,R) =                             // Q*R  (=A)
        0.7402      0.9690      0.6673
        0.9210      0.9251      0.1099
        0.0390      0.4464      0.4702

QR =                                      // Packed QR output
       -1.1822     -1.3421     -0.5190
        0.4791      0.4390      0.5961
        0.0203     -0.6432     -0.2321

*/
```

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 = 8e3;
  array Q, R;
  array A = randu(n, n);                  // Random 8000x8000 matrix
  timer::tic();
//  qr(Q,R,A);                            // Unpacked QR output
```

```
  array QR=qr(A);                        // Packed QR output
  af::sync();
  printf("qr time: %g\n", timer::toc());
  return 0;
}
//qr time: 2.82325                     // QR unpacked time on GTX 580
//qr time: 1.06279                     // QR packed time on GTX 580
```

2.16 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,V,s;
  array A = randu(m, n);                // Random 3x3 matrix
  s = svd(A);                           // Vector of singular values
  svd(S,U,V, A);                        // SVD:  A=U*S*V
  print(A);                             // A
  print(s);                             // Singular values s
  print(S);                             // Matrix with s on diagonal
  print(U);                             // Orthogonal matrix U
  print(V);                             // Orthogonal matrix V
  print(matmul(U,matmul(S,V)));         // Check if U*S*V=A
  print(matmul(U,U.T()));               // Check orthogonality of U
  print(matmul(V,V.T()));               // Check orthogonality of V
  return 0;
}
/*
A =                                     // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
```

```
        0.0390     0.4464     0.4702

s =                                    // Vector of singular values
        1.9311
        0.5739
        0.1087

S =                                    // Matrix with singular values
        1.9311     0.0000     0.0000   // on diagonal
        0.0000     0.5739     0.0000
        0.0000     0.0000     0.1087

U =                                    // Orthogonal matrix U
       -0.7120     0.3365    -0.6163
       -0.6502    -0.6475     0.3975
       -0.2653     0.6837     0.6798

V =                                    // Orthogonal matrix V
       -0.5884    -0.7301    -0.3476
       -0.5586     0.0561     0.8275
       -0.5846     0.6811    -0.4409

matmul(U,matmul(S,V)) =                // Check if U*S*V=A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

matmul(U,U.T()) =                      // Orthogonality of U
        1.0000    -0.0000     0.0000
       -0.0000     1.0000    -0.0000
        0.0000    -0.0000     1.0000

matmul(V,V.T()) =                      // Orthogonality of V
        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 = 8e3;
  array U,S,V,s;                     // U,V - orthogonal matrices
                                     // S - matrix with singul. val.
                                     // s - vector of singul. val.
  array A = randu(n, n);             // 8000x8000 random matrix
  timer::tic();
//  svd(S,U,V,A);                    // SVD, computing S,U,V
  s=svd(A);                          // SVD, computing only s
  af::sync();
  printf("SVD time: %g\n", timer::toc());
  return 0;
}
//SVD time: 69.2596                  // SVD time, computing S,U,V
//SVD time: 19.1714                  // SVD time, computing only s
                                     // on GTX 580
```

2.17 Pseudo-inverse

Pseudo-inverse of A is the unique matrix $B = A^+$, satisfying the following Moore–Penrose conditions:

$$A \cdot B \cdot A = A, \quad B \cdot A \cdot B = B, \quad (B \cdot A)^H = B \cdot A, \quad (A \cdot B)^H = A \cdot B.$$

Let us check that ArrayFire `pinv` function satisfies these conditions.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  array A=randu(3,3);                     // A: random 3x3 matrix
  array PA=pinv(A);                       // PA: pseudo-inverse of A
  float er = norm<float>(matmul(A,matmul(PA,A)) - A,Inf);
  printf("error0: %g\n", er);             // check if A*PA*A=A
  er = norm<float>(matmul(PA,matmul(A,PA)) - PA,Inf);
  printf("error1: %g\n", er);             // Check if PA*A*PA=PA
  er = norm<float>(matmul(A,PA).H() - matmul(A,PA),Inf);
  printf("error2: %g\n", er);             // Check if (A*PA)^H=A*PA
  er = norm<float>(matmul(PA,A).H() - matmul(PA,A),Inf);
  printf("error3: %g\n", er);             // Check if (PA*A)^H=PA*A
```

```
  return 0;
}
/*
error0: 5.96046e-08
error1: 1.90735e-06
error2: 2.0972e-07
error3: 1.79432e-07
*/
```

Now let us try a bigger example.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 7e3;
  array A = randu(n, n);              // A: 7000x7000 random matrix
  timer::tic();
  array PA=pinv(A);                   // Pseudo-inverse of A
  af::sync();
  printf("Pseudo-inverse time: %g\n", timer::toc());
  return 0;
}
//Pseudoinverse time: 1.52192         // Pseudo-inverse time
                                      // on GTX 580
```

2.18 Hessenberg decomposition

Let us recall that an upper Hessenberg matrix is a square matrix which has zero entries below the first sub-diagonal:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1(n-2)} & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2(n-2)} & a_{2(n-1)} & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3(n-2)} & a_{3(n-1)} & a_{3n} \\ 0 & 0 & a_{43} & \cdots & a_{4(n-2)} & a_{4(n-1)} & a_{4n} \\ 0 & 0 & 0 & \cdots & a_{5(n-2)} & a_{5(n-1)} & a_{5n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

In Hessenberg decomposition the matrix A is represented in the form

$$A = U \cdot H \cdot U^H,$$

where U is unitary and H is an upper Hessenberg matrix.
Let us begin with a small example illustrating ArrayFire `hessenberg` function.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int m = 3, n = 3;
  array U,H,h;
  array A = randu(m, n);              // A: random 3x3 matrix
  h = hessenberg(A);                  // h: Hessenberg form of A
  hessenberg(H,U,A);                  // Hessenberg decomposition
  print(A);                           // A=U*H*U^H
  print(h);                           // Output from abbrev. version
  print(H);                           // Output from full version
  print(U);                           // Orthogonal matrix U
  print(matmul(U,matmul(H,U.H())));   // Check if U*H*U^H=A
  print(matmul(U,U.H()));             // Check orthogonality of U
  return 0;
}
/*
A =                                   // A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

h =                                   // Output from abbrev. version
        0.7402    -0.9963     0.6257
       -0.9218     0.9479    -0.0897
        0.0000    -0.4261     0.4475

H =                                   // Output from full version
        0.7402    -0.9963     0.6257
       -0.9218     0.9479    -0.0897
        0.0000    -0.4261     0.4475
```

```
U =                                  // Orthogonal matrix U
        1.0000     0.0000     0.0000
        0.0000    -0.9991    -0.0423
        0.0000    -0.0423     0.9991

matmul(U,matmul(H,U.H())) =          // Check if U*H*U^H=A
        0.7402     0.9690     0.6673
        0.9210     0.9251     0.1099
        0.0390     0.4464     0.4702

matmul(U,U.H()) =                    // Check if U*U^H=I
        1.0000     0.0000     0.0000
        0.0000     1.0000    -0.0000
        0.0000    -0.0000     1.0000
*/
```

Now let us try a 10000x10000 random matrix.

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  int n = 1e4;
  array U,H,h;
  array A = randu(n, n);             // A: 10000x10000 matrix
  hessenberg(H,U,A);                 // Hesenberg decomp. of A
//  h=hessenberg(A);                 // Hessenberg form of A
  af::sync();
  printf("Hessenberg time: %g\n", timer::toc());
  return 0;
}
//Hessenberg time: 24.7794           // Hessenberg decomp. time
//Hessenberg time: 21.5178           // Hessenberg form time
                                     // on GTX 580
```

2.19 Plotting with ArrayFire

2.19.1 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:

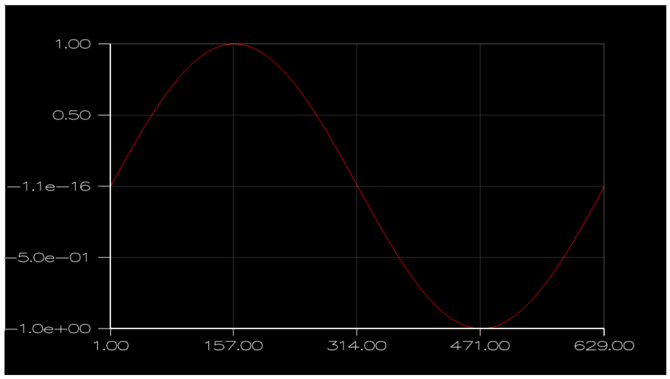

Figure 2.1: Plot of sine function in ArrayFire

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  seq x(0,0.01,6.28);                  // x=0,0.01,...,6.28
  array y=sin(x);                      // y=sin(x)
  plot(y);                             // Two-dimensional plot
  getchar();
  return 0;
}
```

2.19.2 Three-dimensional plot

ArrayFire-C/C++ allows also for three-dimensional plots. The corresponding function is `plot3d`. For example let us show how to plot

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

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  seq x(-5,0.01,5);                    // x=-5.0,-4.99,...,5.0
  array X,Y;
  grid(X,Y,x,x);                       // 1001x1001 grid
  array XX=mul(X,X), YY=mul(Y,Y);
  array Z=cos(sqrt(XX+YY));
```

```
  plot3d(Z);                           // Three-dimensional plot
  getchar();
  return 0;
}
```

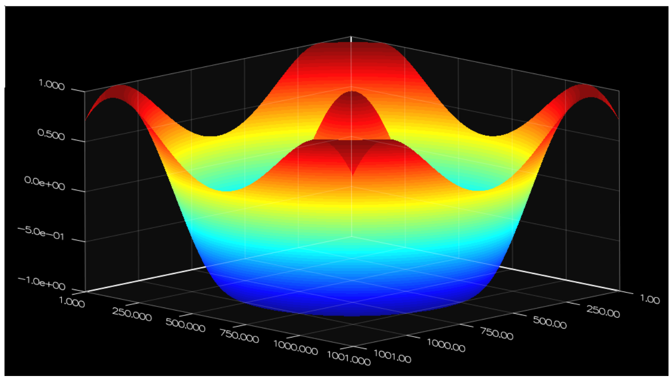

Figure 2.2: Three-dimensional plot in ArrayFire

2.19.3 Image-plot

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

```
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main(){
  seq x(-5,0.01,5);                    // x=-5.0,-4.99,...,5.0
  array X,Y;
  grid(X,Y,x,x);                       // 1001x1001 grid
  array XX=mul(X,X), YY=mul(Y,Y);
  array Z=cos(sqrt(XX+YY));
  imgplot(Z);                          // Image-plot
  getchar();
  return 0;
}
```

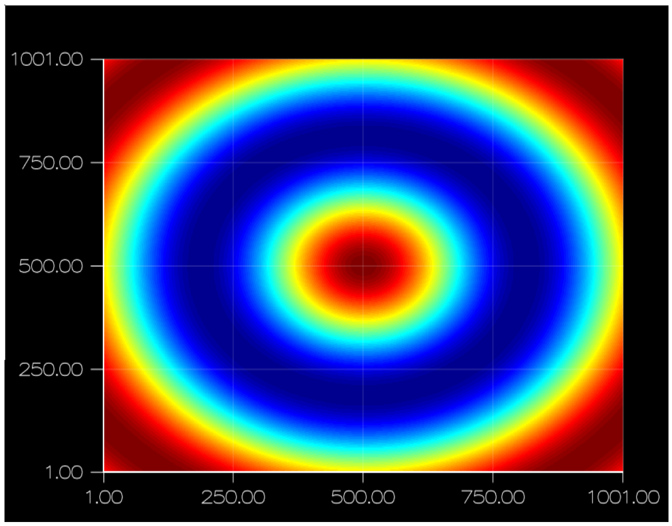

Figure 2.3: Image-plot in ArrayFire