Time-series processing methods for high-dimensional time-series: `runstats` R package

3rd webinar OSS developers in physical behavior field

Marta Karas

Nov 5, 2019

Outline

- Fast time-series processing
  - Rolling statistics
  - Speed-up rolling mean/sd/var with 1-liner trick
  - Speed-up rolling cor/cov with convolution theorem
- `runstats` R package
  - CRAN: https://cran.r-project.org/web/packages/runstats/index.html
  - GitHub: https://github.com/martakarass/runstats (considered in this presentation)

Fast time-series processing: motivation

Recall: raw accelerometry data is voluminous

- Example: raw accelerometry data collected from **1 patient, 1 week**, frequency=100Hz yields 3 * 100 * 60 * 60 * 24 * 7 = **181,440,000** float values

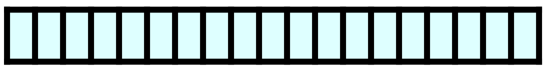
Some often used operations:

- Smoothing (e.g. running window average)
- Running variance, running correlation (with some short signal)

must be done fast

Example 1: running window average (running mean)

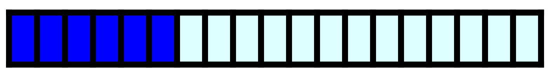
Input:

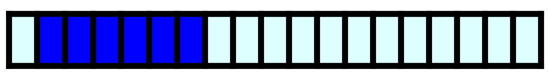


vector `x: len(x) = N`

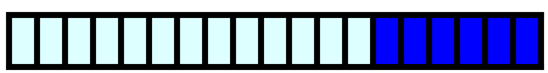
(window length) scalar `win_n`

Output:

`out[1]` mean()

`out[2]` mean()

⋮

`out[N-n+1]` mean()

Simple R is not fast: running window average

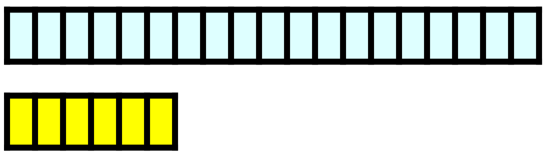
```
## Running window average of a time-series
RunningMean.sapply <- function(x, win_n){
  l_x <- length(x)
  sapply(1:(l_x - win_n + 1), function(i){
    mean(x[i:(i + win_n - 1)])
  })
}

N <- 10000000  # 10,000,000
x <- runif(N)
win_n <- 100

system.time({
  RunningMean.sapply(x, win_n)
})
#   user  system elapsed
# 75.880   3.545  79.678
```

~18h of fs=100Hz 1-dimensional time-series

~ 1.25 minute of execution

Example 2: running correlation

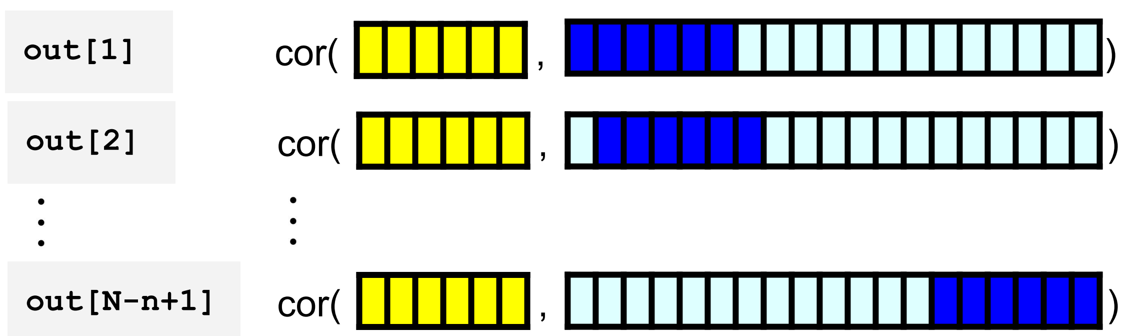
Input:



vector **x**: `len(x) = N`

vector **y**: `len(y) = n, n<N`

Output:



Simple R is not fast: running correlation

```
## Running covariance of long time-series x and short(er) y
RunningCor.sapply <- function(x, y){
  l_x <- length(x)
  l_y <- length(y)
  sapply(1:(l_x - l_y + 1), function(i){
    cor(x[i:(i+l_y-1)], y)
  })
}

N <- 10000000  # 10,000,000
n <- 100
x <- runif(N)
y <- runif(n)

system.time({
  RunningCor.sapply(x, y)
})
#    user  system elapsed
# 516.994   2.554 519.946
```

~18h of fs=100Hz 1-dimensional time-series

~ 8.5 minutes of execution

Outline

- Fast time-series processing
  - Rolling statistics
  - **Speed-up rolling mean/sd/var with 1-liner trick**
  - Speed-up rolling cor/cov with convolution theorem
- `runstats` R package
  - CRAN: https://cran.r-project.org/web/packages/runstats/index.html
  - GitHub: https://github.com/martakarass/runstats (considered in this presentation)

1-liner trick implemented in `runstats` R package

Goal: compute `x` vector running average over moving window of length `W`

```
runningMean(x, W){

   diff(c(0, cumsum(x)), lag = W) / W

}
```

Acknowledgement: this piece is the most recent improvement contributed by **Lacey Etzkorn** (PhD student at JHU Biostat); previously it had been previously implemented also via FFT.

`runstats` R package: running window average

```
## Running window average of a time-series
RunningMean.sapply <- function(x, win_n){
  l_x <- length(x)
  sapply(1:(l_x - win_n + 1), function(i){
    mean(x[i:(i + win_n - 1)])
  })
}

N <- 10000000  # 10,000,000
x <- runif(N)
win_n <- 100

system.time({
  RunningMean.sapply(x, win_n)
})
#   user  system elapsed
# 75.880   3.545  79.678

system.time({
  runstats::RunningMean(x, win_n)
})
#  user  system elapsed
# 0.216   0.019   0.237
```

~18h of fs=100Hz 1-dimensional time-series

~ 1.25 minute of execution

~ 0.2 seconds of execution (~350x faster)

Outline

- Fast time-series processing
  - Rolling statistics
  - Speed-up rolling mean/sd/var with 1-liner trick
  - **Speed-up rolling cor/cov with convolution theorem**
- `runstats` R package
  - CRAN: https://cran.r-project.org/web/packages/runstats/index.html
  - GitHub: https://github.com/martakarass/runstats (considered in this presentation)

Speed-up computing with convolution theorem [1/]

Convolution
Convolution is a mathematical operation on two functions, denote f and g, defined as the integral of the product of the two functions after one is reversed and shifted:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau. \tag{1}$$

Discrete convolution
For functions x, h defined on the set $\mathbb{Z}$ of integers, the discrete convolution of x and h is given by

$$(x * h)[n] = \sum_{i=-\infty}^{\infty} x[i]h[n-i], \quad n \in \mathbb{Z}. \tag{2}$$

Speed-up computing with convolution theorem [2/]

Discrete convolution, finite support

Consider x, h defined on the finite set:

$$x[n],\ 0 \leq n \leq M-1,\ \ len(x) = M, \tag{3}$$

$$h[n],\ 0 \leq n \leq N-1,\ \ len(h) = N. \tag{4}$$

Then

$$(x * h)[n] = \sum_{i=0}^{M-1} x[i]h[n-i] = \sum_{i=0}^{N-1} h[i]x[n-i], \quad 0 \leq n < M+N-1. \tag{5}$$

Running product of two vectors

Consider denoting:

- x - a (longer) numeric vector of length M, for which we want to compute running window average with window length m,
- y - a (shorter) numeric vector of length m, $m < M$.

Then

$$(x * y)[n] = \sum_{i=0}^{m-1} y[i]x[n-i] = \sum_{i=0}^{m-1} y[i]x'[i] \tag{7}$$

is a product of vector y and vector x' which is a (reverse of) part of x starting from x's index $i = (n - m + 1)$ to $i = n$.

Computing whole convolution function $(x * y)[n]$ gives values of product of subsequent windows of x and vector y.

Convolution theorem (where the speed-up comes from)

The convolution theorem states that, under suitable conditions, the Fourier transform of a convolution of two functions f,g is the pointwise product of their Fourier transforms:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \tag{8}$$

where:

- $\mathcal{F}\{f * g\}$, $\mathcal{F}\{f\}$ and $\mathcal{F}\{g\}$ – Fourier transform operators for $f * g$, f and g, respectively,
- $\mathcal{F}\{f\}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx$ – Fourier transform of a function f.

By applying the inverse Fourier transform, we get

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}. \tag{9}$$

Speed-up computing with convolution theorem [5/]

The convolution representation given by RHS of

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}.$$

can be used for fast implementation of convolution:

- The standard convolution algorithm has quadratic computational complexity, $O\left(n^2\right)$.
- Using above result, and using a fast Fourier transform (FFT) algorithm that computes the discrete Fourier transform of a sequence, the complexity of the convolution can be reduced to $O(n \log n)$.

Convolution used in `runstats` R package

Goal: compute rolling covariance between (longer) `x` and (shorter) `y`

```
RunningCov(x, y){

    # (...)

    covxy <- (conv(x, y) - W * meanx * meany)/(W - 1)

}
```

Convolution used in `runstats` R package

Goal: compute rolling covariance between (longer) `x` and (shorter) `y`

```
RunningCov(x, y){

    # (...)

    covxy <- (conv(x, y) - W * meanx * meany)/(W - 1)

}
```

Equivalent formulas for unbiased sample covariance estimator

$$\boxed{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i y_i - n\overline{xy}\right)} = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)\left(y_i - \bar{y}\right)$$

Convolution used in `runstats` R package

Goal: compute rolling covariance between (longer) `x` and (shorter) `y`

```
RunningCov(x, y){

    # (...)

    covxy <- ( conv(x, y) - W * meanx * meany)/(W - 1)

}
```

convolution of (longer) x and (shorter) y
:= "rolling product" of x and y

(precomputed) rolling mean of x

Convolution used in `runstats` R package

Goal: compute rolling covariance between (longer) `x` and (shorter) `y`

```
RunningCov(x, y){

    # (...)

    covxy <- ( conv(x, y) - W * meanx * meany)/(W - 1)

}
```

convolution of (longer) x and (shorter) y
:= "rolling product" of x and y

(precomputed) rolling mean of x

Computing whole convolution function $(x * y)[n]$ gives values of product of subsequent windows of x and vector y.

`runstats` R package: running correlation

```
## Running covariance of long time-series x and short(er) y
RunningCor.sapply <- function(x, y){
  l_x <- length(x)
  l_y <- length(y)
  sapply(1:(l_x - l_y + 1), function(i){
    cor(x[i:(i+l_y-1)], y)
  })
}

N <- 10000000  # 10,000,000
n <- 100
x <- runif(N)
y <- runif(n)

system.time({
  RunningCor.sapply(x, y)
})
#    user  system elapsed
# 516.994   2.554 519.946

system.time({
  runstats::RunningCor(x, y)
})
#  user  system elapsed
# 5.922   0.452   6.383
```

~18h of fs=100Hz 1-dimensional time-series

~ 8.5 minutes of execution

~ 6 seconds of execution (~87x faster)

`runstats` R package

Provides methods for fast computation of running sample statistics for a time-series.

Implemented running sample statistics:

- **mean**, **standard deviation**, and **variance** over a fixed-length window of time-series,
- **correlation**, **covariance**, and **Euclidean distance** (L2 norm) between short-time pattern and time-series.

CRAN index: https://cran.r-project.org/web/packages/runstats/index.html

`runstats` R package - a comparator example

Dane Van Domelen (personal website)

- Former post doc in JHU Biostat
- Biostatistician at Karyopharm Therapeutics Inc
- Authored a bunch of interesting R packages
- R package dvmisc: Convenience Functions, Moving Window Statistics, and Graphics
  - includes `sliding_cor`, `sliding_cov` functions implemented in rcpp; very fast!

- accelerometry
- crowdopt
- dvmisc
- nhanesaccel
- nhanesdata
- pooling
- tab
- stocks

Note:

- Implementation of convolution via convolution theorem + FFT is a general way that can be used to speed-up convolution in mostly any language (i.e. Python)
- **Nearest future plans** for `runstats` update: search for fastest FFT implementation I can plug to use in R (perhaps rcpp?)