NumericExtensions.jl Documentation

Release 0.2.16

Dahua Lin

July 29, 2015

1 Package Overview **3**
- 1.1 Motivating example . . . 3
- 1.2 Main features . . . 4

2 Functors **5**
- 2.1 Motivating example . . . 5
- 2.2 Functor types . . . 6
- 2.3 Pre-defined functors . . . 6

3 Mapping **9**
- 3.1 General usage . . . 9
- 3.2 Additional functions . . . 9
- 3.3 Pre-defined mapping functions . . . 10
- 3.4 Performance . . . 10

4 Vector Broadcasting **13**

5 Reduction **15**
- 5.1 Full reduction . . . 15
- 5.2 Reduction along dimensions . . . 15
- 5.3 Basic reduction functions . . . 16
- 5.4 Derived reduction functions . . . 18
- 5.5 Weighted Sum . . . 20
- 5.6 Performance . . . 21

6 Vector Norms and Normalization **23**
- 6.1 Vector Norm Evaluation . . . 23
- 6.2 Normalization . . . 23

7 Array Scanning **25**
- 7.1 Generic Scanning . . . 25
- 7.2 Common Scanning Functions . . . 26

8 Fast Shared-memory Views **27**

9 Utilities for Data Manipulation **29**

10 Positive Definite Matrices **31**
- 10.1 Positive definite matrix types . . . 31

10.2 Common interface 31

NumericExtensions.jl is a Julia package that provides high performance support of numerical computation. This package is an extension of the Julia Base – part of the material may be migrated into the Base in future.

NumericExtensions.jl provides a wide range of tools, which include:

- higher order functions for mapping, reduction, and map-reduce operation that takes typed functors to achieve performance comparable to hand-crafted loops.
- Functions that allow inplace updating and writing results to pre-allocated arrays for mapping, reduction, and map-reduce operations.
- Convenient functions for inplace vectorized computation.
- Vector broadcasting.
- Fast views for operating on contiguous blocks/slices.

Contents:

CHAPTER 1

Package Overview

Julia provides a fantastic technical computing environment that allows you to write codes that are both performant and generic. However, as it is still at its early stage, some functions are not as performant as they can be and writing computational algorithms directly based on builtin functions may not give you the best performance. This package provides you with a variety of tools to address such issues.

1.1 Motivating example

To see how this package may help you, let's first consider a simple example, that is, to compute the sum of squared difference between two vectors. This can be easily done in Julia in one line as follows

```
r = sum(abs2(x - y))
```

Whereas this is simple, this expression involves some unnecessary operations that would lead to suboptimal performance: (1) it creates two temporary arrays respectively to store `x - y` and `abs(x - y)`, (2) it completes the computation through three passes over the data – computing `x - y`, computing `abs2(x - y)`, and finally computing the sum. Julia provides a `mapreduce` function which allows you to complete the operation in a single pass without creating any temporaries:

```
r = mapreduce((x, y) -> abs2(x - y), +, x, y)
```

However, if you really run this you may probably find that this is even slower. The culprit here is that the anonymous function `(x, y) -> abs2(x - y)` is not lined, which will be resolved and called at each iteration. Therefore, to compute this efficiently, one has to write loops as below

```
s = 0.
for i = 1 : length(x)
    s += abs2(x[i] - y[i])
end
```

This is not too bad though, until you have more complex needs, e.g. computing this along each row/column of the matrix. Then writing the loops can become more involved, and it is more tricky to implement it in a cache-friendly way.

With this package, we can compute this efficiently without writing loops, as

```
r = mapdiff_reduce(Abs2(), Add(), x, y)

# or more concise:
r = sum_fdiff(Abs2(), x, y)
```

```
# to compute this along a specific dimension
r = sum_fdiff(Abs2(), x, y, dim)
```

Here, `Abs2` and `Add` are *typed functors* provided by this package, which, unlike normal functions, can still be properly inlined with passed into a higher order function (thus causing zero overhead). This package extends `map`, `reduce`, and `mapreduce` to accept typed functors and as well introduces additional high order functions like `mapdiff`, `mapdiff_reduce`, `sum_fdiff` etc to simplify the usage in common cases.

Benchmark shows that writing in this way is over *8x faster* than `sum(abs2(x - y))`.

This package also provides a collection of specific functions to directly support very common computation. For this particular example, you can write `sqdiffsum(x, y)`, where `sqdiffsum` is one of such functions provided here.

1.2 Main features

Main features of this package are highlighted below:

- Pre-defined functors that cover most typical mathematical computation;
- A easy way for user to define customized functors;
- Extended/specialized methods for `map`, `map!`, `reduce`, and `mapreduce`. These methods are carefully optimized, which often result in *2x - 10x* speed up;
- Additional functions such as `map1!`, `reduce!`, and `mapreduce!` that allow inplace updating or writing results to preallocated arrays;
- Vector broadcasting computation (supporting both inplace updating and writing results to new arrays).
- Fast shared-memory views of arrays.

Since many of the methods are extensions of base functions. Simply adding a statement `using NumericExtensions` is often enough for substantial performance improvement. Consider the following code snippet:

```
using NumericExtensions

x = rand(1000, 1000)
r = sum(x, 2)
```

Here, when adding the statement `using NumericExtensions` *transparently replace* the method provided in the Base module by the specialized method in *NumericExtensions*. As a consequence, the statement `r = sum(x, 2)` becomes *6x* faster. Using additional functions provided by this package can further improve the performance.

CHAPTER 2

Functors

Passing functions as arguments are essential in writing generic algorithms. However, function arguments do not get inlined in Julia (at current version), usually resulting in suboptimal performance.

2.1 Motivating example

Passing functions as arguments are essential in writing generic algorithms. However, function arguments do not get inlined in Julia (at current version), usually resulting in suboptimal performance. Consider the following example:

```
plus(x, y) = x + y
map_plus(x, y) = map(plus, x, y)

a = rand(1000, 1000)
b = rand(1000, 1000)

 # warming up and get map_plus compiled
a + b
map_plus(a, b)

 # benchmark
@time for i in 1 : 10 map_plus(a, b) end  # -- statement (1)
@time for i in 1 : 10 a + b end           # -- statement (2)
```

Run this script in you computer, you will find that statement (1) is over *20+ times* slower than statement (2). The reason is that the function argument `plus` is resolved and called at each iteration of the inner loop within the `map` function.

This package addresses this issue through *type functors* (*i.e.* function-like objects of specific types) and a set of highly optimized higher level functions for mapping and reduction. The codes above can be rewritten as

```
using NumericExtensions

 # benchmark
@time for i in 1 : 10 map(Add(), a, b) end     # -- statement(1)
@time for i in 1 : 10 a + b end                # -- statement(2)
```

Here, using a typed functor `Add` statement (1) is *10%* faster than statement (2) in my benchmark.

2.2 Functor types

`Functor` is the abstract base type for all functors, which are formally defined as below

```
abstract Functor{N}  # N: the number of arguments

typealias UnaryFunctor Functor{1}
typealias BinaryFunctor Functor{2}
typealias TernaryFunctor Functor{3}
```

Below is an example that shows how to define a functor that computes the squared difference:

```
type SqrDiff <: BinaryFunctor end

NumericExtensions.evaluate(::SqrDiff, x, y) = abs2(x - y)
NumericExtensions.result_type(::SqrDiff, t1::Type, t2::Type) = promote_type(t1, t2)
```

To define multiple functors, it would be more concise to first import `evaluate` and `result_type` before extending them, as follows:

```
import NumericExtensions.evaluate, NumericExtensions.result_type

type SqrDiff <: BinaryFunctor end

evaluate(::SqrDiff, x, y) = abs2(x - y)
result_type(::SqrDiff, t1::Type, t2::Type) = promote_type(t1, t2)
```

Note: Higher order functions such as `map` and `reduce` rely on the `result_type` method to determine the element type of the result. This is necessary, as Julia does not provide a generic mechanism to acquire the return type of a method.

2.3 Pre-defined functors

NumericExtensions.jl has defined a series of functors as listed below:

- Arithmetic functors: `Add`, `Subtract`, `Multiply`, `Divide`, `Negate`, `Abs`
- Max and Min functors: `Max`, `Min`
- Rounding functors: `Floor`, `Ceil`, `Round`, `Trunc`
- Power functors: `Pow`, `Sqrt`, `Cbrt`, `Abs2`, `Hypot`
- Exp and log functors: `Exp`, `Exp2`, `Exp10`, `Log`, `Log2`, `Log10`, `Expm1`, `Log1p`
- Trigonometric functors: `Sin`, `Cos`, `Tan`, `Asin`, `Acos`, `Atan`, `Atan2`
- Hyperbolic functors: `Sinh`, `Cosh`, `Tanh`, `Asinh`, `Acosh`, `Atanh`
- Error functors: `Erf`, `Erfc`
- Gamma functors: `Gamma`, `Lgamma`, `Digamma`
- Comparison functors: `Greater`, `GreaterEqual`, `Less`, `LessEqual`, `Equal`, `NotEqual`
- Number class functors: `Isfinite`, `Isinf`, `Isnan`
- Fused multiply and add: `FMA` (i.e. `(a, b, c) -> a + b * c`)
- Others: `Logit`, `Logistic`, `Xlogx`, `Xlogy`

Except for several functors that corresponding to operators, most functors are named using the capitalized version of the corresponding math function. Therefore, you don't have to look up this list to find the names. The collection of pre-defined functors will be extended in future. Please refer to `src/functors.jl` for the most updated list.

CHAPTER 3

Mapping

NumericExtensions.jl extends `map` and `map!` to accept functors for efficient element-wise mapping:

3.1 General usage

Synopsis:

Let `f1`, `f2`, and `f3` be respectively unary, binary, and ternary functors. Generic usage of `map` and `map!` is summarized as follows:

```
map(f1, x)
map(f2, x1, x2)
map(f3, x1, x2, x3)

map!(f1, dst, x)
map!(f2, dst, x1, x2)
map!(f3, dst, x1, x2, x3)
```

Here, `map` creates and returns the resultant array, while `map!` writes results to a pre-allocated `dst` and returns it. Each argument can be either an array or a scalar number. At least one argument should be an array, and all array arguments should have compatible sizes.

Examples:

```
map(Abs(), x)           # returns abs(x)
map(FMA(), x, y, z)     # returns x + y .* z
map!(Add(), dst, x, 2)  # writes x + 2 to dst
```

3.2 Additional functions

NumericExtensions.jl provides additional functions (`map1!`, `mapdiff`, and `mapdiff!`) to simplify common use:

Synopsis

`map1!` updates the first argument inplace with the results, `mapdiff` maps a functor to the difference between two arguments, and `mapdiff!` writes the results of `mapdiff` to a pre-allocated array.

```
map1!(f1, x1)             # x1 <-- f1(x1)
map1!(f2, x1, x2)         # x1 <-- f2(x1, x2)
map1!(f3, x1, x2, x3)     # x1 <-- f3(x1, x2, x3)

```

```
mapdiff(f1, x, y)          # returns f1(x - y)
mapdiff!(f1, dst, x, y)    # dst <-- f1(x - y)
```

Here, `x1` (*i.e.* the first argument to `map1!` must be an array, while `x2` and `x3` can be either an array or a number).

Note that `mapdiff` and `mapdiff!` uses an efficient implementation, which completes the computation in one-pass and never creates the intermediate array `x - y`.

Examples

```
map1!(Mul(), x, 2)         # multiply x by 2 (inplace)
mapdiff(Abs2(), x, y)      # compute squared differences between x and y
mapdiff(Abs(), x, 1)       # compute |x - 1|
```

3.3 Pre-defined mapping functions

Julia already provides vectorized function for most math computations. In this package, we additionally define several functions for vectorized inplace computation (based on `map!`), as follows

```
add!(x, y)          # x <- x + y
subtract!(x, y)     # x <- x - y
multiply!(x, y)     # x <- x .* y
divide!(x, y)       # x <- x ./ y
negate!(x)          # x <- -x
pow!(x, y)          # x <- x .^ y

abs!(x)             # x <- abs(x)
abs2!(x)            # x <- abs2(x)
rcp!(x)             # x <- 1 ./ x
sqrt!(x)            # x <- sqrt(x)
exp!(x)             # x <- exp(x)
log!(x)             # x <- log(x)

floor!(x)           # x <- floor(x)
ceil!(x)            # x <- ceil(x)
round!(x)           # x <- round(x)
trunc!(x)           # x <- trunc(x)
```

In the codes above, `x` must be an array (*i.e.* an instance of `AbstractArray`), while `y` can be either an array or a scalar.

In addition, this package also define some useful functions using compound functos:

```
absdiff(x, y)       # abs(x - y)
sqrdiff(x, y)       # abs2(x - y)
fma(x, y, c)        # x + y .* c, where c can be array or scalar
fma!(x, y, c)       # x <- x + y .* c
```

3.4 Performance

For simple functions, such as `x + y` or `exp(x)`, the performance of the map version such as `map(Add(), x, y)` and `map(Exp(), x)` is comparable to the Julia counter part. However, `map` can accelerate computation considerably in a variety of cases:

- When the result storage has been allocated (e.g. in iterative updating algorithms) or you want inplace update, then `map!` or the pre-defined inplace computation function can be used to avoid unnecessary memory allocation/garbage collection, which can sometimes be the performance killer.
- When the inner copy contains two or multiple steps, `map` and `map!` can complete the computation in one-pass without creating intermediate arrays, usually resulting in about `2x` or even more speed up. Benchmark shows that `absdiff(x, y)` and `sqrdiff(x, y)` are about *2.2x* faster than `abs(x - y)` and `abs2(x - y)`.
- The script `test/benchmark_map.jl` runs a series of benchmarks to compare the performance `map` and the Julia vectorized expressions for a variety of computation.

Vector Broadcasting

Julia has very nice and performant functions for broadcasting: `broadcast` and `broadcast!`, which however does not work with functors. For customized computation, you have to pass in function argument, which would lead to severe performance degradation. *NumericExtensions.jl* provides `vbroadcast` and `vbroadcast!` to address this issue.

Synopsis

```
vbroadcast(f, x, y, dim)  # apply a vector y to vectors of x along a specific dimension
vbroadcast!(f, dst, x, y, dim)  # write results to dst
vbroadcast1!(f, x, y)            # update x with the results
```

Here, `f` is a binary functor, and `x` and `y` are arrays such that `length(y) == size(x, dim)`.

Examples

```
vbroadcast(f, x, y, 1)    # r[:,i] = f(x[:,i], y) for each i
vbroadcast(f, x, y, 2)    # r[i,:] = f(x[i,:], y) for each i
vbroadcast(f, x, y, 3)    # r[i,j,:] = f(x[i,j,:], y) for each i, j

vbroadcast1!(Add(), x, y, 1)   # x[:,i] += y[:,i] for each i
vbroadcast1!(Mul(), x, y, 2)   # x[i,:] .*= y[i,:] for each i
```

For cubes, it supports computation along two dimensions

```
x = rand(2, 3, 4)
y = rand(2, 3)

vbroadcast(x, y, (1, 2))    # this adds y to each page of x
```

Difference from `broadcast`

Unlike `broadcast`, you have to specify the vector dimension for `vbroadcast`. The benefit is two-fold: (1) the overhead of figuring out broadcasting shape information is elimintated; (2) the shape of `y` can be flexible.

```
x = rand(5, 6)
y = rand(6)

vbroadcast(Add(), x, y, 2)    # this adds y to each row of x
broadcast(+, x, reshape(y, 1, 6))  # with broadcast, you have to first reshape y into a row
```

CHAPTER 5

Reduction

A key advantage of this package are highly optimized reduction and map-reduction functions, which sometimes lead to over `10x` speed up.

5.1 Full reduction

Synopsis

This package extends `reduce` and `mapreduce`, and additionally provides `mapdiff_reduce` for generic reduction and map-reduction.

Let `f1`, `f2`, and `f3` be respectively unary, binary, and ternary functors, and `op` be an binary functor. The general usage of these extended methods is summarized below:

```
reduce(op, x)    # reduction using op to combine values

mapreduce(f1, op, x)              # reduction using op to combine terms as f1(x)
mapreduce(f2, op, x1, x2)         # reduction using op to combine terms as f2(x1, x2)
mapreduce(f3, op, x1, x2, x3)     # reduction using op to combine terms as f3(x1, x2, x3)

mapdiff_reduce(f2, op, x, y)      # reduction using op to combine terms as f2(x - y)
```

Examples

```
mapreduce(Abs2(), Add(), x)           # compute the sum of squared of x (i.e. sum(abs2(x)))
mapreduce(Multiply(), Add(), x, y)    # compute the dot product between x, y
mapdiff_reduce(Abs2(), Max(), x, y)   # compute the maximum squared difference between x and y
```

5.2 Reduction along dimensions

The extended `reduce` and `mapreduce` and the additional `mapdiff_reduce` also allow reduction along specific dimension(s):

Synopsis

```
reduce(op, x, dims)

mapreduce(f1, op, x, dims)
mapreduce(f2, op, x1, x2, dims)
mapreduce(f3, op, x1, x2, x3, dims)
```

```
mapdiff_reduce(f2, op, x, y, dims)
```

Here, `dims` can be either an integer to specify arbitrary dimension, or a pair of integers such as `(1, 2)` for reduction along two dimensions.

When `dims` is a pair of integers such as `(1, 2)` or `(2, 3)`, each argument must be either a cube or a scalar. We believe this has covered most usage in practice. That being said, we will try to support cases where `dims` can be an arbitrary tuple in the future.

The package additionally provides `reduce!`, `mapreduce!`, and `mapdiff_reduce!`, which allow to write the results of reduction/map-reduction along dimensions to pre-allocated arrays:

```
reduce!(dst, op, x, dims)

mapreduce!(dst, f1, op, x1)
mapreduce!(dst, f2, op, x1, x2, dims)
mapreduce!(dst, f3, op, x1, x2, x3, dims)

mapdiff_reduce!(dst, f2, op, x, y, dims)
```

Examples

```
reduce(Add(), x, 1)       # sum x along columns
reduce(Add(), x, 2)       # sum x along rows

reduce(Add(), x, (1, 2))    # sum each page of x
reduce(Add(), x, (1, 3))    # sum along both the first and the third dimension

mapreduce(Abs(), Max(), x, 1)    # compute maximum absolute value along each column
mapreduce(Sqr(), Add(), x, 2)    # compute sum square along each row

mapdiff_reduce(Abs(), Min(), x, y, (1, 2))  # compute minimum absolute difference
                                            # between x and y for each page
```

5.3 Basic reduction functions

The package extends/specializes `sum`, `mean`, `max`, and `min`, and additionally provides `sum!`, `mean!`, `max!`, and `min!`, as follows

The funtion `sum` and its variant forms:

```
sum(x)
sum(f1, x)            # compute sum of f1(x)
sum(f2, x1, x2)       # compute sum of f2(x1, x2)
sum(f3, x1, x2, x3)   # compute sum of f3(x1, x2, x3)

sum(x, dims)
sum(f1, x, dims)
sum(f2, x1, x2, dims)
sum(f3, x1, x2, x3, dims)

sum!(dst, x, dims)    # write results to dst
sum!(dst, f1, x1, dims)
sum!(dst, f2, x1, x2, dims)
sum!(dst, f3, x1, x2, x3, dims)

sumfdiff(f2, x, y)     # compute sum of f2(x - y)
sumfdiff(f2, x, y, dims)
sumfdiff!(dst, f2, x, y, dims)
```

The funtion `mean` and its variant forms:

```
mean(x)
mean(f1, x)             # compute mean of f1(x)
mean(f2, x1, x2)        # compute mean of f2(x1, x2)
mean(f3, x1, x2, x3)    # compute mean of f3(x1, x2, x3)

mean(x, dims)
mean(f1, x, dims)
mean(f2, x1, x2, dims)
mean(f3, x1, x2, x3, dims)

mean!(dst, x, dims)     # write results to dst
mean!(dst, f1, x1, dims)
mean!(dst, f2, x1, x2, dims)
mean!(dst, f3, x1, x2, x3, dims)

meanfdiff(f2, x, y)      # compute mean of f2(x - y)
meanfdiff(f2, x, y, dims)
meanfdiff!(dst, f2, x, y, dims)
```

The function `max` and its variants:

```
max(x)
max(f1, x)             # compute maximum of f1(x)
max(f2, x1, x2)        # compute maximum of f2(x1, x2)
max(f3, x1, x2, x3)    # compute maximum of f3(x1, x2, x3)

max(x, (), dims)
max(f1, x, dims)
max(f2, x1, x2, dims)
max(f3, x1, x2, x3, dims)

max!(dst, x, dims)     # write results to dst
max!(dst, f1, x1, dims)
max!(dst, f2, x1, x2, dims)
max!(dst, f3, x1, x2, x3, dims)

maxfdiff(f2, x, y)      # compute maximum of f2(x - y)
maxfdiff(f2, x, y, dims)
maxfdiff!(dst, f2, x, y, dims)
```

The function `min` and its variants

```
min(x)
min(f1, x)             # compute minimum of f1(x)
min(f2, x1, x2)        # compute minimum of f2(x1, x2)
min(f3, x1, x2, x3)    # compute minimum of f3(x1, x2, x3)

min(x, (), dims)
min(f1, x, dims)
min(f2, x1, x2, dims)
min(f3, x1, x2, x3, dims)

min!(dst, x, dims)     # write results to dst
min!(dst, f1, x1, dims)
```

```
min!(dst, f2, x1, x2, dims)
min!(dst, f3, x1, x2, x3, dims)

minfdiff(f2, x, y)      # compute minimum of f2(x - y)
minfdiff(f2, x, y, dims)
minfdiff!(dst, f2, x, y, dims)
```

Note: when computing maximum/minimum along specific dimension, we use `max(x, (), dims)` and `min(x, (), dims)` instead of `max(x, dims)` and `min(x, dims)` to avoid ambiguities that would otherwise occur.

5.4 Derived reduction functions

In addition to these basic reduction functions, we also define a set of derived reduction functions, as follows:

```
var(x)
var(x, dim)
var!(dst, x, dim)

std(x)
std(x, dim)
std!(dst, x, dim)

sumabs(x)   # == sum(abs(x))
sumabs(x, dims)
sumabs!(dst, x, dims)

meanabs(x)    # == mean(abs(x))
meanabs(x, dims)
meanabs!(dst, x, dims)

maxabs(x)    # == max(abs(x))
maxabs(x, dims)
maxabs!(dst, x, dims)

minabs(x)    # == min(abs(x))
minabs(x, dims)
minabs!(dst, x, dims)

sumsq(x)   # == sum(abs2(x))
sumsq(x, dims)
sumsq!(dst, x, dims)

meansq(x)   # == mean(abs2(x))
meansq(x, dims)
meansq!(dst, x, dims)

dot(x, y)   # == sum(x .* y)
dot(x, y, dims)
dot!(dst, x, y, dims)

sumabsdiff(x, y)    # == sum(abs(x - y))
sumabsdiff(x, y, dims)
sumabsdiff!(dst, x, y, dims)

meanabsdiff(x, y)    # == mean(abs(x - y))
meanabsdiff(x, y, dims)
```

```
meanabsdiff!(dst, x, y, dims)

maxabsdiff(x, y)   # == max(abs(x - y))
maxabsdiff(x, y, dims)
maxabsdiff!(dst, x, y, dims)

minabsdiff(x, y)   # == min(abs(x - y))
minabsdiff(x, y, dims)
minabsdiff!(dst, x, y, dims)

sumsqdiff(x, y)  # == sum(abs2(x - y))
sumsqdiff(x, y, dims)
sumsqdiff!(dst, x, y, dims)

meansqdiff(x, y)  # == mean(abs2(x - y))
meansqdiff(x, y, dims)
meansqdiff!(dst, x, y, dims)
```

Although this is quite a large set of functions, the actual code is quite concise, as most of such functions are generated through macros (see `src/reduce.jl`)

In addition to the common reduction functions, this package also provides a set of statistics functions that are particularly useful in probabilistic or information theoretical computation, as follows

```
sumxlogx(x)  # == sum(xlogx(x)) with xlog(x) = x > 0 ? x * log(x) : 0
sumxlogx(x, dims)
sumxlogx!(dst, x, dims)

sumxlogy(x, y)  # == sum(xlog(x,y)) with xlogy(x,y) = x > 0 ? x * log(y) : 0
sumxlogy(x, y, dims)
sumxlogy!(dst, x, y, dims)

entropy(x)   # == - sumxlogx(x)
entropy(x, dims)
entropy!(dst, x, dims)

logsumexp(x)   # == log(sum(exp(x)))
logsumexp(x, dim)
logsumexp!(dst, x, dim)

softmax!(dst, x)    # dst[i] = exp(x[i]) / sum(exp(x))
softmax(x)
softmax!(dst, x, dim)
softmax(x, dim)
```

For `logsumexp` and `softmax`, special care is taken to ensure numerical stability for large x values, that is, their values will be properly shifted during computation.

5.5 Weighted Sum

Computation of weighted sum as below is common in practice.

$$\sum_{i=1}^{n} w_i x_i$$

$$\sum_{i=1}^{n} w_i f(x_i, \ldots)$$

$$\sum_{i=1}^{n} w_i f(x_i - y_i)$$

NumericExtensions.jl directly supports such computation via `wsum` and `wsumfdiff`:

```
wsum(w, x)                  # weighted sum of x with weights w
wsum(w, f1, x1)             # weighted sum of f1(x1) with weights w
wsum(w, f2, x1, x2)         # weighted sum of f2(x1, x2) with weights w
wsum(w, f3, x1, x2, x3)     # weighted sum of f3(x1, x2, x3) with weights w
wsumfdiff(w, f2, x, y)     # weighted sum of f2(x - y) with weights w
```

These functions also support computing the weighted sums along a specific dimension:

```
wsum(w, x, dim)
wsum!(dst, w, x, dim)

wsum(w, f1, x1, dim)
wsum!(dst, w, f1, x1, dim)

wsum(w, f2, x1, x2, dim)
wsum!(dst, w, f2, x1, x2, dim)

wsum(w, f3, x1, x2, x3, dim)
wsum!(dst, w, f3, x1, x2, x3, dim)

wsumfdiff(w, f2, x, y, dim)
wsumfdiff!(dst, w, f2, x, y, dim)
```

Furthermore, `wsumabs`, `wsumabsdiff`, `wsumsq`, `wsumsqdiff` are provided to compute weighted sum of absolute values / squares to simplify common use:

```
wsumabs(w, x)               # weighted sum of abs(x)
wsumabs(w, x, dim)
wsumabs!(dst, w, x, dim)

wsumabsdiff(w, x, y)        # weighted sum of abs(x - y)
wsumabsdiff(w, x, y, dim)
wsumabsdiff!(dst, w, x, y, dim)

wsumsq(w, x)              # weighted sum of abs2(x)
wsumsq(w, x, dim)
wsumsq!(dst, w, x, dim)

wsumsqdiff(w, x, y)       # weighted sum of abs2(x - y)
wsumsqdiff(w, x, y, dim)
wsumsqdiff!(dst, w, x, y, dim)
```

5.6 Performance

The reduction and map-reduction functions are carefully optimized. In particular, several tricks lead to performance improvement:

- computation is performed in a cache-friendly manner;
- computation completes in a single pass without creating intermediate arrays;
- kernels are inlined via the use of typed functors;
- inner loops use linear indexing (with pre-computed offset);
- opportunities of using BLAS are exploited.

Generally, many of the reduction functions in this package can achieve *3x - 10x* speed up as compared to the typical Julia expression.

We observe further speed up for certain functions: * full reduction with `sumabs`, `sumsq`, and `dot` utilize BLAS level 1 routines, and they achieve *10x* to *30x* speed up. * For `var` and `std`, we devise dedicated procedures, where computational steps are very carefully scheduled such that most computation is conducted in a single pass. This results in about *25x* speedup.

Vector Norms and Normalization

This package provides functions for evaluating vector norms and normalizing vectors.

6.1 Vector Norm Evaluation

Synopsis

```
vnorm(x, p)             # compute L-p norm of vec(x)
vnorm(x, p, dim)        # compute L-p norm of x along dimension dim
vnorm!(r, x, p, dim)    # compute L-p norm along specific dimension and
                        # write results to r

vnormdiff(x, y, p)              # compute L-p norm of vec(x - y)
vnormdiff(x, y, p, dim)         # compute L-p norm of x - y along dim
vnormdiff!(r, x, y, p, dim)     # compute L-p norm of x - y along dim and write to r
```

Notes:

- For `vnormdiff` and `vnormdiff!`, `x` or `y` can be either an array or a scalar.
- When `p` is 1, 2, or Inf, specialized fast routines are used.

Examples

```
vnorm(x, 2)           # compute L-2 norm of x
vnorm(x, 2, 1)        # compute L-2 norm of each column of x
vnorm(x, Inf, 2)      # compute L-inf norm of each row of x
vnorm!(r, x, 2, 1)    # compute L-2 norm of each column, and write results to r

vnormdiff(x, 2.5, 2)    # compute L-2 norm of x - 2.5
vnormdiff(x, y, 1, 2)   # compute L-1 norm of x - y for each column
```

6.2 Normalization

Normalizing a vector w.r.t L-p norm means to scale a vector such that the L-p norm of the vector becomes 1.

Synopsis

```
normalize(x, p)         # returns a normalized vector w.r.t. L-p norm
normalize!(x, p)        # normalize x w.r.t. L-p norm inplace
normalize!(r, x, p)     # write the normalized vector to a pre-allocated array r
```

```
normalize(x, p, dim)        # returns an array comprised of normalized vectors along dim
normalize!(x, p, dim)       # normalize vectors of x along dim inplace w.r.t. L-p norm
normalize!(r, x, p, dim)    # write the normalized vectors along dim to r
```

CHAPTER 7

Array Scanning

Scanning refers to the operation that applies a binary function recursively as follows:

Let `x` be an input vector and `f` be a binary operator, then the output `y` is a vector of the same length given by

$$y[1] = x[1]$$
$$y[i] = f(y[i-1], x[i]), \forall i = 2, 3, \ldots$$

For example, the function `cumsum` is a special case of array scanning.

This package provides both generic scanning functions, such as `scan`, `scan!`, `mapscan`, and `mapscan!`, as well as specialized scanning functions, such as `cumsum` and `cumsum!`, etc.

7.1 Generic Scanning

Synopsis

Let `op` be a binary operator, `f1`, `f2`, and `f3` be respectively unary, binary, and ternary functors.

```
scan(op, x)                 # scan x using operator op
mapscan(op, f1, x)          # scan f1(x)
mapscan(op, f2, x, y)       # scan f2(x, y)
mapscan(op, f3, x, y, z)    # scan f3(x, y, z)

scan!(r, op, x)             # write the results of scanning of x to r
mapscan!(r, op, f1, x)
mapscan!(r, op, f2, x, y)
mapscan!(r, op, f3, x, y, z)

scan!(op, x)                # inplace scanning of x using operator op
```

Both `scan` and `scan!` accepts a dimension argument for scanning vectors along a specific dimension.

```
scan(op, x, dim)                # scan vectors of x along dim using operator op
mapscan(op, f1, x, dim)         # scan vectors of f1(x) along dim
mapscan(op, f2, x, y, dim)      # scan vectors of f2(x, y) along dim
mapscan(op, f3, x, y, z, dim)   # scan vectors of f3(x, y, z) along dim

scan!(r, op, x, dim)            # write the results of scanning of x to r
mapscan!(r, op, f1, x, dim)
mapscan!(r, op, f2, x, y, dim)
mapscan!(r, op, f3, x, y, z, dim)
```

```
scan!(op, x, dim)                # inplace scanning of x along dim
```

Note: `mapscan` and `mapscan!` use efficient ways to scan the terms without creating a temporary array such as `f1(x)`. Hence, `mapscan(op, f1, x)` is generally faster than `scan(op, f(x))`.

Examples

```
scan(Add(), x)          # equivalent to cumsum(x)
scan(Add(), x, 1)       # equivalent to cumsum(x, 1)
scan!(Add(), x)         # writing cumsum(x) inplace (i.e. back to x)
```

7.2 Common Scanning Functions

For some common scanning operations, we provide specific functions to simplify the use.

```
# cumsum uses Add() as scanning operator

cumsum(x)
cumsum!(r, x)
cumsum!(x)

cumsum(f1, x)
cumsum(f2, x, y)
cumsum(f3, x, y, z)

cumsum!(r, f1, x)
cumsum!(r, f2, x, y)
cumsum!(r, f3, x, y, z)

# cummax uses Max() as scanning operator

cummax(x)
cummax!(r, x)
cummax!(x)

cummax(f1, x)
cummax(f2, x, y)
cummax(f3, x, y, z)

cummax!(r, f1, x)
cummax!(r, f2, x, y)
cummax!(r, f3, x, y, z)

# cummin uses Min() as scanning operator

cummin(x)
cummin!(r, x)
cummin!(x)

cummin(f1, x)
cummin(f2, x, y)
cummin(f3, x, y, z)

cummin!(r, f1, x)
cummin!(r, f2, x, y)
cummin!(r, f3, x, y, z)
```

CHAPTER 8

Fast Shared-memory Views

Getting a slice/part of an array is common in numerical computation. Julia itself provides two ways to do this: reference (e.g. `x[:, J]`) and the `sub` function (e.g. `sub(x, :, J)`). Both have performance issues: the former makes a copy each time you call it, while the latter results in an `SubArray` instance. Despite that `sub` does not create a copy, accessing elements of a `SubArray` instance is usually very slow (with current implementation).

NumericExtensions.jl addresses this problem by providing a `unsafe_view` function, which returns a view of specific part of an array. Below is a list of valid usage:

```
unsafe_view(a)

unsafe_view(a, :)
unsafe_view(a, i0:i1)

unsafe_view(a, :, :)
unsafe_view(a, :, j)
unsafe_view(a, i0:i1, j)
unsafe_view(a, :, j0:j1)

unsafe_view(a, :, j, k)
unsafe_view(a, i0:i1, j, k)
unsafe_view(a, :, :, k)
unsafe_view(a, :, j0:j1, k)

unsafe_view(a, :, :, :)
unsafe_view(a, :, :, k0:k1)
```

Benchmark shows that using `unsafe_view` often increases the throughput of element accessing by *50%*.

Notes

- `unsafe_view` only applies to the case when the part being referenced is contiguous.
- `unsafe_view` only does not maintain reference to the source array (instead, it relies on pointers) and does not perform bounds checking. Please use it with caution. Preferrably, it should be used within a context where you can ensure that the source array is existent and the indices are correct.

CHAPTER 9

Utilities for Data Manipulation

This package provides some useful functions for data manipulation:

eachrepeat (*x*, *rt*)

Repeats each element in x for rt times. Three cases are supported:

- `x` is a vector, and `rt` is an integer
- `x` is a vector, and `rt` is a vector of the same size
- `x` is a matrix, and `rt` is a pair as (p, q)

Examples:

```
eachrepeat(1:3, 2)    # ==> [1,1,2,2,3,3]
eachrepeat([1,4,5],[2,3,1])   #==> [1,1,4,4,4,5]

eachrepeat([1 2 3; 4 5 6], (3, 2))
#==> [1 1 2 2 3 3;
#     1 1 2 2 3 3;
#     1 1 2 2 3 3;
#     4 4 5 5 6 6;
#     4 4 5 5 6 6;
#     4 4 5 5 6 6]
```

sortindexes (*x*[, *k*])

This is similar to `sortperm`. But it only applies to a sequence of elements, whose values are in `1:k`. This function implements an algorithm of complexity `O(n)`, which is faster than `sortperm`.

This function returns a pair of two outputs:

- `sinds`: the sorted indexes, i.e. `sortperm(x)`.
- `cnts`: the counts of occurrences of each value in `1:k`, i.e. `cnts[i] == sum(x .== i)`.

Examples:

```
x = [1, 1, 1, 2, 2, 3, 3, 3, 3, 1, 1, 2, 1]
s, c = sortindexes(x, k)
# s == sortperm(x) == [1, 2, 3, 10, 11, 13, 4, 5, 12, 6, 7, 8, 9]
# c == [6, 3, 4]
```

Node: The second argument `k` may be omitted, in such cases, it is equivalent to `sortindexes(x, max(x))`.

sortindexes!(x, sinds, cnts)

Perform same functionality as `sortindexes`, but write the results to pre-allocated arrays.

groupindexes (*x*[, *k*])

Group indexes according to the integers in `x`, which can take values in `1:k`. This function returns a vector of index vectors (say `r`), such that `r[i] == find(x .== i)`.

This function relies on `sortindexes` internally, and its complexity is `O(n)`.

Examples:

```
julia> x = [1, 1, 1, 2, 2, 3, 3, 3, 3, 1, 1, 2, 1];
julia> groupindexes(x)
3-element Array{Array{Int64,1},1}:
[1,2,3,10,11,13]
[4,5,12]
[6,7,8,9]
```

Node: The second argument `k` may be omitted, in such cases, it is equivalent to `groupindexes(x, max(x))`.

CHAPTER 10

Positive Definite Matrices

Positive definite matrices are widely used in machine learning and probabilistic modeling, especially in applications related to graph analysis and Gaussian models. It is not uncommon that positive definite matrices used in practice have special structures (e.g. diagonal), which can be exploited to accelerate computation.

NumericExtensions.jl supports efficient computation on positive definite matrices of various structures. In particular, it provides uniform interfaces to use positive definite matrices of various structures for writing generic algorithms, while ensuring that the most efficient implementation is used in actual computation.

10.1 Positive definite matrix types

This package defines an abstract type `AbstractPDMat` to capture positive definite matrices of various structures, as well as three concrete sub-types: `PDMat`, `PDiagMat`, `ScalMat`, which can be constructed as follows

- `PDMat`: representing a normal positive definite matrix in its full matrix form. **Construction:** `PDMat(C)`.
- `PDiagMat`: representing a positive diagonal matrix. **Construction:** `PDiagMat(v)`, where `v` is the vector of diagonal elements.
- `ScalMat(d, v)`: representing a scaling matrix of the form `v * eye(d)`. **Construction:** `ScalMat(d, v)`, where `d` is the matrix dimension (the size of the matrix is `d x d`), and `v` is a scalar value.

Notes: Compact representation is used internally. For example, an instance of `PDiagMat` only contains a vector of diagonal elements instead of the full diagonal matrix, and `ScalMat` only contains a scalar value. While, for `PDMat`, a Cholesky factorization is computed and contained in the instance for efficient computation.

10.2 Common interface

Functions are defined to operate on positive definite matrices through a uniform interface. In the description below, We let `a` be a positive definite matrix, *i.e* an instance of a subtype of `AbstractPDMat`, `x` be a column vector or a matrix, and `c` be a positive scalar.

`dim`(*a*)
    Return the dimension of the matrix. If it is a `d x d` matrix, this returns `d`.

`full`(*a*)
    Return a copy of the matrix in full form.

`logdet`(*a*)
    Return the log-determinant of the matrix.

diag (*a*)
Return a vector of diangonal elements.

a * x
Perform matrix-vector/matrix-matrix multiplication.

a \ x
Solve linear equation, equivalent to `inv(a) * x`, but implemented in a more efficient way.

a * c, c * a
Scalar product, multiply `a` with a scalar `c`.

unwhiten (*a, x*)
Unwhitening transform.

If `x` satisfies the standard Gaussian distribution, then `unwhiten(a, x)` has a distribution of covariance `a`.

whiten (*a, x*)
Whitening transform.

If `x` satisfies a distributiion of covariance `a`, then the covariance of `whiten(a, x)` is the identity matrix.

Note: `whiten` and `unwhiten` are mutually inverse operations.

unwhiten!(a, x)
Inplace unwhitening, `x` will be updated.

whiten!(a, x)
Inplace whitening, `x` will be updated.

quad (*a, x*)
Compute `x' * a * x` in an efficient way. Here, `x` can be a vector or a matrix.

If `x` is a vector, it returns a scalar value. If `x` is a matrix, is performs column-wise computation and returns a vector `r`, such that `r[i]` is `x[:,i]' * a * x[:,i]`.

invquad (*a, x*)
Compute `x' * inv(a) * x` in an efficient way (without computing `inv(a)`). Here, `x` can be a vector or a matrix (for column-wise computation).

quad!(r, a, x)
Inplace column-wise computation of `quad` on a matrix `x`.

invquad!(r, a, x)
Inplace column-wise computation of `invquad` on a matrix `x`.

X_A_Xt (*a, x*)
Computes `x * a * x'` for matrix `x`.

Xt_A_X (*a, x*)
Computes `x' * a * x` for matrix `x`.

X_invA_Xt (*a, x*)
Computes `x * inv(a) * x'` for matrix `x`.

Xt_invA_X (*a, x*)
Computes `x' * inv(a) * x` for matrix `x`.

a1 + a2
Add two positive definite matrices (promoted to a proper type).

a + x
Add a positive definite matrix and an ordinary square matrix (returns an ordinary matrix).

`add!(x, a)`
Add the positive definite matrix `a` to an ordinary matrix `m` (inplace).

`add_scal!(x, a, c)`
Add `a * c` to an ordinary matrix `x` (inplace).

`add_scal`(*a1*, *a2*, *c*)
Return `a1 + a2 * c` (promoted to a proper type).

Note: Specialized version of each of these functions are implemented for each specific postive matrix types using the most efficient routine (depending on the corresponding structures.)

Index

A

add_scal() (built-in function), 33

D

diag() (built-in function), 31
dim() (built-in function), 31

E

eachrepeat() (built-in function), 29

F

full() (built-in function), 31

G

groupindexes() (built-in function), 29

I

invquad() (built-in function), 32

L

logdet() (built-in function), 31

Q

quad() (built-in function), 32

S

sortindexes() (built-in function), 29

U

unwhiten() (built-in function), 32

W

whiten() (built-in function), 32

X

X_A_Xt() (built-in function), 32
X_invA_Xt() (built-in function), 32
Xt_A_X() (built-in function), 32
Xt_invA_X() (built-in function), 32