Università della Svizzera italiana
Faculty of Informatics
Institute of Computational Science ICS

A2: K-means

looking for the best implementation

CECAM workshop, Mainz, 2019

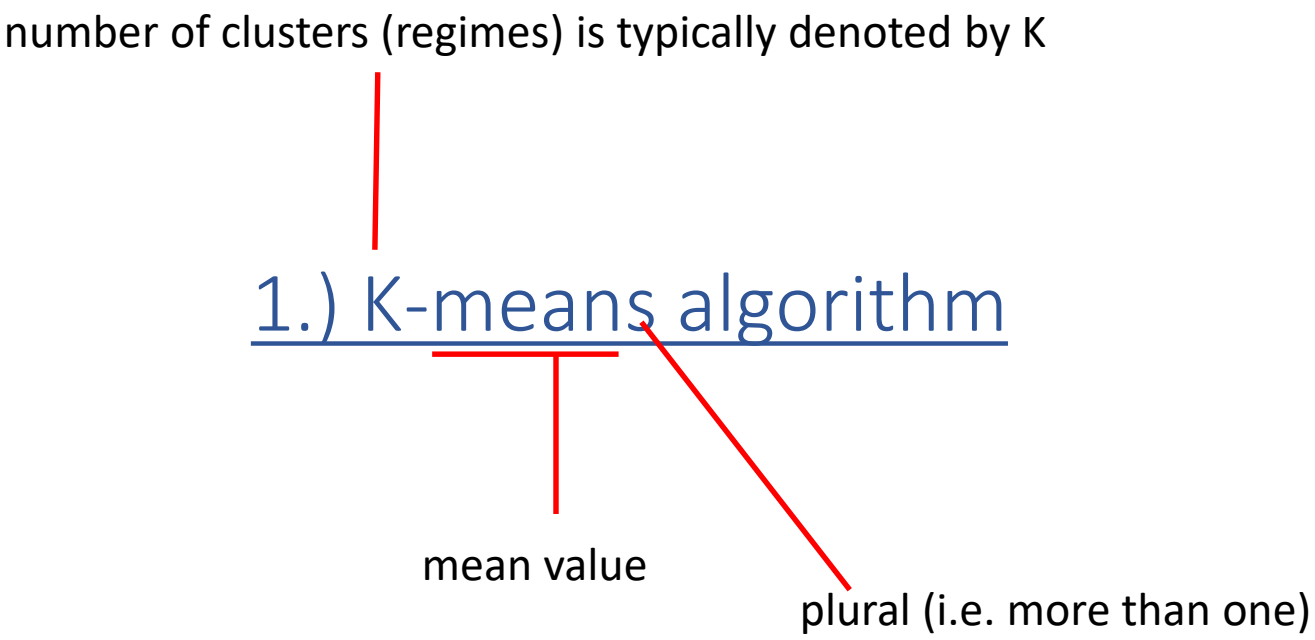
Illia Horenko, Lukas Pospisil

Outline

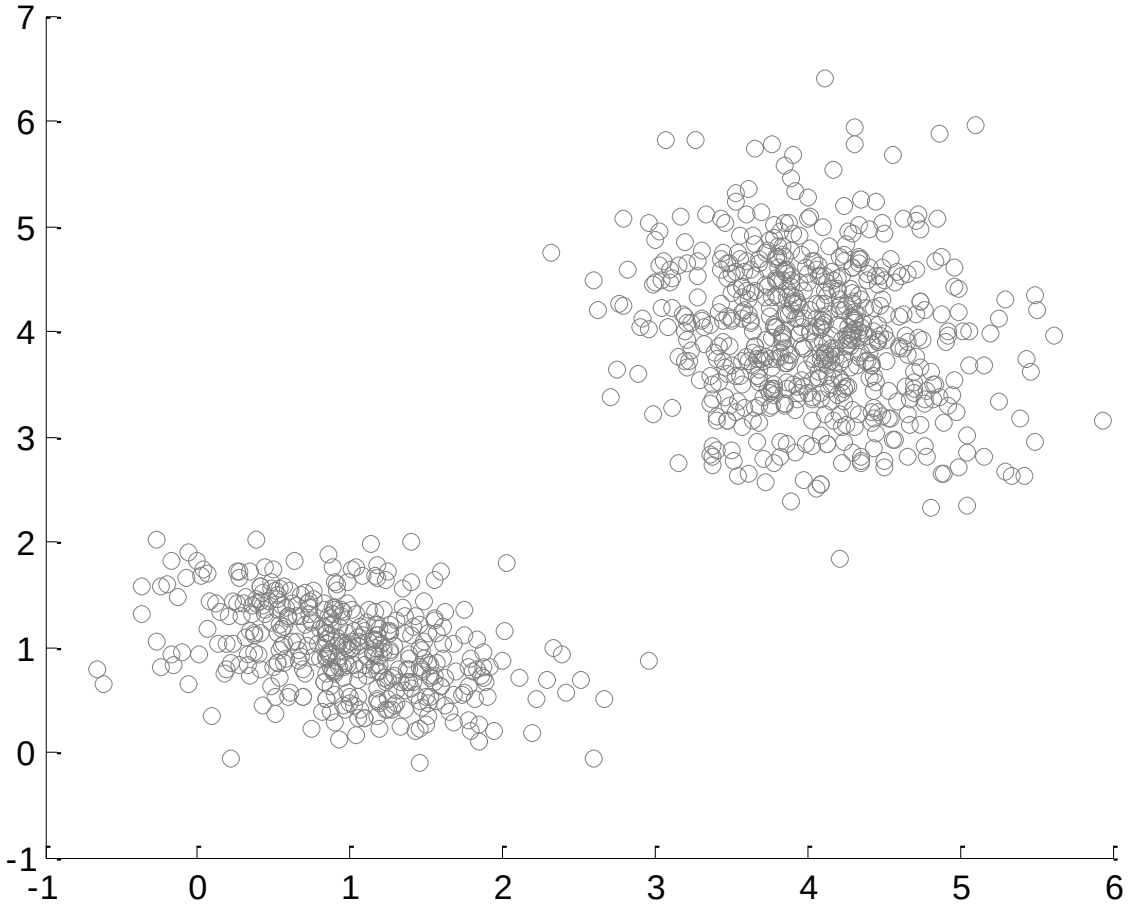
1.) K-means algorithm
(short reminder)

2.) Asymptotic time complexity in practice
(using the best implementation of linear algebra routines)

3.) implementation of K-means algorithm
(looking for fast clustering)

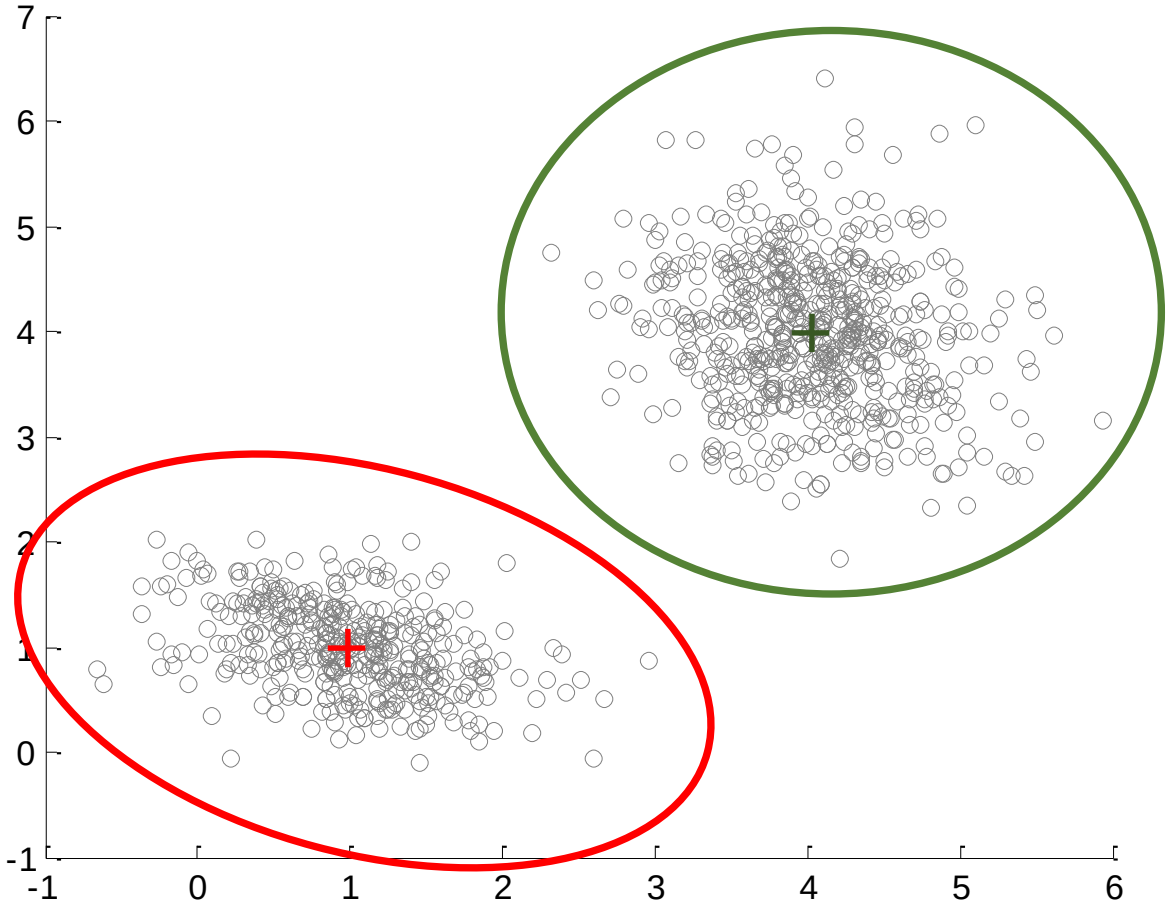

number of clusters (regimes) is typically denoted by K
1.) K-means algorithm
mean value
plural (i.e. more than one)

Reminder: Clustering



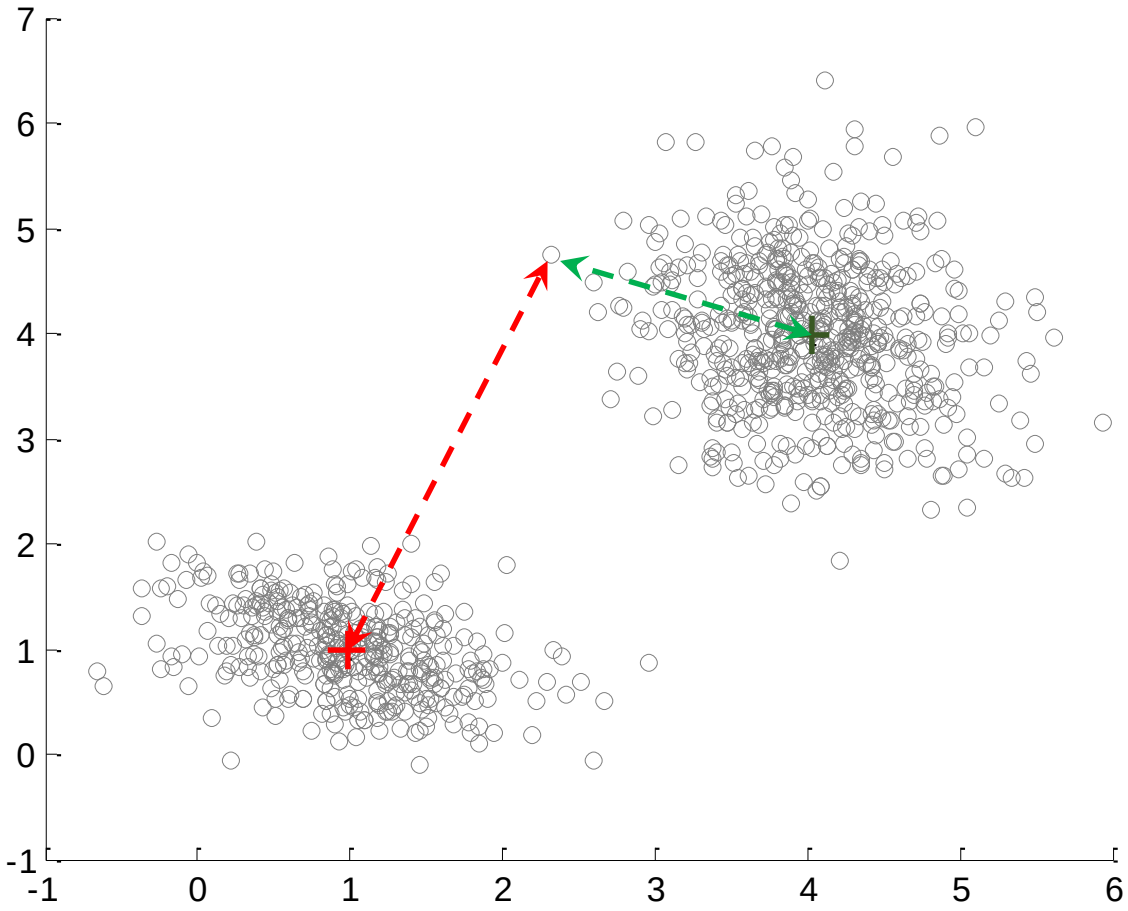
- we don't have any apriori classification for supervised learning (e.g., SVM)
- each cluster consists of *similar* points (points in cluster are close to each other)
- each cluster can be characterised by *mean value* of points inside it

Reminder: Clustering



- we don't have any apriori classification for supervised learning (e.g., SVM)
- each cluster consists of *similar* points (points in cluster are close to each other)
- each cluster can be characterised by *mean value* of points inside it

Reminder: Clustering



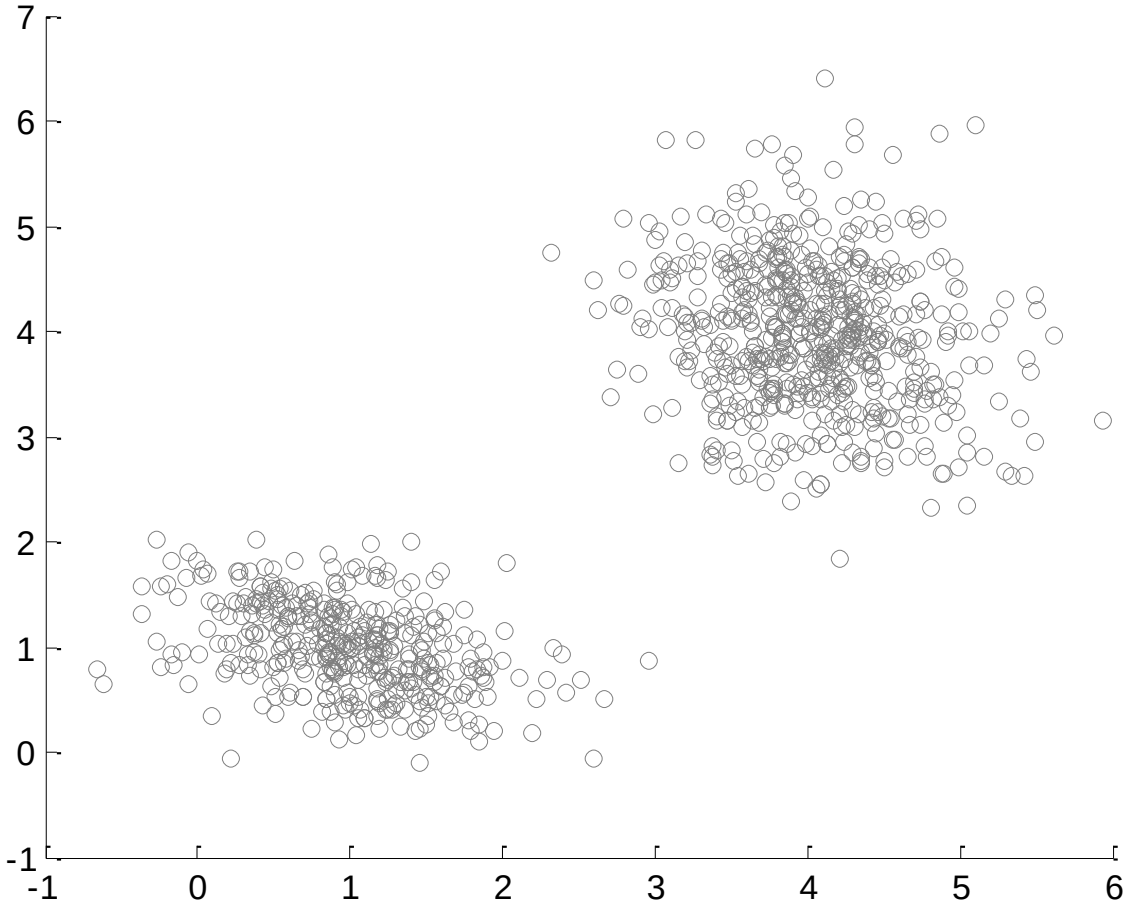
- if we have a *right mean values* of clusters, then clustering problem is solved, because ...
- the point belongs to the *closer* cluster (with closer mean value)

Reminder: Clustering

- on the other hand - if we have *right point clustering*, then computation of mean value is trivial:

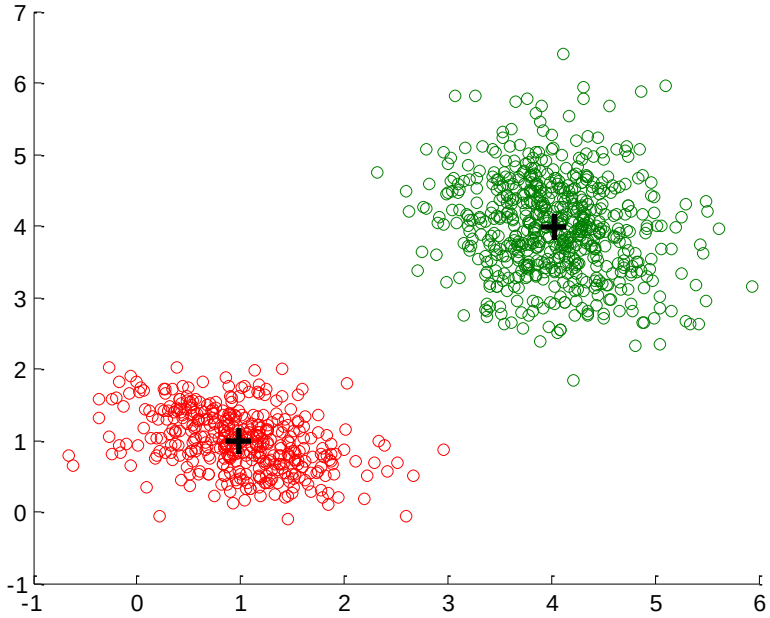
$$\mu = \frac{1}{N}\sum_t x_t \qquad \mu = \frac{1}{N}\sum_t x_t$$

Reminder: Clustering



- but we don't know anything
- both of *mean values of clusters* and *affiliation to clusters* is unknown

Reminder: K-means



set feasible initial approximation $\Gamma^0 \in \Omega_\Gamma$

while $\|L(\Gamma^{it}, \Theta^{it}) - L(\Gamma^{it-1}, \Theta^{it-1})\| > \varepsilon$

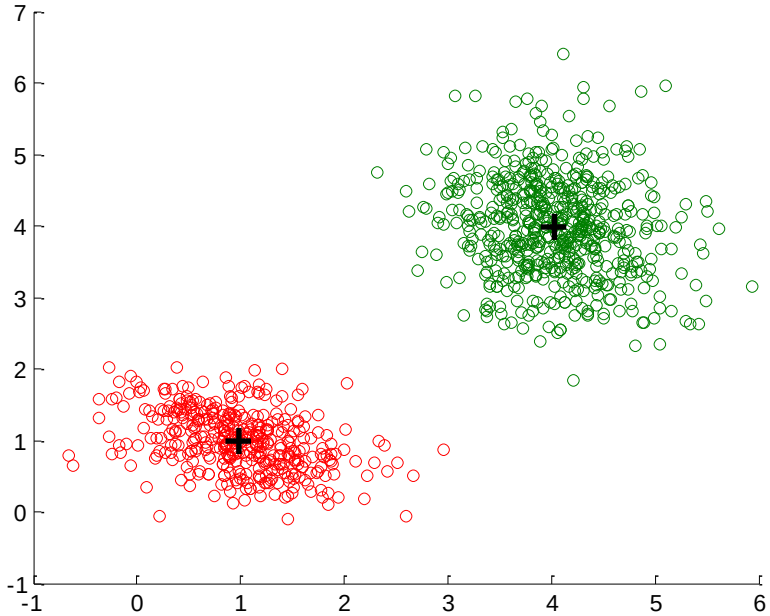
$\quad$ *solve* $\Theta^{it} := \arg\min_{\Theta} L(\theta, \Gamma^{it})$ *(with fixed* Γ^{it}*)*

$\quad$ *solve* $\Gamma^{it} := \arg\min_{\Gamma \in \Omega_\Gamma} L(\Theta^{it}, \Gamma)$ *(with fixed* Θ^{it}*)*

$\quad$ $it := it + 1$

endwhile

Reminder: K-means



set feasible initial approximation $\Gamma^0 \in \Omega_\Gamma$

while $\|L(\Gamma^{it}, \Theta^{it}) - L(\Gamma^{it-1}, \Theta^{it-1})\| > \varepsilon$

$\quad$ *solve* $\Theta^{it} := \arg\min_{\Theta} L(\theta, \Gamma^{it})$ *(with fixed* Γ^{it}*)*

$\quad$ *solve* $\Gamma^{it} := \arg\min_{\Gamma \in \Omega_\Gamma} L(\Theta^{it}, \Gamma)$ *(with fixed* Θ^{it}*)*

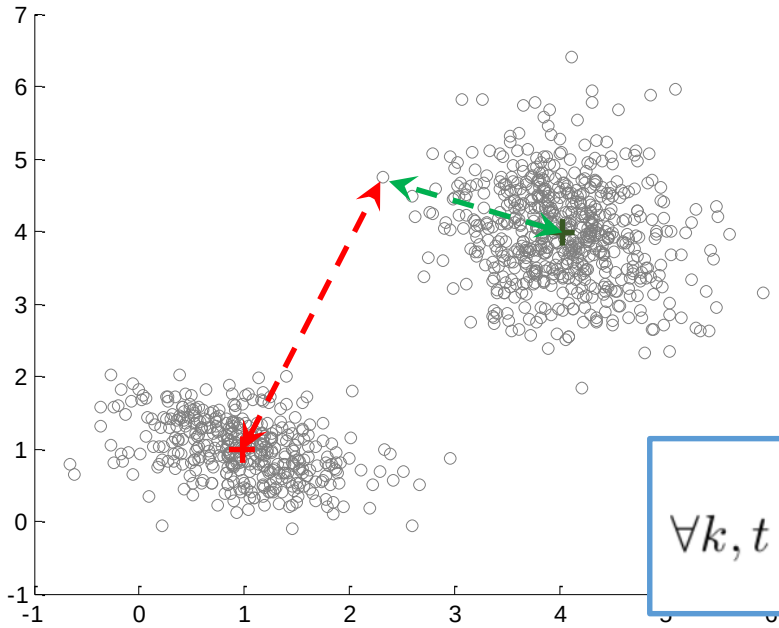
$\quad$ $it := it + 1$

endwhile

$$\forall k : \theta_k^* = \frac{\sum_{t=1}^{T} \gamma_{k,t} x_t}{\sum_{t=1}^{T} \gamma_{k,t}}$$

(separable in k)

Reminder: K-means



$$\forall k,t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \|x_t - \theta_{\hat{k}}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

(separable in t)

set feasible initial approximation $\Gamma^0 \in \Omega_\Gamma$

while $\|L(\Gamma^{it}, \Theta^{it}) - L(\Gamma^{it-1}, \Theta^{it-1})\| > \varepsilon$

$\quad$ *solve* $\Theta^{it} := \arg\min_{\Theta} L(\theta, \Gamma^{it})$ *(with fixed* Γ^{it}*)*

$\quad$ *solve* $\Gamma^{it} := \arg\min_{\Gamma \in \Omega_\Gamma} L(\Theta^{it}, \Gamma)$ *(with fixed* Θ^{it}*)*

$\quad$ $it := it + 1$

endwhile

K-means algorithm – some remarks

- solution depends on initial approximation (e.g., permutation of clusters)
- number of clusters K is known apriori? (AIC should give answer)
- can be very easily applied to any data dimension

- the norm (for measuring similarity between cluster points) can be changed
- additional regularization can be added (...) to implement apriori information

2.) Asymptotic time complexity in practice

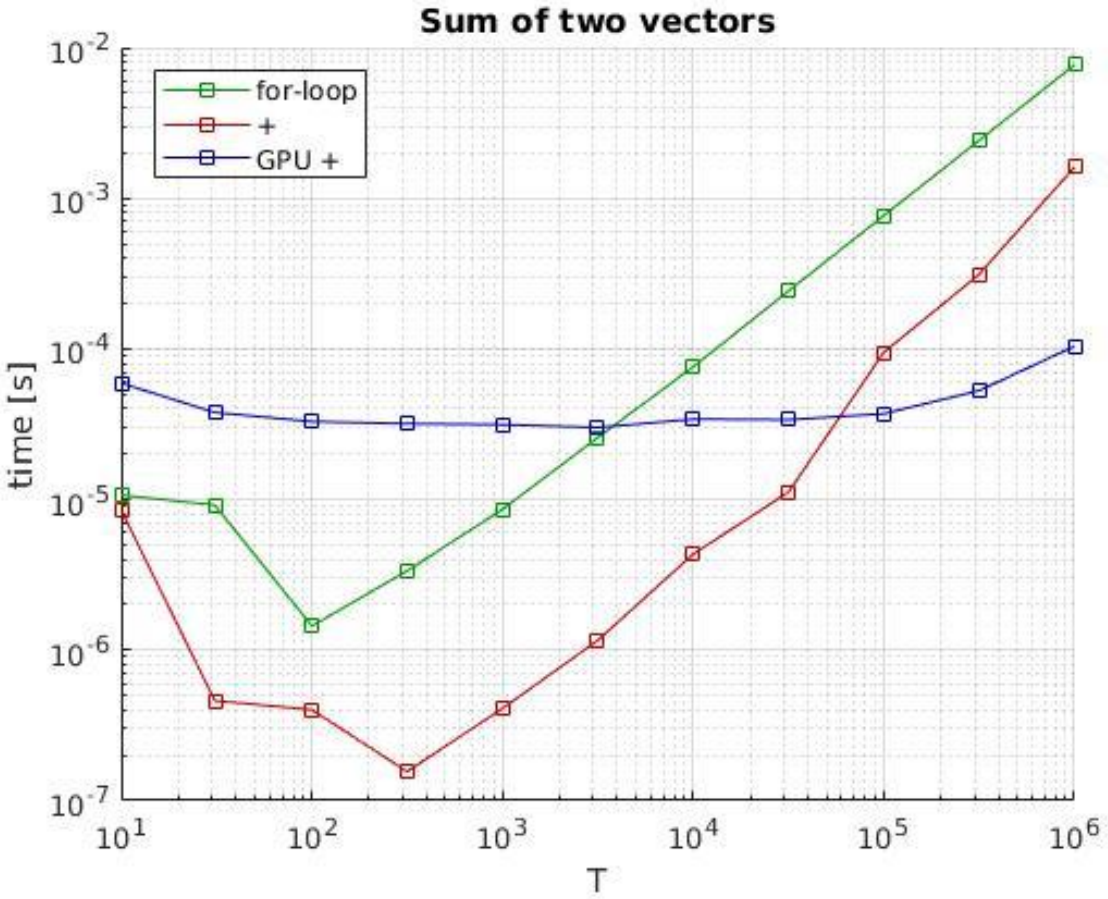
Time complexity of BLAS operations

$x, y \in \mathbb{R}^T$ given
compute $z := x + y$

Time complexity of BLAS operations

$x, y \in \mathbb{R}^T$ given
compute $z := x + y$

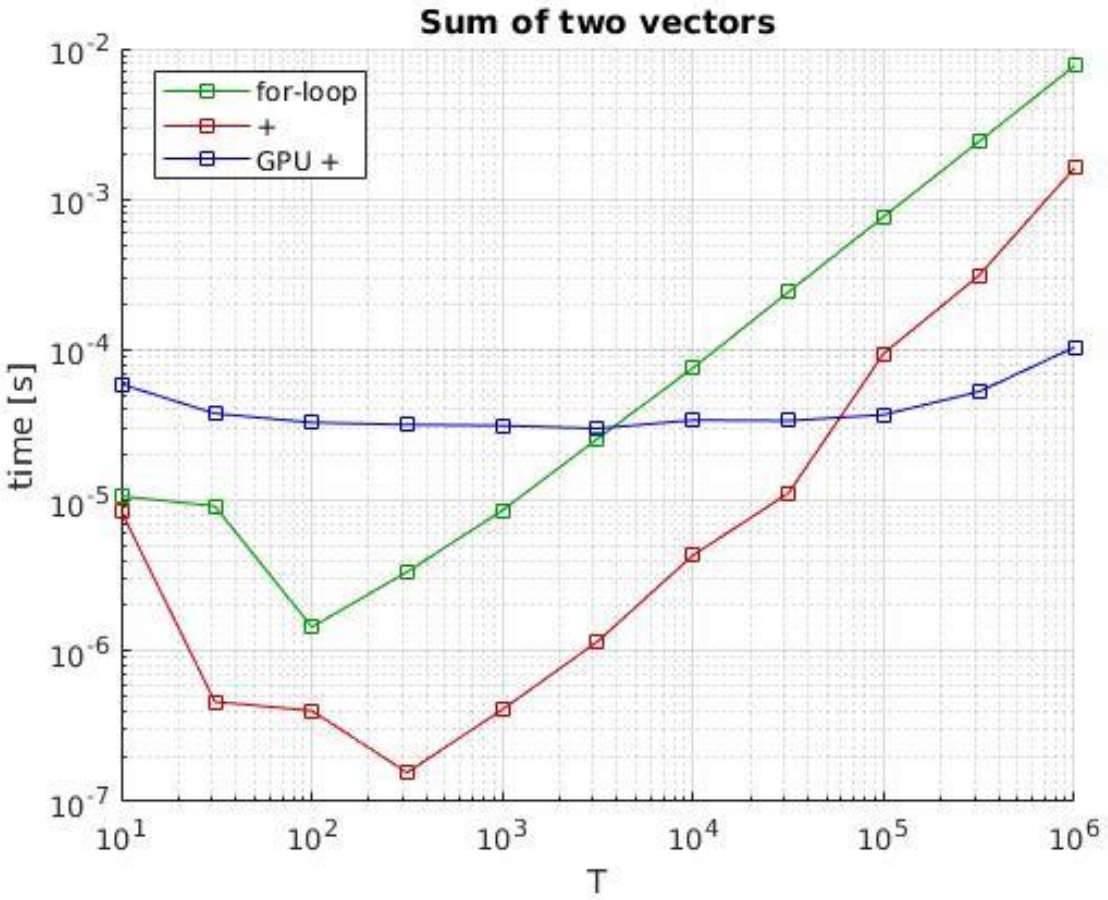
```
% some really boring data
x = ones(T,1);
y = ones(T,1);
z = zeros(T,1);
Ntests = 1e3;
```



Time complexity of BLAS operations

$x, y \in \mathbb{R}^T$ given
compute $z := x + y$

```
% some really boring data
x = ones(T,1);
y = ones(T,1);
z = zeros(T,1);
Ntests = 1e3;
```




```
% for-loop
timer1 = tic;
for n=1:Ntests
    for j=1:T
        z(j) = x(j) + y(j);
    end
end
times1(i) = toc(timer1)/Ntests;
```

```
% matlab +
timer2 = tic;
for n=1:Ntests
    z = x + y;
end
times2(i) = toc(timer2)/Ntests;
```

```
% GPU matlab +
x_gpu = gpuArray(x);
y_gpu = gpuArray(y);
z_gpu = gpuArray(z);
timer3 = tic;
for n=1:Ntests
    z_gpu = x_gpu + y_gpu;
    wait(gpudev);
end
times3(i) = toc(timer3)/Ntests;
```

Vectorization - software support

MATLAB is an interactive software programming environment for numerical computations and visualization. Internally MATLAB uses Intel MKL **Basic Linear Algebra Subroutines (BLAS)** and **Linear Algebra package (LAPACK)** routines to perform the underlying computations when running on Intel processors.

[https://software.intel.com/en-us/articles/using-intel-math-kernel-library-with-mathworks-matlab-on-intel-xeon-phi-coprocessor-system]

Basic Linear Algebra Subroutines

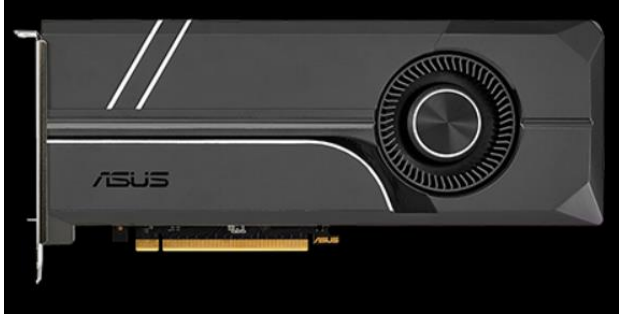
- BLAS is a specification that prescribes a set of low-level routines for performing common linear algebra operations
- optimized for speed on a particular machine, so using them can bring substantial performance benefits
- take advantage of special floating point hardware such as vector registers or SIMD instructions
- Fortran library (with interfaces/wrappers to other languages), 1979
- AMD Core Math Library (ACML), ATLAS, Intel Math Kernel Library (MKL), OpenBLAS

Level 1: Vector-vector operations. *O(n)* data and *O(n)* work.
Level 2: Matrix-vector operations. *O(n^2)* data and *O(n^2)* work.
Level 3: Matrix-matrix operations. *O(n^2)* data and *O(n^3)* work.

[http://www.netlib.org/blas/]

Conclusion: In Matlab, vector and matrix operations are not performed by Matlab code, but using external (and optimized) libraries.
Do not implement these operations by yourself (especially in Matlab :).

CUDA in Matlab



Compute **U**nified **D**evice **A**rchitecture

MATLAB supports CUDA kernel development by providing a language and development environment for prototyping algorithms and incrementally developing and testing CUDA kernels.

[https://developer.nvidia.com/matlab-cuda]

The following functions and their symbol operators are enhanced to accept gpuArray input arguments so that they execute on the GPU:

abs	compan	flip	isnan	pcg	spdiags
acos	complex	fliplr	isnumeric	perms	sph2cart
acosd	cond	flipud	isreal	permute	sprand
acosh	conj	floor	isrow	pinv	sprandn
acot	conv	fprintf	issorted	planerot	sprandsym
acotd	conv2	full	issparse	plot (and related)	spconvert
acoth	convn	gamma	issymmetric	plus	sph2cart
acsc	corrcoef	gammainc	istril	pol2cart	sprand
acscd	cos	gammaincinv	istriu	poly	sprandn
acsch	cosd	gammaln	isvector	polyarea	sprandsym
accumarray	cosh	gather	kron	polyder	sprintf
all	cot	ge	ldivide	polyfit	sqrt
and	cotd				

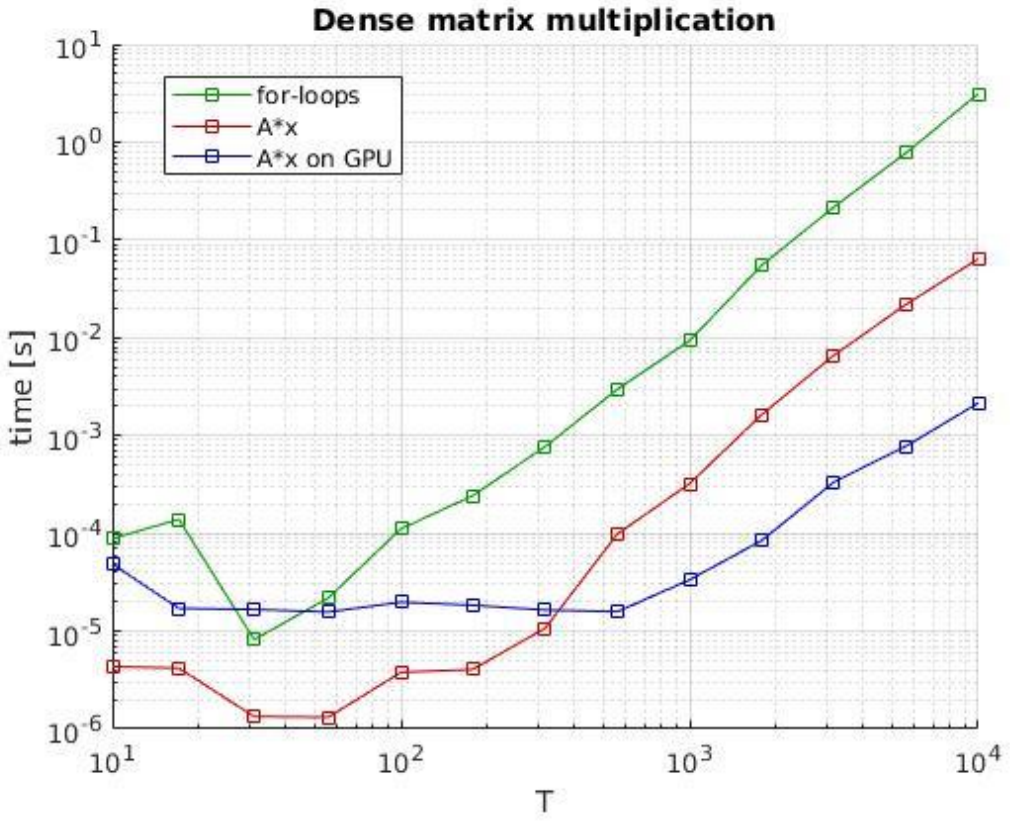
[https://ch.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html]

<u>What we will need:</u>

- hardware (GeForce GTX 1080Ti Turbo - 11GB GDDR5X, 3584 CUDA cores)
- sofware
    - NVidia CUDA toolkit (free, includes nvcc, cublas, cusparse...)
    - Matlab (as latest version as possible)

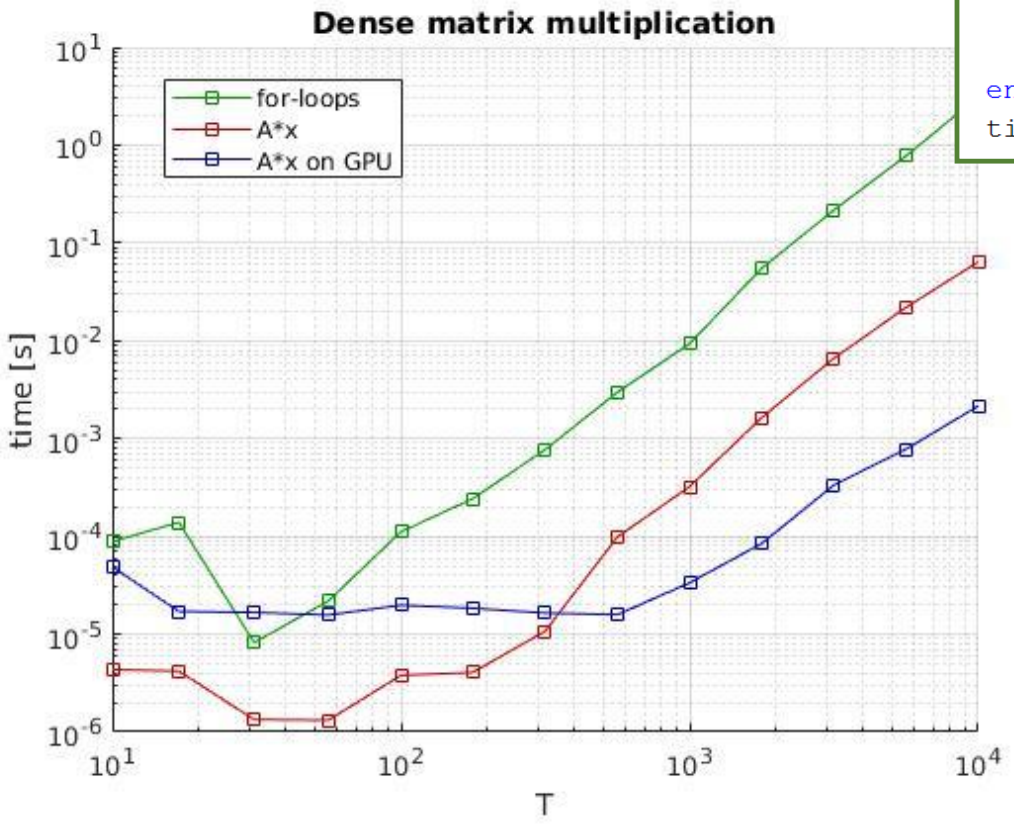
Time complexity of BLAS operations

$x \in \mathbb{R}^T, A \in \mathbb{R}^{T,T}$ given
compute $y := Ax$



Time complexity of BLAS operations

$x \in \mathbb{R}^T, A \in \mathbb{R}^{T,T}$ given
compute $y := Ax$



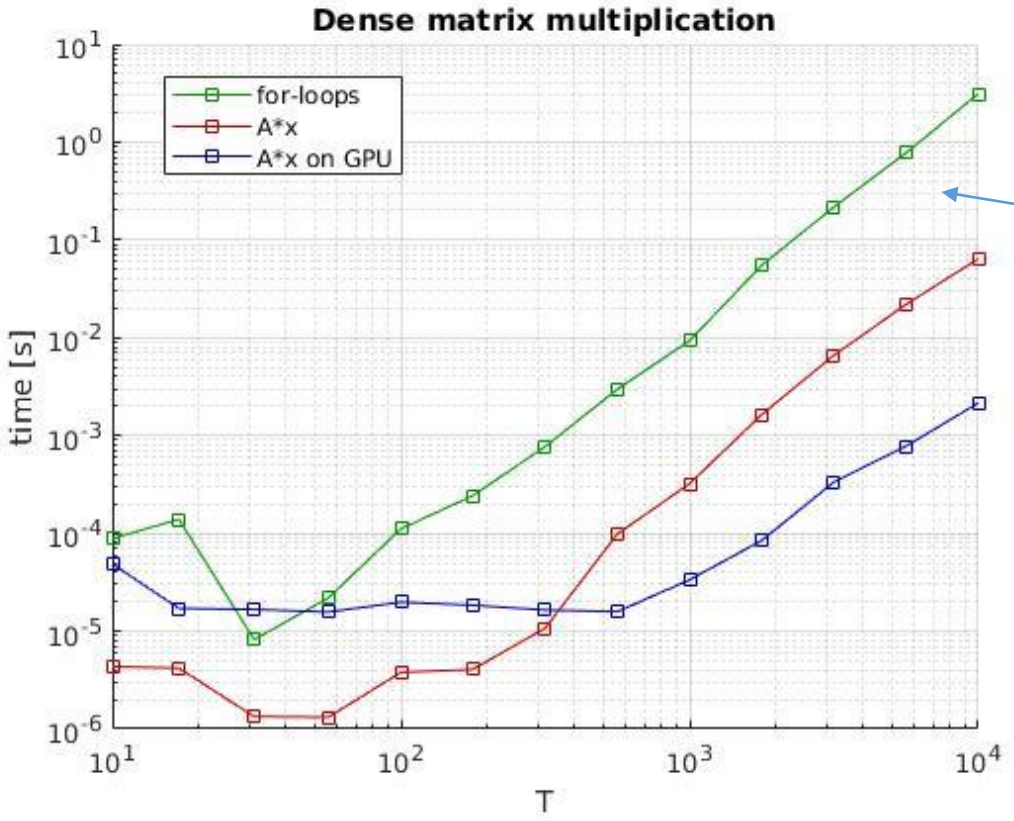

```
timer1 = tic;
for n=1:Ntests
    for row=1:T
        y(row) = 0;
        for col=1:T
            y(row) = y(row) + A(row,col)*x(col);
        end
    end
end
times1(i) = toc(timer1)/Ntests;
```

```
timer2 = tic;
for n=1:Ntests
    y = A*x;
end
times2(i) = toc(timer2)/Ntests;
```

```
x_gpu = gpuArray(x);
A_gpu = gpuArray(A);
y_gpu = gpuArray(y);
timer3 = tic;
for n=1:Ntests
    y_gpu = A_gpu*x_gpu;
    wait(gpudev);
end
times3(i) = toc(timer3)/Ntests;
```

Time complexity of BLAS operations

$x \in \mathbb{R}^T, A \in \mathbb{R}^{T,T}$ given
compute $y := Ax$



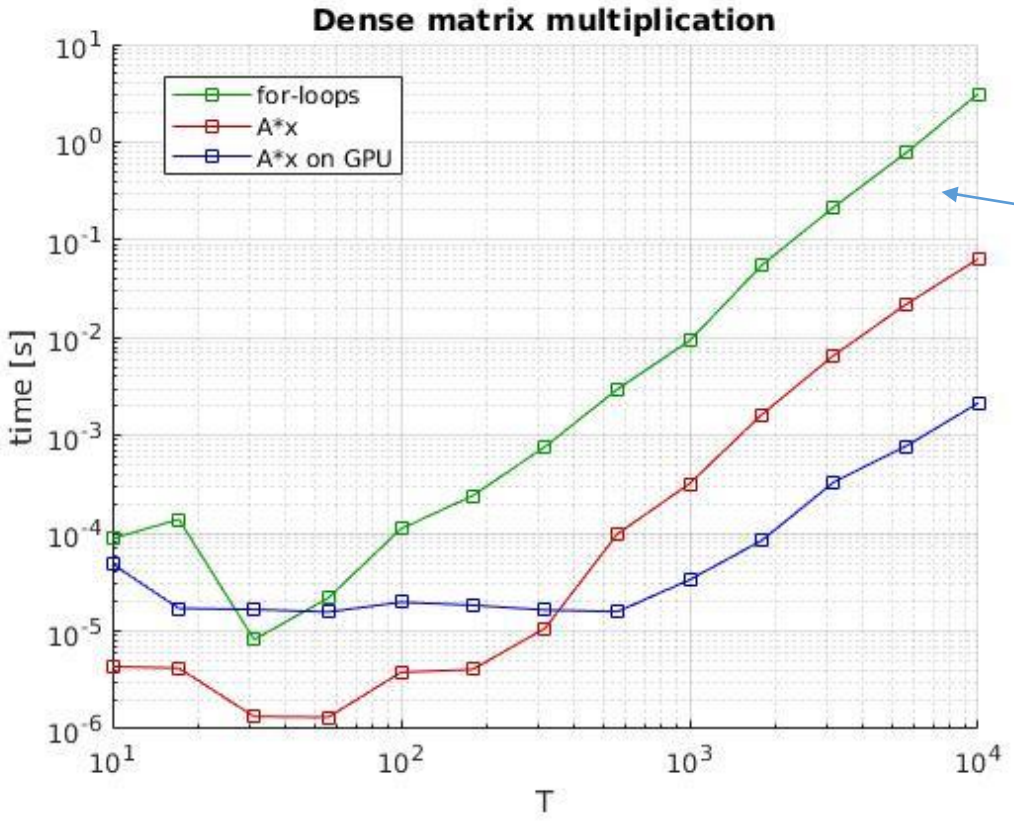
Is it linear !?

Time complexity of BLAS operations

$x \in \mathbb{R}^T, A \in \mathbb{R}^{T,T}$ given
compute $y := Ax$

!!! quadratic function is linear in log-log scale

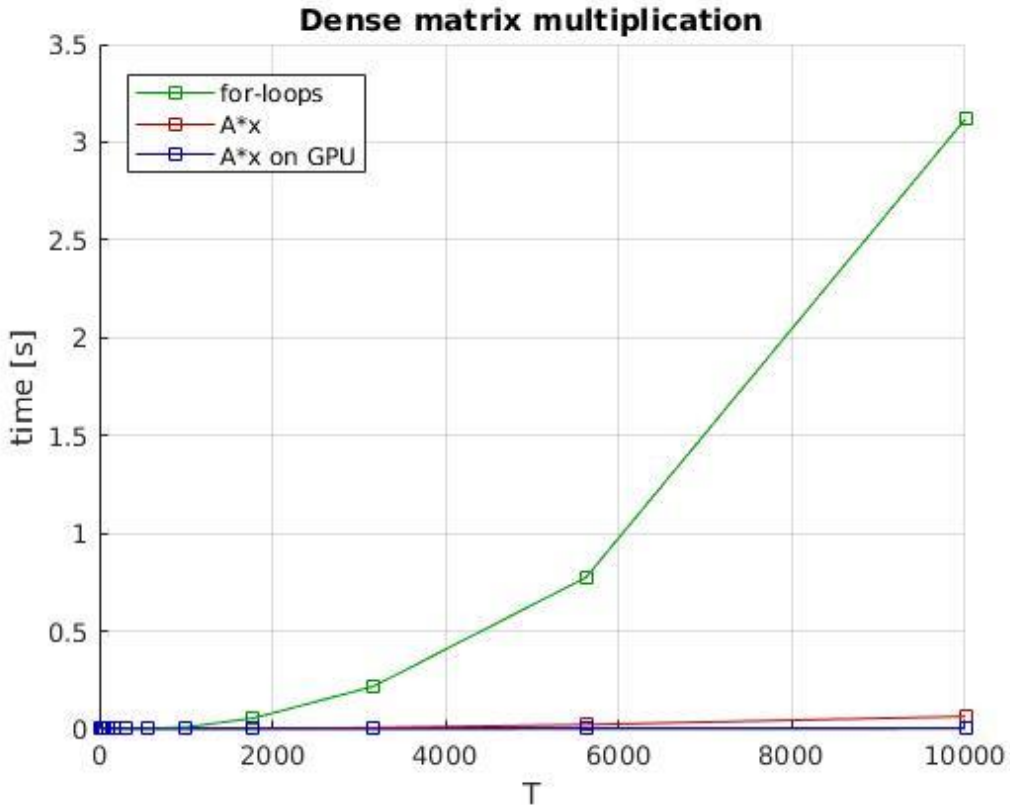
$$
\begin{aligned}
y(x) &= x^2 \\
\log y(x) &= \log x^2 \\
\log y(x) &= 2 \log x \\
y_{\log}(x) &= 2x_{\log}
\end{aligned}
$$



Is it linear !?

Time complexity of BLAS operations

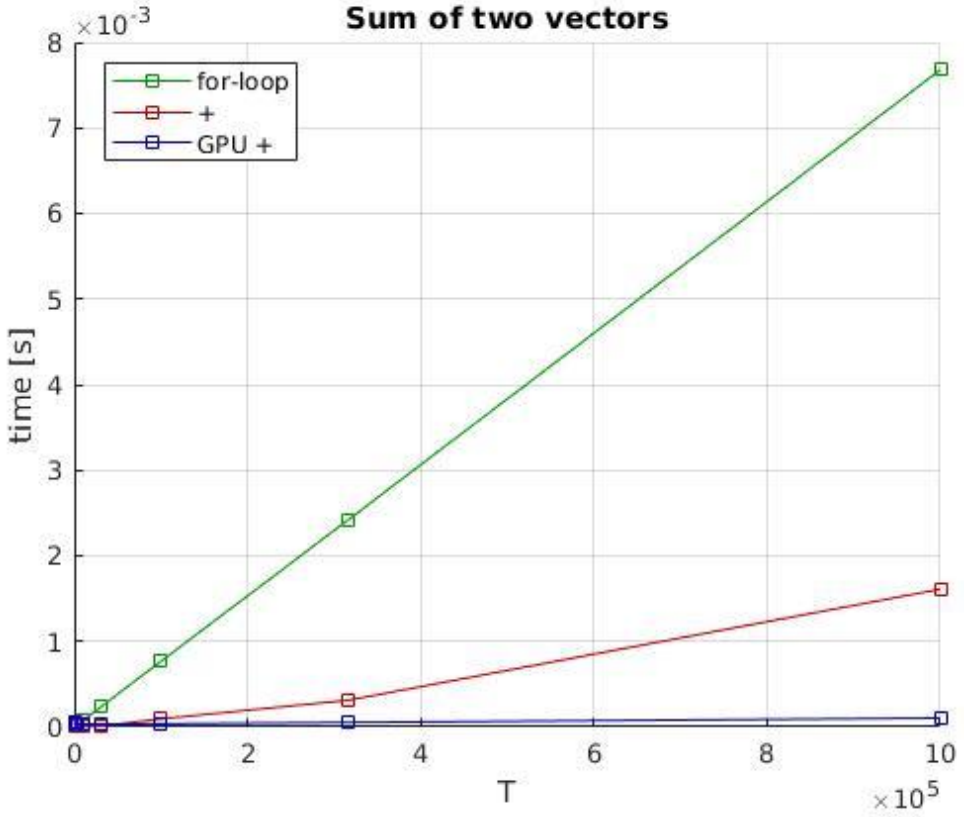
$x \in \mathbb{R}^T, A \in \mathbb{R}^{T,T}$ given
compute $y := Ax$



Time complexity of BLAS operations

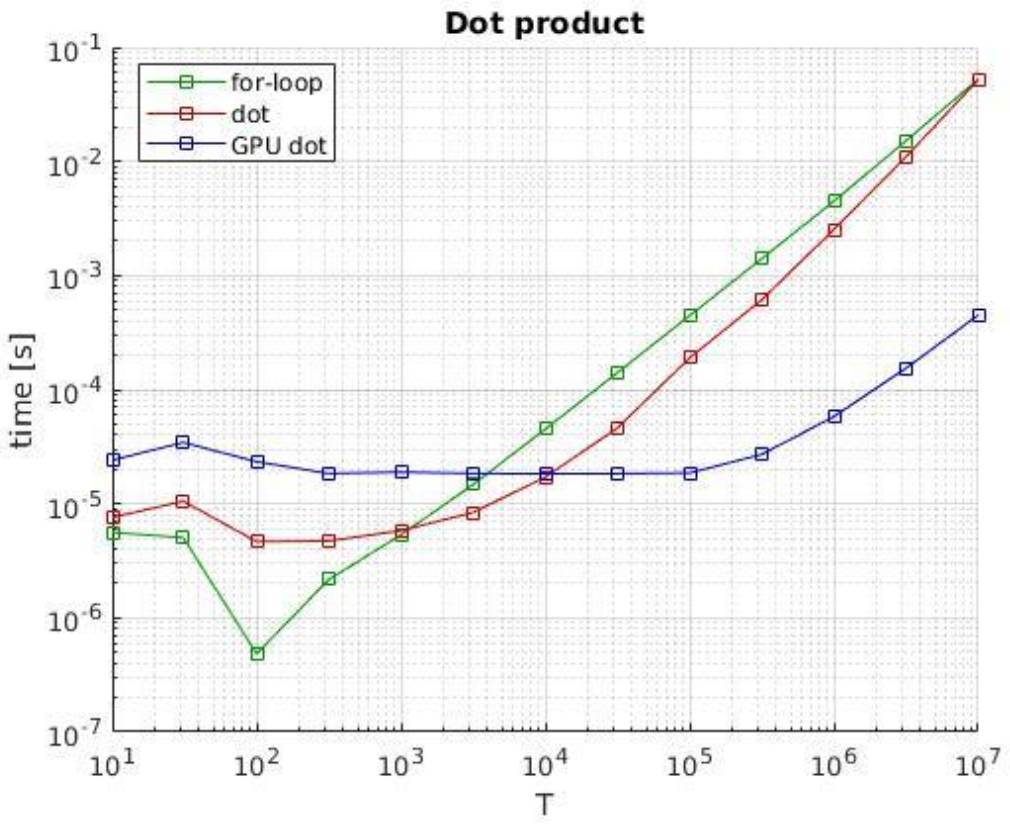
$x, y \in \mathbb{R}^T$ given
compute $z := x + y$

(back to sum - just to be sure)



Time complexity of BLAS operations

$x, y \in \mathbb{R}^T$ given
compute $\alpha := \langle x, y \rangle$



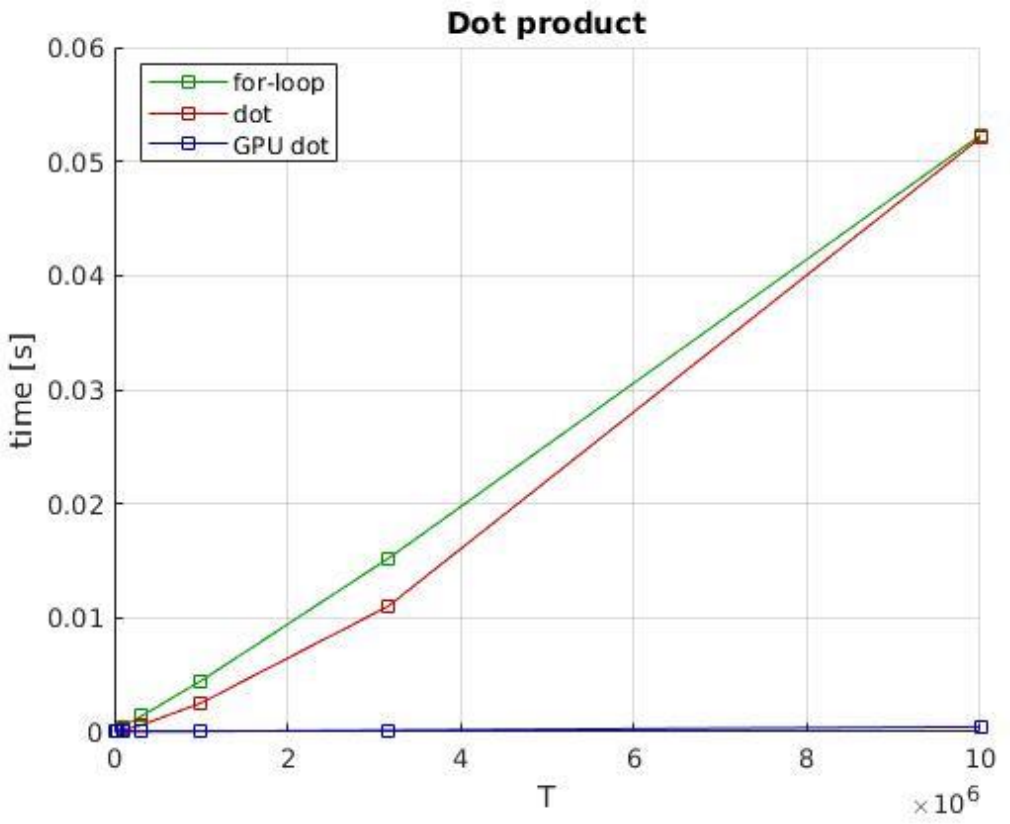

```
% for-loop
timer1 = tic;
for n=1:Ntests
    alpha1 = 0;
    for j=1:T
        alpha1 = alpha1 + x(j)*y(j);
    end
end
times1(i) = toc(timer1)/Ntests;
```

```
% matlab dot
timer2 = tic;
for n=1:Ntests
    alpha2 = dot(x,y);
end
times2(i) = toc(timer2)/Ntests;
```

```
% GPU matlab dot
x_gpu = gpuArray(x);
y_gpu = gpuArray(y);
timer3 = tic;
for n=1:Ntests
    alpha3 = dot(x_gpu,y_gpu);
end
wait(gpudev)
times3(i) = toc(timer3)/Ntests;
```

Time complexity of BLAS operations

$x, y \in \mathbb{R}^T$ given
compute $\alpha := \langle x, y \rangle$



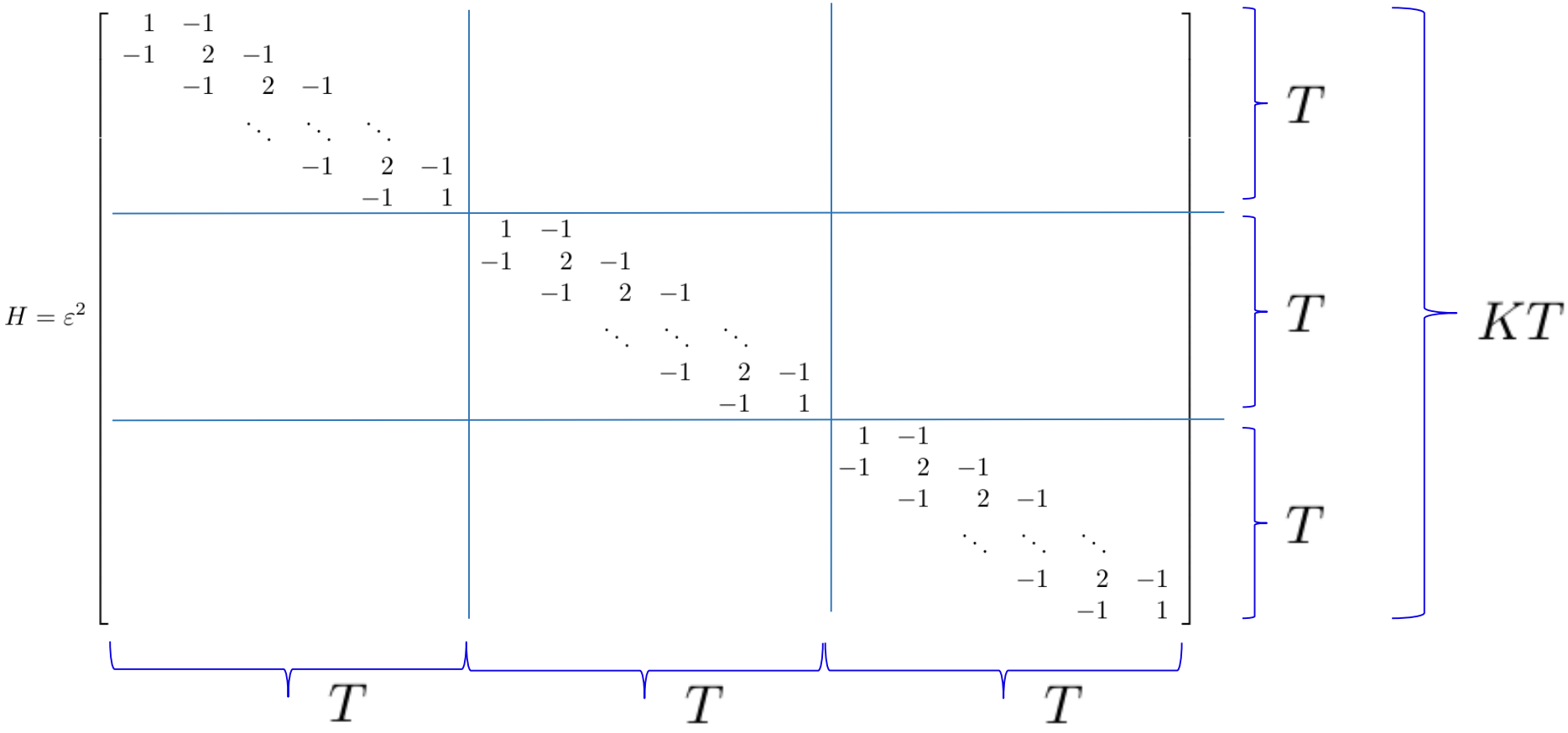

```
% for-loop
timer1 = tic;
for n=1:Ntests
    alpha1 = 0;
    for j=1:T
        alpha1 = alpha1 + x(j)*y(j);
    end
end
times1(i) = toc(timer1)/Ntests;
```

```
% matlab dot
timer2 = tic;
for n=1:Ntests
    alpha2 = dot(x,y);
end
times2(i) = toc(timer2)/Ntests;
```

```
% GPU matlab dot
x_gpu = gpuArray(x);
y_gpu = gpuArray(y);
timer3 = tic;
for n=1:Ntests
    alpha3 = dot(x_gpu,y_gpu);
end
wait(gpudev)
times3(i) = toc(timer3)/Ntests;
```

Time complexity of BLAS operations

Example - matrix free multiplication:




```
ondiag = [ 1 2*ones(1,n-2) 1];
offdiag = -ones(1,n);
Hk = spdiags([offdiag' ondiag' offdiag'],-1:1,n,n);
H = kron(speye(K),Hk);
```

Time complexity of BLAS operations

Example - matrix free multiplication:

$$H = \varepsilon^2 \begin{bmatrix}
1 & -1 & & & & & & & & & & & & & & & & & & & \\
-1 & 2 & -1 & & & & & & & & & & & & & & & & & & \\
 & -1 & 2 & -1 & & & & & & & & & & & & & & & & & \\
 & & \ddots & \ddots & \ddots & & & & & & & & & & & & & & & & \\
 & & & -1 & 2 & -1 & & & & & & & & & & & & & & & \\
 & & & & -1 & 1 & & & & & & & & & & & & & & & \\
 & & & & & & 1 & -1 & & & & & & & & & & & & & \\
 & & & & & & -1 & 2 & -1 & & & & & & & & & & & & \\
 & & & & & & & -1 & 2 & -1 & & & & & & & & & & & \\
 & & & & & & & & \ddots & \ddots & \ddots & & & & & & & & & & \\
 & & & & & & & & & -1 & 2 & -1 & & & & & & & & & \\
 & & & & & & & & & & -1 & 1 & & & & & & & & & \\
 & & & & & & & & & & & & 1 & -1 & & & & & & & \\
 & & & & & & & & & & & & -1 & 2 & -1 & & & & & & \\
 & & & & & & & & & & & & & -1 & 2 & -1 & & & & & \\
 & & & & & & & & & & & & & & \ddots & \ddots & \ddots & & & & \\
 & & & & & & & & & & & & & & & -1 & 2 & -1 & & & \\
 & & & & & & & & & & & & & & & & -1 & 1 & & &
\end{bmatrix}$$

```
for k=1:K
    b2((k-1)*T+2:k*T-1) = epssqr*(2*x((k-1)*T+2:k*T-1) - x((k-1)*T+1:k*T-2) - x((k-1)*T+3:k*T)); % middle
    b2((k-1)*T+1) = epssqr*(x((k-1)*T+1) - x((k-1)*T+2)); % first
    b2(k*T) = epssqr*(x(k*T) - x(k*T-1)); % last
end
```

Time complexity of BLAS operations

Example - matrix free multiplication:

```
/* Ax = A*x + beta*Ax */
void LaplaceFreeMatrix::compute_dgemm(double *Ax, const double *x, const double beta) const {
    int k,t;

    for(int tk = 0; tk < T*K; tk++){
        int k = (int)(tk/(double)T); /* k = 0,..,K-1 */
        int t = tk-k*T; /* t = 0,...,T-1 */

        Ax[tk] = beta*Ax[tk];
        if(t==0){
            Ax[tk] += scale*(x[tk] - x[tk+1]);
        }
        if(t==T-1){
            Ax[tk] += scale*(x[tk] - x[tk-1]);
        }
        if(t>0 & t<T-1){
            Ax[tk] += scale*(2*x[tk] - x[tk-1] - x[tk+1]);
        }
    }
}
```

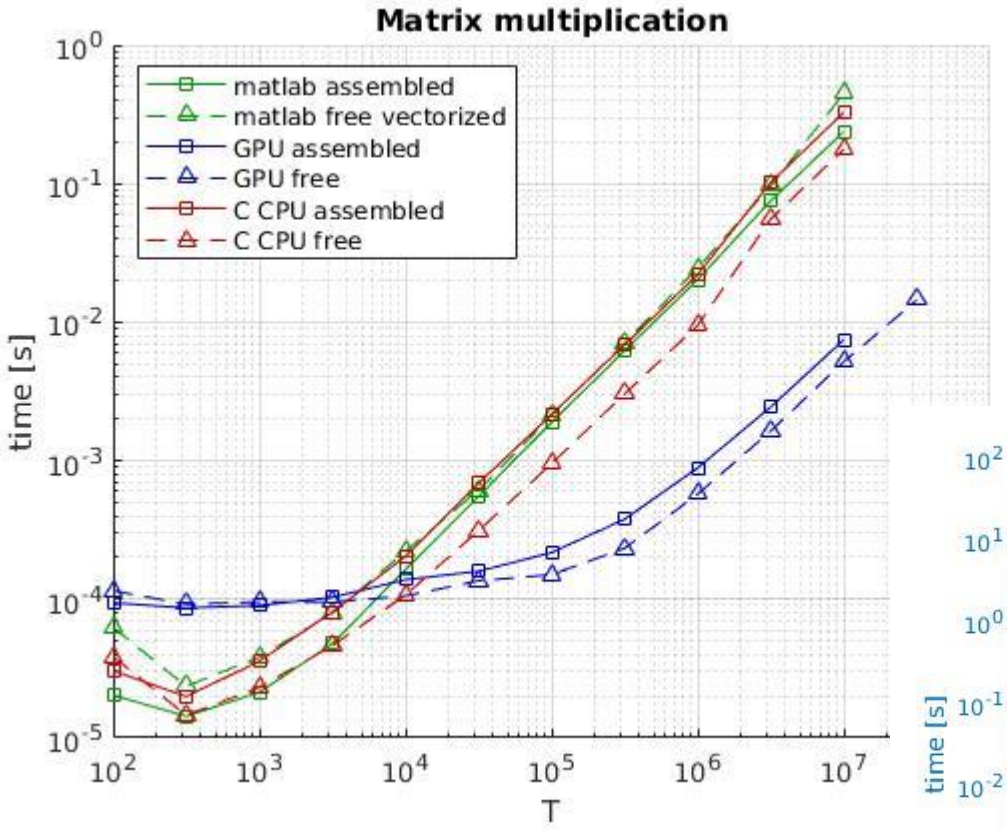
Time complexity of BLAS operations

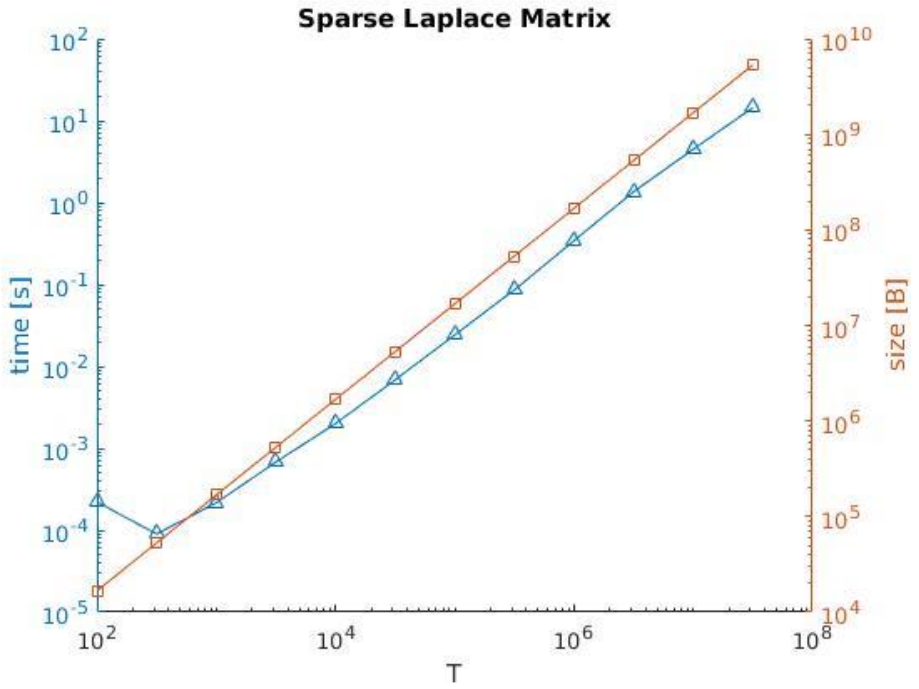
Example - matrix free multiplication:

```
__global__ void matmult_free( double *Ax,
                  const double *x,
                  const double epssqr,
                  const int K, const int T) {

    int tk = blockIdx.x*blockDim.x + threadIdx.x;

    if(tk<T*K){
        int k = (int)(tk/(double)T); /* k = 0,..,K-1 */
        int t = tk-k*T; /* t = 0,...,T-1 */

        if(t==0){
            Ax[tk] = epssqr*(x[tk] - x[tk+1]);
        }
        if(t==T-1){
            Ax[tk] = epssqr*(x[tk] - x[tk-1]);
        }
        if(t>0 & t<T-1){
            Ax[tk] = epssqr*(2*x[tk] - x[tk-1] - x[tk+1]);
        }
    }

    /* if tk >= TK then relax and do nothing */
}
```

Time complexity of BLAS operations

Example - matrix free multiplication:





Conclusion

- we have BLAS, use it!
- in scripting laguages like Matlab: eliminate for-loops, use vectorization
- consider parallelization (shared memory, distributed memory, GPU, multiGPU)
- matrix-free operations?

3.) Implementation of K-means algoritm

[a] the first code

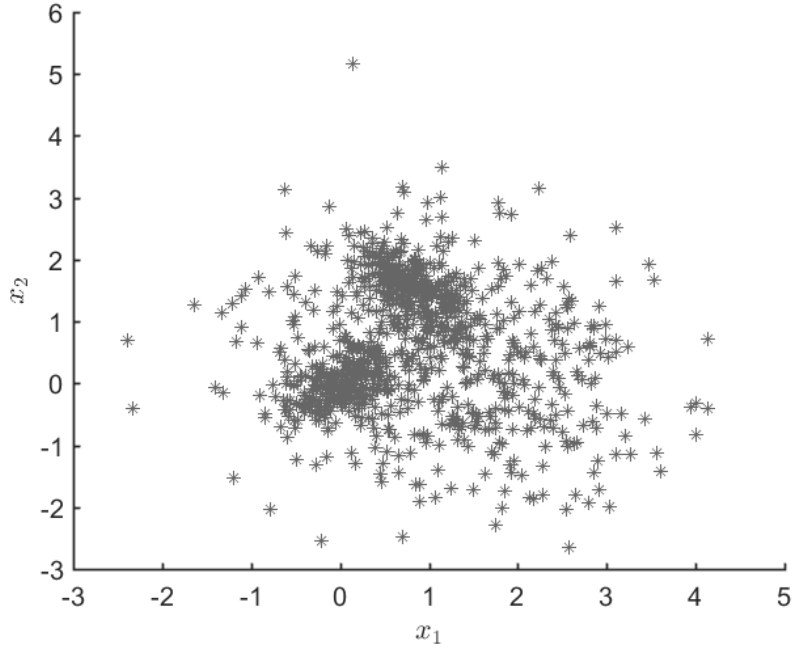
Benchmark

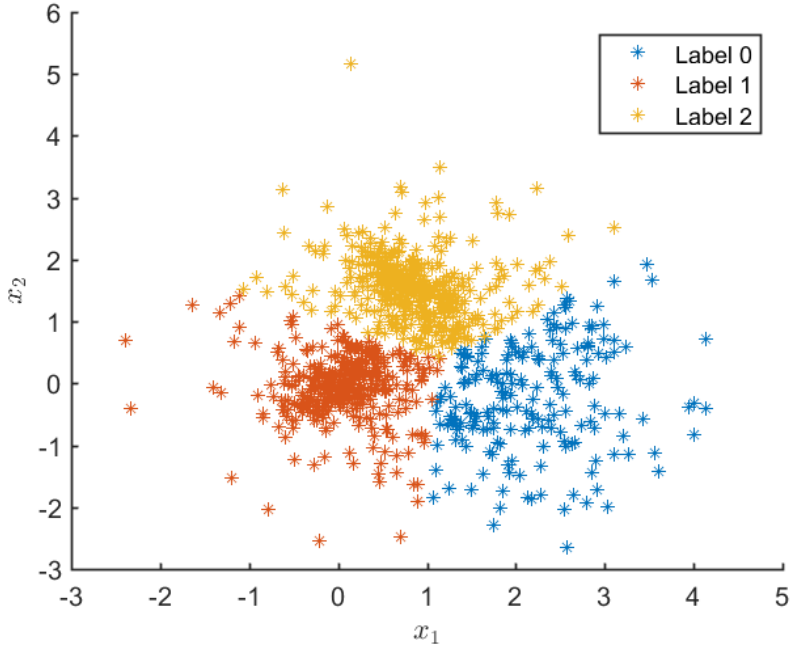
T given parameter

$$
\begin{aligned}
&\text{for } t = 1, \ldots T/4: && x_t \sim \mathcal{N}(\mu_1, \Sigma_1) \\
&\text{for } t = T/4+1, \ldots 2T/4: && x_t \sim \mathcal{N}(\mu_2, \Sigma_2) \\
&\text{for } t = 2T/4+1, \ldots 3T/4: && x_t \sim \mathcal{N}(\mu_3, \Sigma_3) \\
&\text{for } t = 3T/4+1, \ldots T: && x_t \sim \mathcal{N}(\mu_4, \Sigma_4)
\end{aligned}
$$

$$
\mu_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \ \Sigma_1 = \begin{bmatrix} 0.1 & 0.05 \\ 0.05 & 0.1 \end{bmatrix}, \quad \mu_2 = \begin{bmatrix} 0.8 \\ 1.6 \end{bmatrix}, \ \Sigma_2 = \begin{bmatrix} 0.1 & -0.05 \\ -0.05 & 0.1 \end{bmatrix},
$$

$$
\mu_3 = \begin{bmatrix} 1.6 \\ 0 \end{bmatrix}, \ \Sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mu_4 = \begin{bmatrix} 0.8 \\ 0.8 \end{bmatrix}, \ \Sigma_4 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
$$





The first (and the simplest) implementation

set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

$\quad$ $it = it + 1$

endwhile

```
Gamma = zeros(K,T);
for t=1:T
    randidx = randi([1,K],1,1);
    Gamma(randidx,t) = 1;
end
```

The first (and the simplest) implementation

```
it = 0;
L = Inf;
while it < 1000

    ....

    L_old = L;
    L = compute_L(X,Theta,Gamma);

    % check stopping criteria
    Ldelta = L_old - L;
    if Ldelta < 1e-6
        break;
    end

    it = it + 1;
end
```

set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

$\quad$ $it = it + 1$

endwhile

The first (and the simplest) implementation

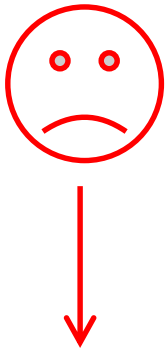
set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$
  solve $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*
  solve $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*
  $it = it + 1$
endwhile

$$\forall k : \theta_k^* = \frac{\sum_{t=1}^{T} \gamma_{k,t} x_t}{\sum_{t=1}^{T} \gamma_{k,t}}$$

```
for k=1:K
    sumgamma = sum(Gamma(k,:));
    if sumgamma ~= 0
        for n = 1:N
            Theta(n,k) = dot(X(n,:),Gamma(k,:))/sumgamma;
        end
    end
end
```

The first (and the simplest) implementation



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

solve $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

solve $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

$it = it + 1$

endwhile

$$\forall k, t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \underbrace{\|x_t - \theta_{\hat{k}}\|^2}_{=: g_k(t)} \\ 0 & \text{otherwise} \end{cases}$$

```
for t=1:T
    g = zeros(K,1);
    for k=1:K
        g(k) = dot(X(:,t) - Theta(:,k),X(:,t) - Theta(:,k));
    end

    Gamma(:,t) = zeros(K,1);
    [~,minidx] = min(g);
    Gamma(minidx,t) = 1;
end
```

The first (and the simplest) implementation

set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

  solve $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

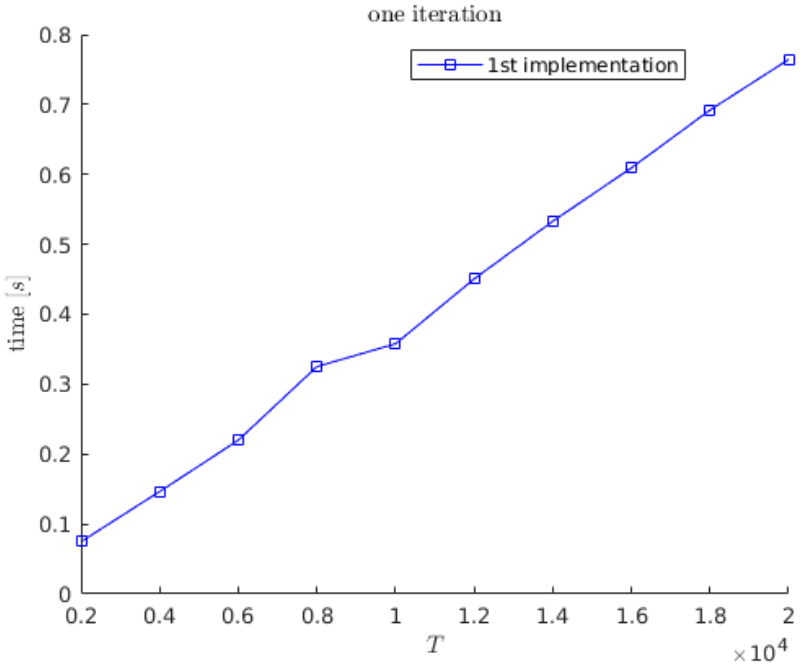
  solve $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

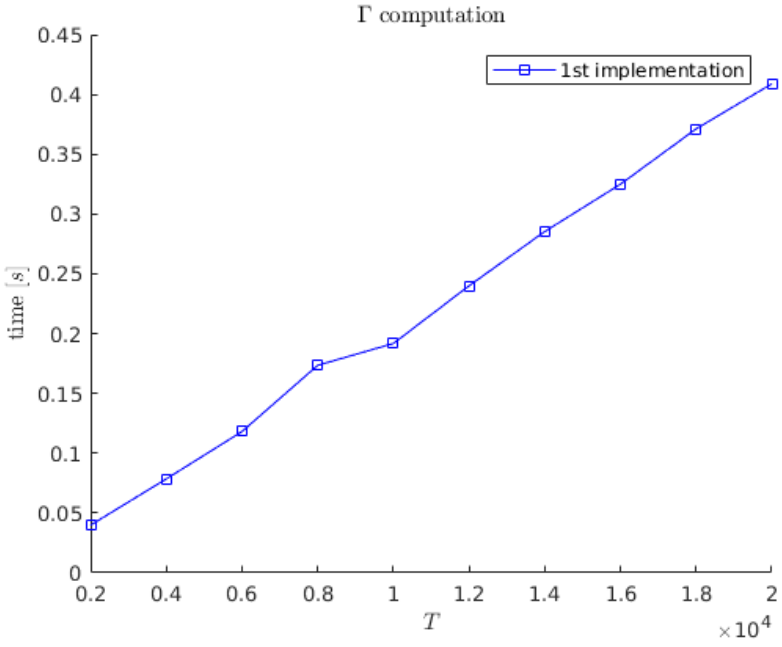
  $it = it + 1$

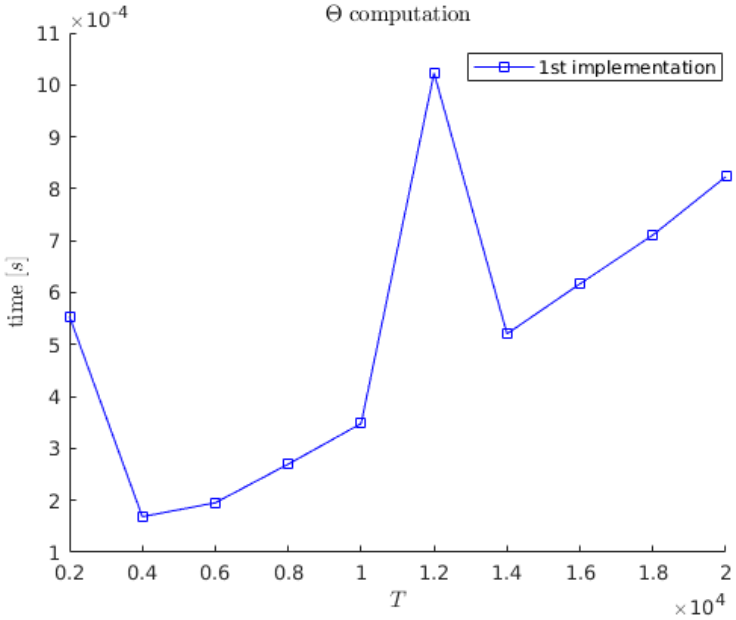
endwhile

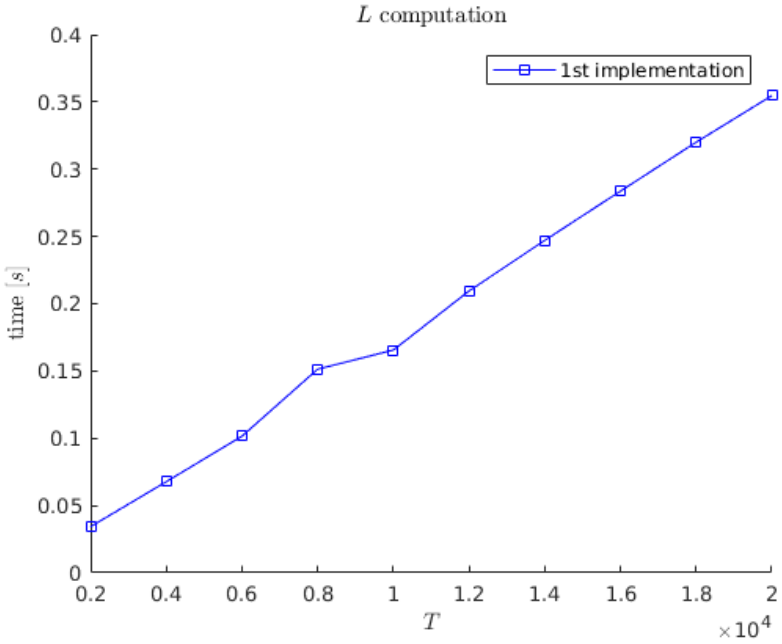
$$L(\Theta, \Gamma) = \frac{1}{Tn} \sum_{k=1}^{K} \sum_{t=1}^{T} \gamma_{k,t} \cdot \|x_t - \theta_k\|^2$$

```
L = 0;
for k=1:K
    for t=1:T
        L = L + ...
          Gamma(k,t)*dot(X(:,t) - Theta(:,k),X(:,t) - Theta(:,k));
    end
end
L = L/(T*N);
```


one iteration
1st implementation
time [s]
T
×10^4


Γ computation
1st implementation
time [s]
T
×10^4

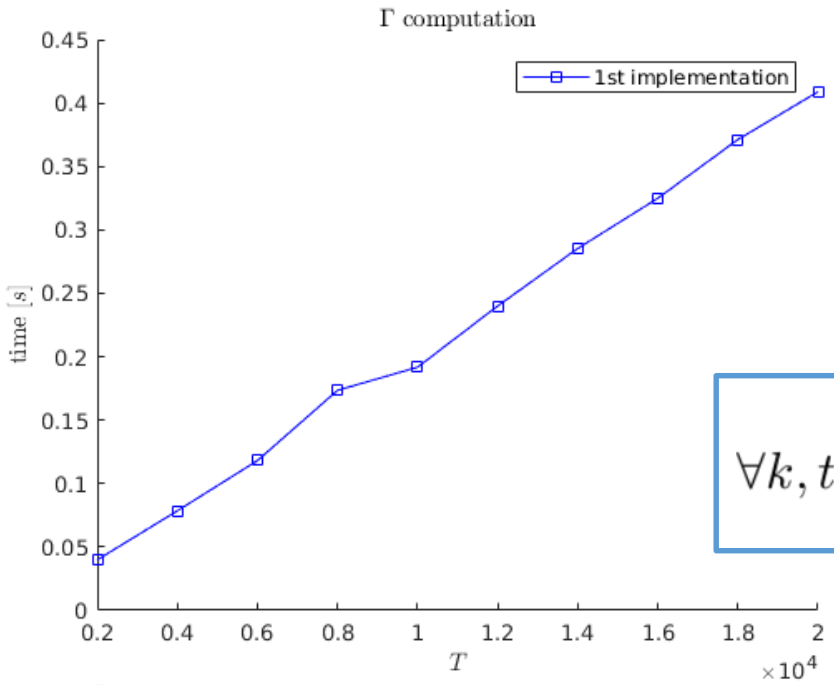

Θ computation
×10^-4
1st implementation
time [s]
T
×10^4


L computation
1st implementation
time [s]
T
×10^4

3.) Implementation of K-means algoritm

[b] vectorization

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

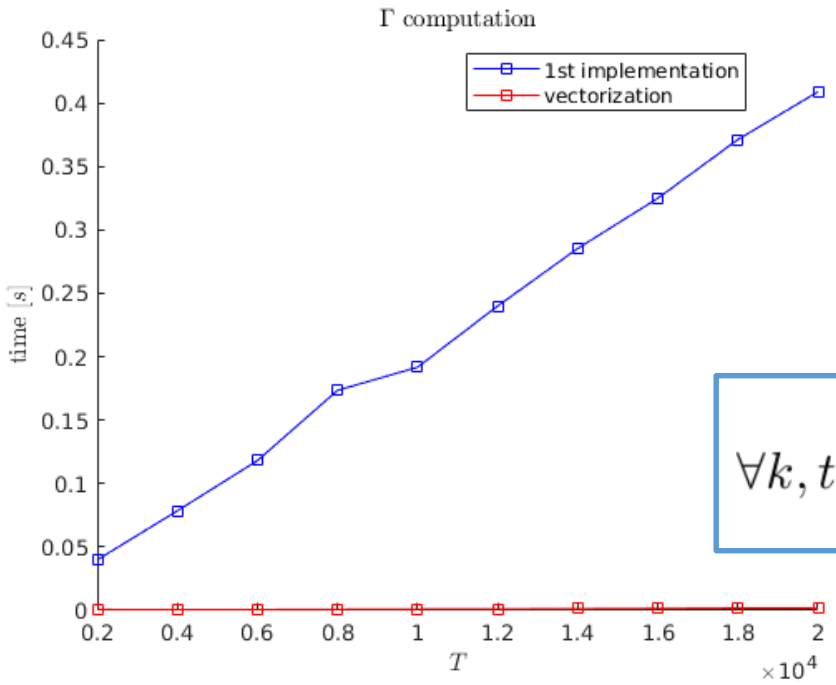
$\quad$ $it = it + 1$

endwhile

$$\forall k, t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \underbrace{\|x_t - \theta_{\hat{k}}\|^2}_{=: g_k(t)} \\ 0 & \text{otherwise} \end{cases}$$

```
for t=1:T
    g = zeros(K,1);
    for k=1:K
        g(k) = dot(X(:,t) - Theta(:,k),X(:,t) - Theta(:,k));
    end

    Gamma(:,t) = zeros(K,1);
    [~,minidx] = min(g);
    Gamma(minidx,t) = 1;
end
```

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

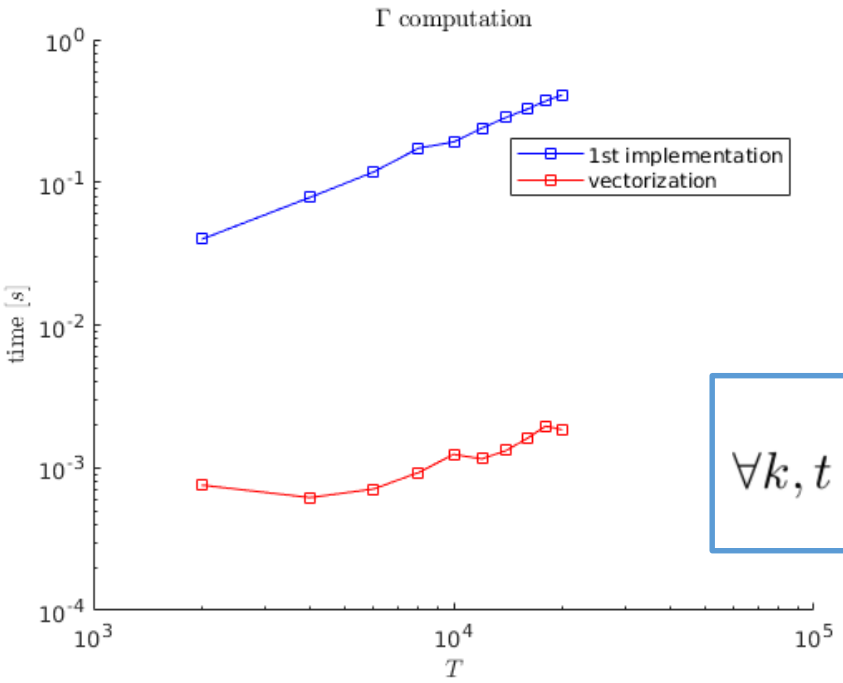
$\quad$ $it = it + 1$

endwhile

$$\forall k, t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \underbrace{\|x_t - \theta_{\hat{k}}\|^2}_{=: g_k(t)} \\ 0 & \text{otherwise} \end{cases}$$

```
g = zeros(size(Gamma));
for k=1:K
    g(k,:) = dot(X - Theta(:,k),X - Theta(:,k));
end
[~,idx] = min(g,[],1);
Gamma = zeros(K,T);
for k=1:K
    Gamma(k,idx==k) = 1;
end
```

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

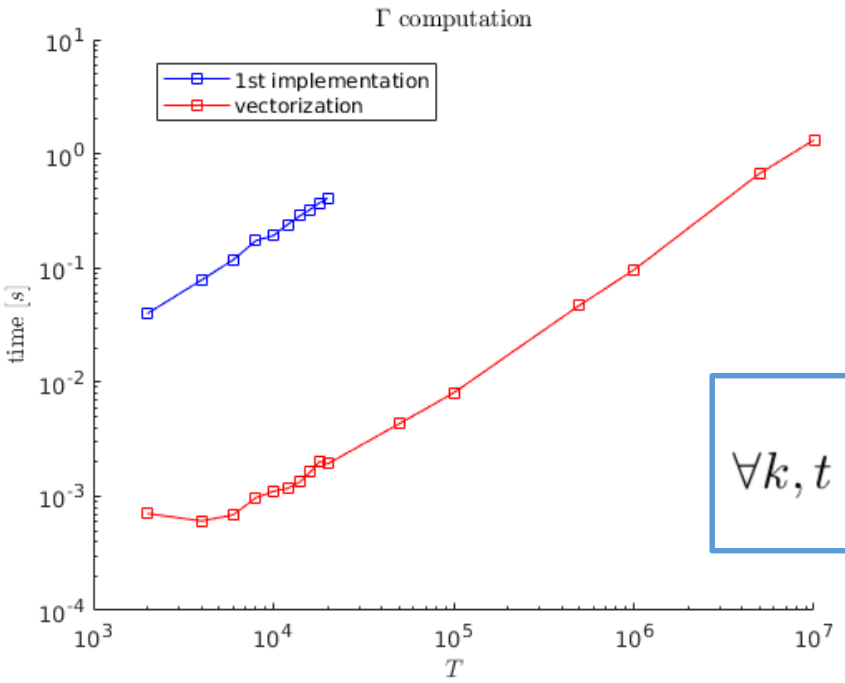
$\quad$ $it = it + 1$

endwhile

$$\forall k, t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \underbrace{\|x_t - \theta_{\hat{k}}\|^2}_{=: g_k(t)} \\ 0 & \text{otherwise} \end{cases}$$

```
g = zeros(size(Gamma));
for k=1:K
    g(k,:) = dot(X - Theta(:,k),X - Theta(:,k));
end
[~,idx] = min(g,[],1);
Gamma = zeros(K,T);
for k=1:K
    Gamma(k,idx==k) = 1;
end
```

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

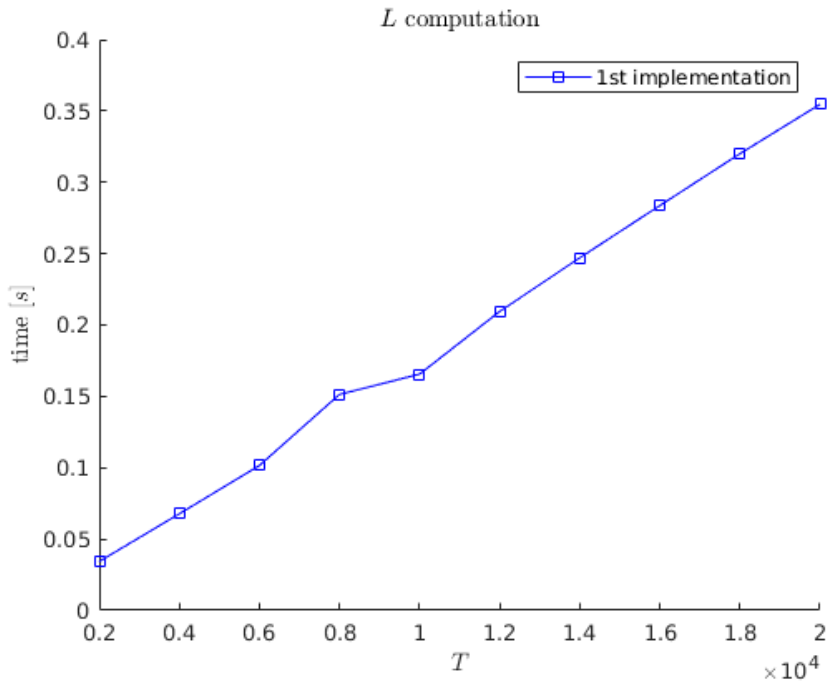
$\quad$ $it = it + 1$

endwhile

$$\forall k, t : \gamma^*_{k,t} = \begin{cases} 1 & \text{if } k = \arg_{\hat{k}} \min \underbrace{\|x_t - \theta_{\hat{k}}\|^2}_{=: g_k(t)} \\ 0 & \text{otherwise} \end{cases}$$

```
g = zeros(size(Gamma));
for k=1:K
    g(k,:) = dot(X - Theta(:,k),X - Theta(:,k));
end
[~,idx] = min(g,[],1);
Gamma = zeros(K,T);
for k=1:K
    Gamma(k,idx==k) = 1;
end
```

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

solve $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

solve $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

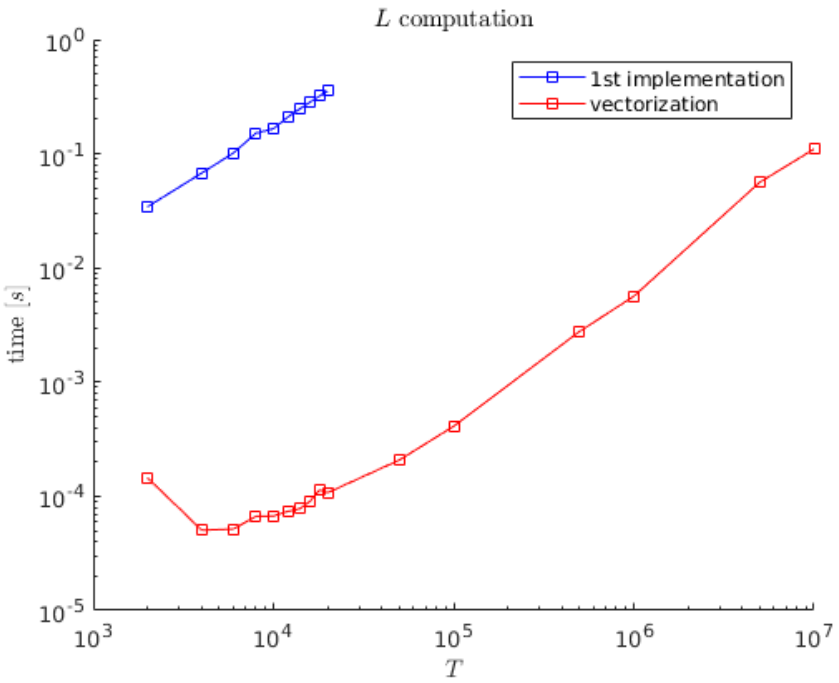
$it = it + 1$

endwhile

$$L(\Theta, \Gamma) = \frac{1}{Tn} \sum_{k=1}^{K} \sum_{t=1}^{T} \gamma_{k,t} \cdot \|x_t - \theta_k\|^2$$

```
L = 0;
for k=1:K
    for t=1:T
        L = L + ...
            Gamma(k,t)*dot(X(:,t) - Theta(:,k),X(:,t) - Theta(:,k));
    end
end
L = L/(T*N);
```

Vectorization in T



set feasible initial approximation Γ_0

while $\|L(\Gamma_{it}, \theta_{it}) - L(\Gamma_{it-1}, \theta_{it-1})\| > \varepsilon$

$\quad$ *solve* $\theta_{it} = \arg\min_{\theta} L(\theta, \Gamma_{it})$ *(with fixed* Γ_{it}*)*

$\quad$ *solve* $\Gamma_{it} = \arg\min_{\Gamma \in \Omega_\Gamma} L(\theta_{it}, \Gamma)$ *(with fixed* θ_{it}*)*

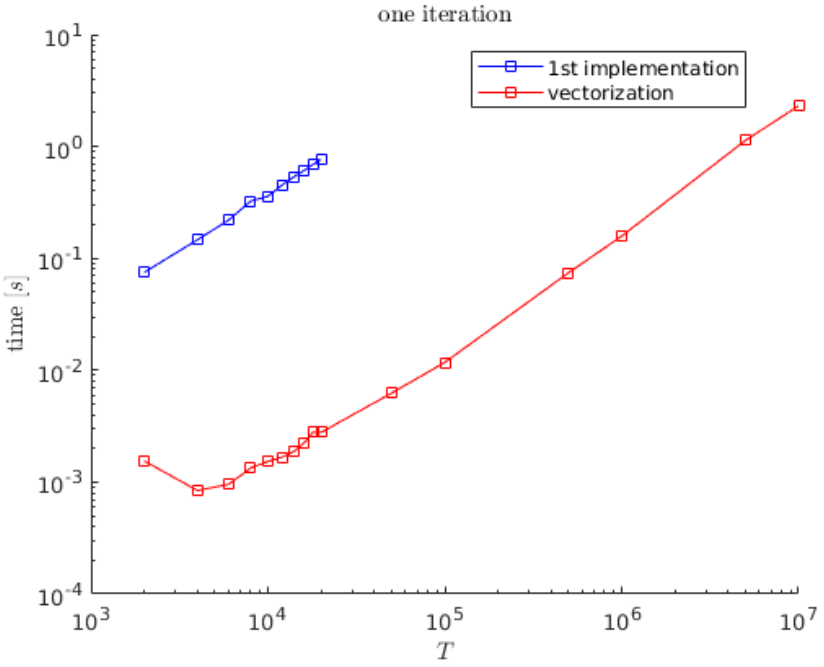
$\quad$ $it = it + 1$

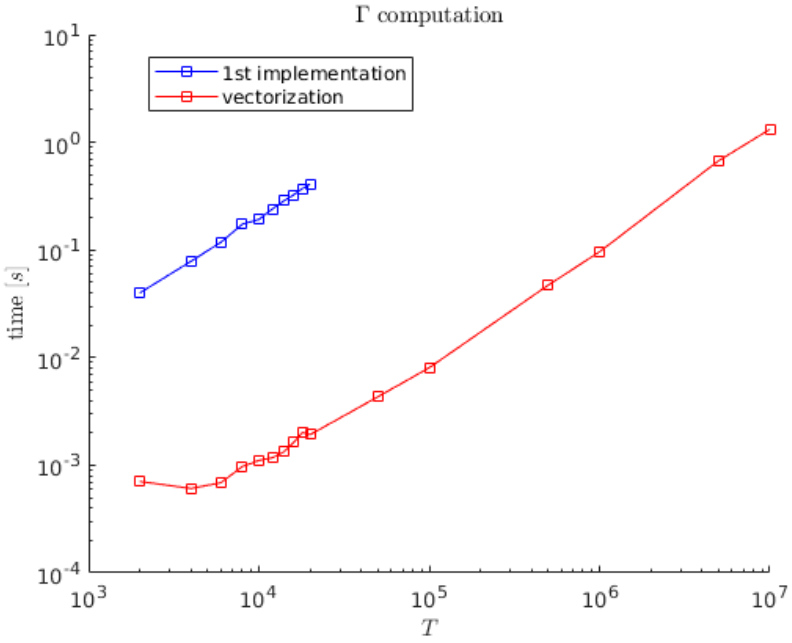
endwhile

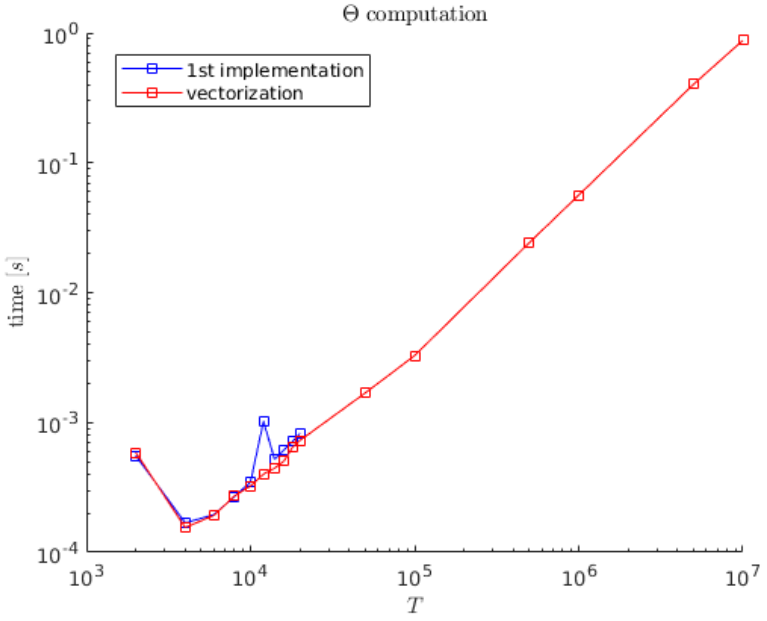
$$L(\Theta, \Gamma) = \frac{1}{Tn} \sum_{k=1}^{K} \sum_{t=1}^{T} \gamma_{k,t} \cdot \underbrace{\|x_t - \theta_k\|^2}_{=: g_k(t)}$$

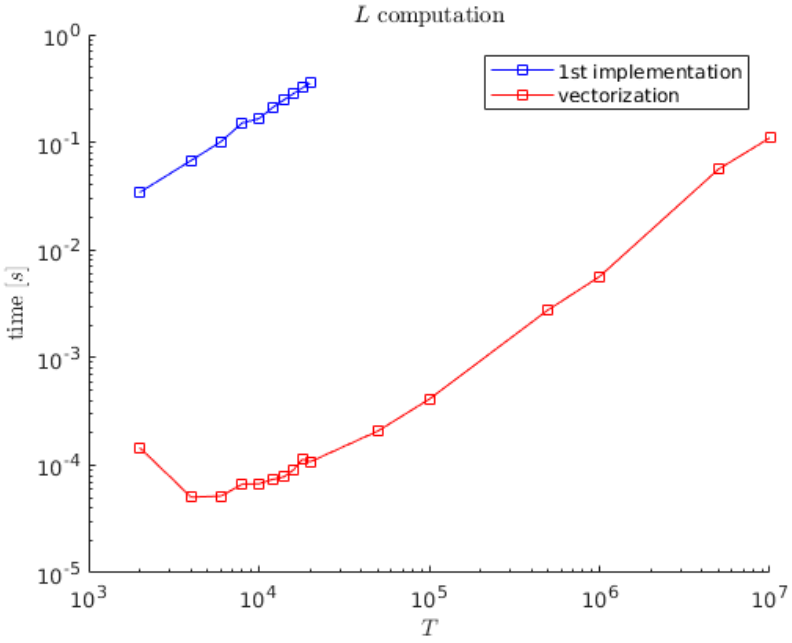
```
L = sum(sum(bsxfun(@times,Gamma,g)))/(T*N);
```

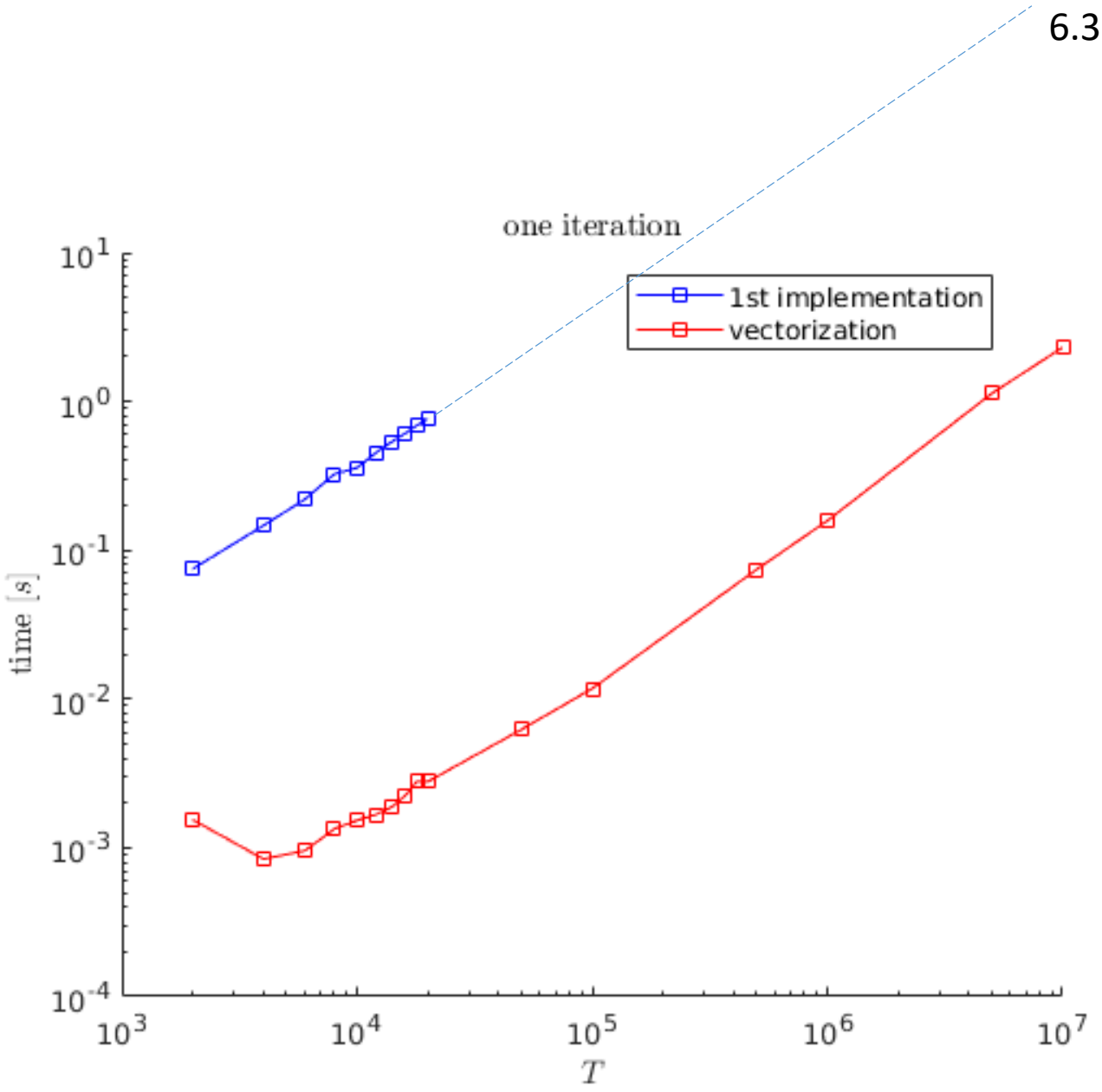
already computed in Γ step


one iteration
1st implementation
vectorization
time [s]
T


Γ computation
1st implementation
vectorization
time [s]
T


Θ computation
1st implementation
vectorization
time [s]
T


L computation
1st implementation
vectorization
time [s]
T


6.3 minutes
one iteration
1st implementation
vectorization
2 seconds
time [s]
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
T

3.) Implementation of K-means algoritm

[c] GPU CUDA computation

CUDA in Matlab

Plan:

- transfer data to GPU
- take our vectorized code, change BLAS to CUBLAS
- transfer results back to CPU
- maybe some other GPU-related stuff?

CUDA in Matlab

Plan:

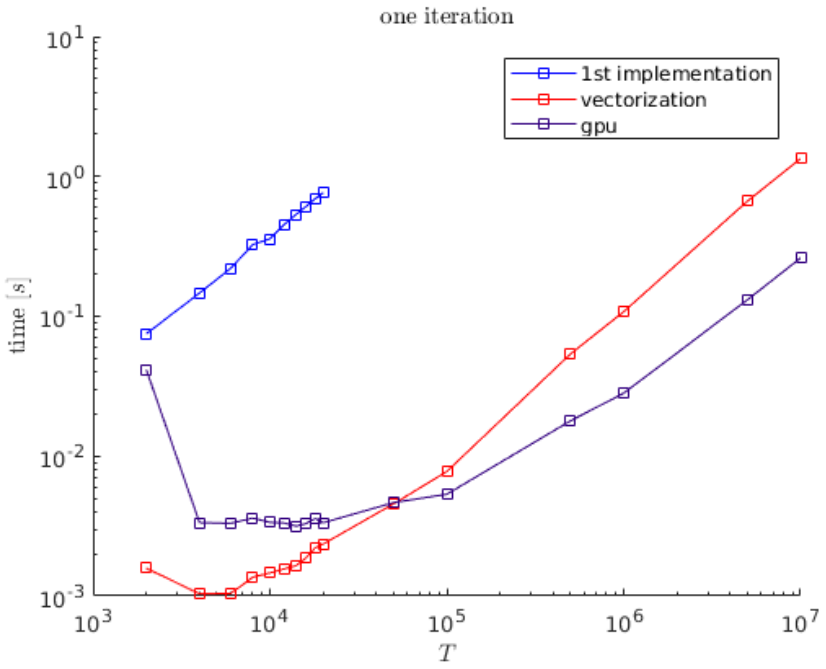
- transfer data to GPU
- take our vectorized code, change BLAS to CUBLAS
- transfer results back to CPU
- maybe some other GPU-related stuff?

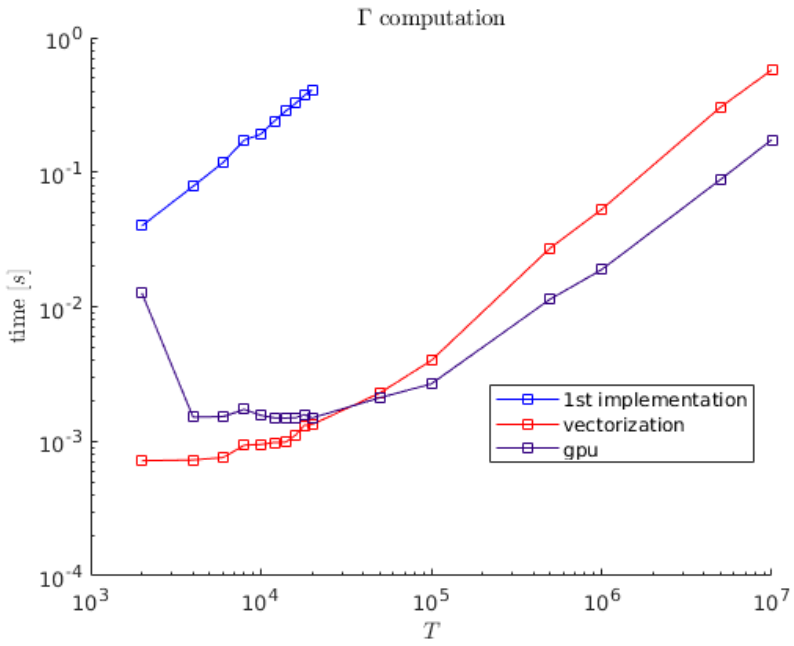
```
% generate problem
Xcpu = generate_clustering(T,N);
X = gpuArray(Xcpu);
```

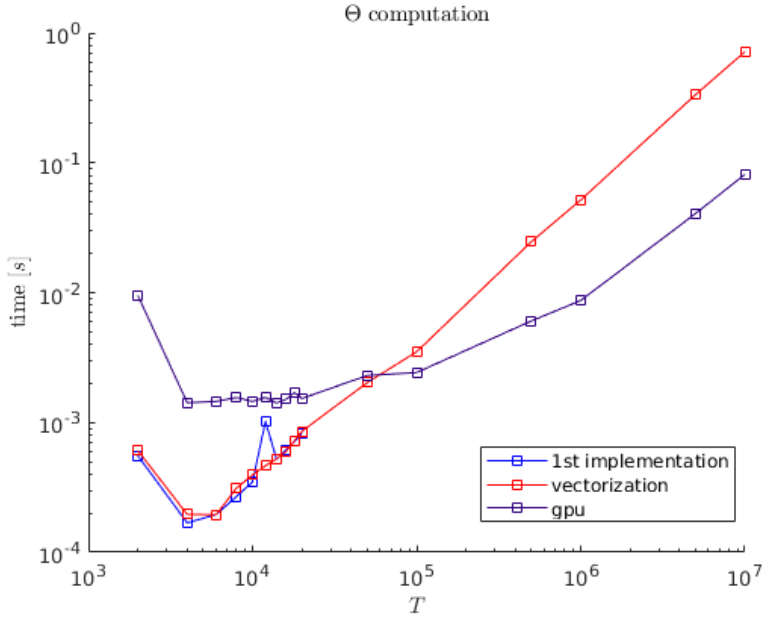
```
Theta = gpuArray.zeros(N,K);
g = gpuArray.zeros(size(Gamma));
Gamma = gpuArray.zeros(K,T);
```

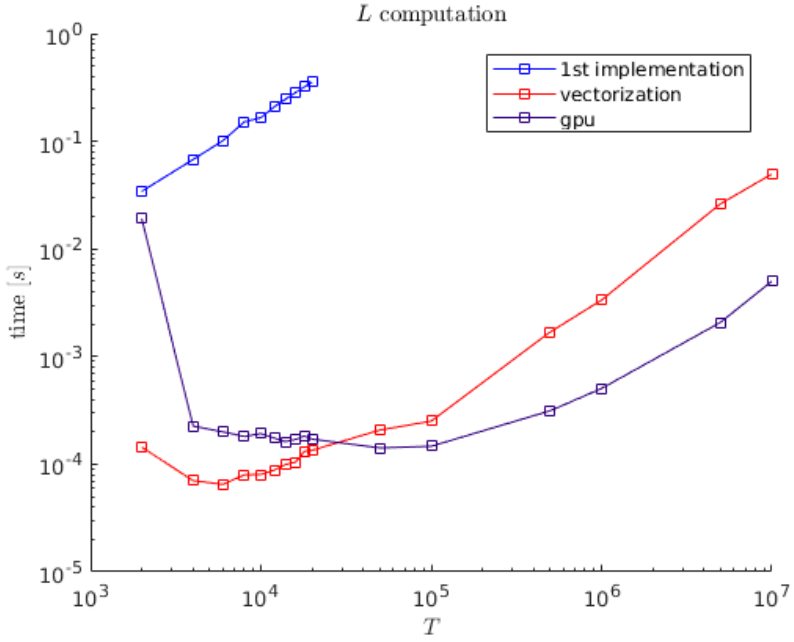
```
Gammacpu = gather(Gamma);
```

```
% prepare device
gpu = gpuDevice(1);
reset(gpu);


wait(gpu);
tic;
... computation on GPU ...
wait(gpu);
mytime = toc;
```


one iteration
1st implementation
vectorization
gpu
time [s]
T


Γ computation
1st implementation
vectorization
gpu
time [s]
T


Θ computation
1st implementation
vectorization
gpu
time [s]
T


L computation
1st implementation
vectorization
gpu
time [s]
T

CUDA in Matlab

- we can write our own CUDA kernels

```
% define kernel as matlab function
kernel = parallel.gpu.CUDAKernel( 'mykernel.ptx', 'mykernel.cu' );

% compute optimal number of threads and grid
kernel.ThreadBlockSize = kernel.MaxThreadsPerBlock;
kernel.GridSize = ceil(T/kernel.MaxThreadsPerBlock);

% call cuda kernel
[Gamma,g] = feval( kernel, Gamma, g, Theta, X, N, K, T );
```

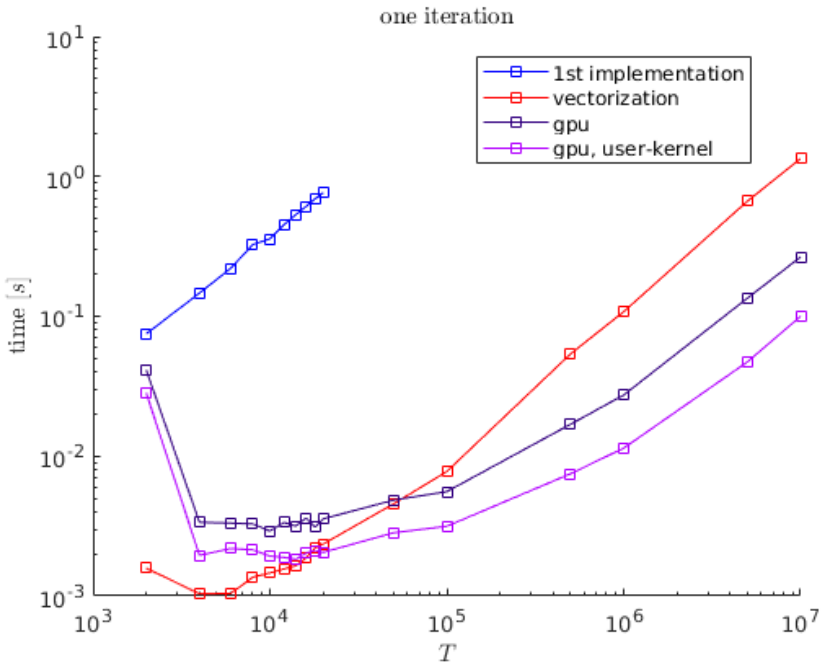
in C/C++: "mykernel<<<gridSize, blockSize>>>(Gamma, g, Theta, X, N, K, T); "

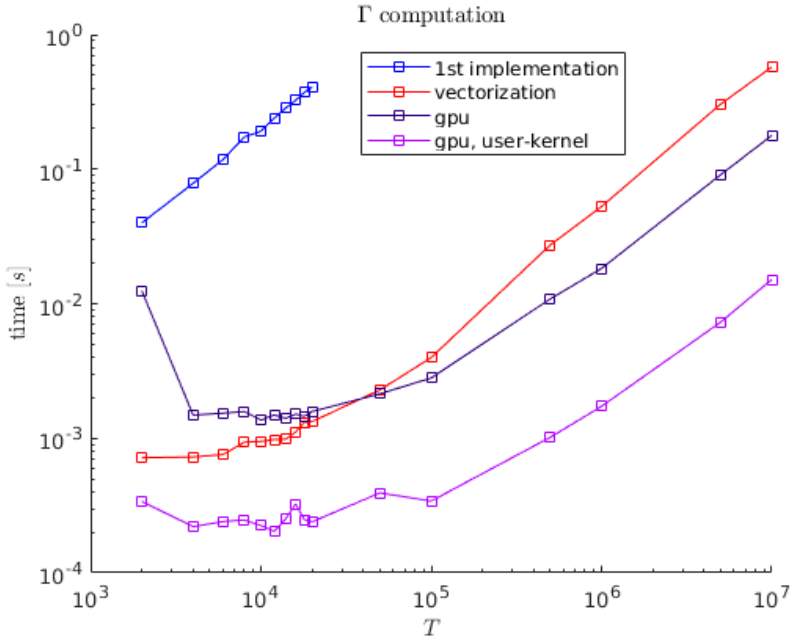
CUDA in Matlab

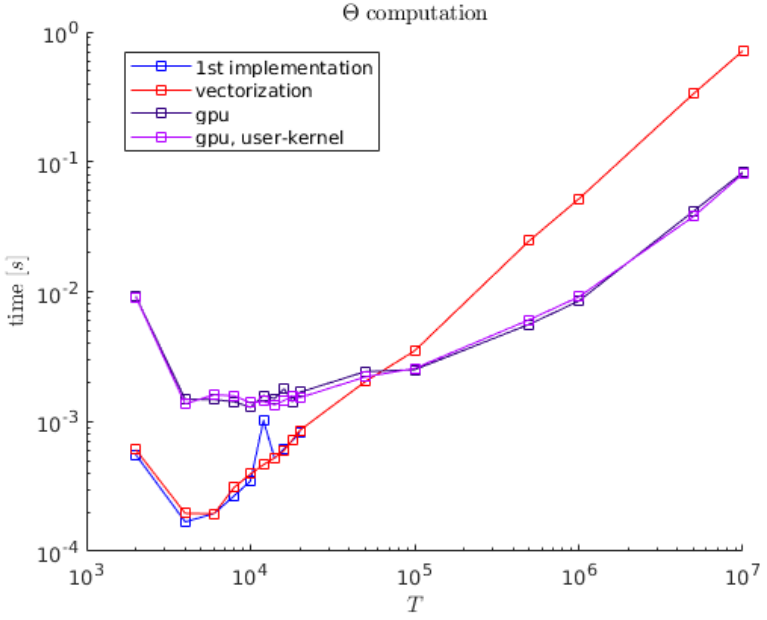
- we can write our own CUDA kernels

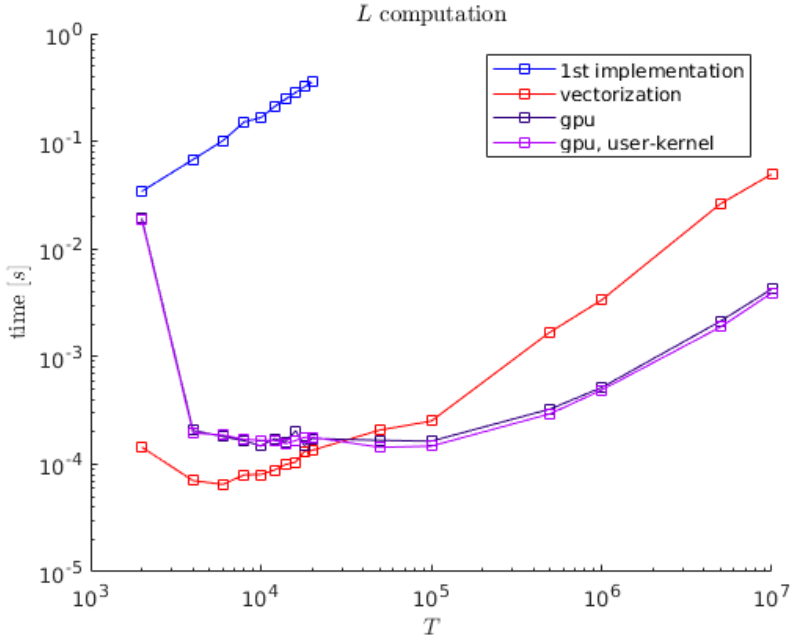
```
__global__ void mykernel( double *Gamma, double *g,
                          double *Theta, double *X, int N, int K, int T) {

    /* compute kernel index */
    int t = blockIdx.x*blockDim.x + threadIdx.x;

    if(t<T){
        int mink;

        /* compute g(:,t) */
        for(int k=0;k<K;k++){
            /* compute dot product g(k,t) = <X(:,t) - Theta(:,k),X(:,t) - Theta(:,k)> */
            g[t*K+k] = 0;
            for(int n=0;n<N;n++){
                g[t*K+k] += (X[t*N+n] - Theta[k*N+n])*(X[t*N+n] - Theta[k*N+n]);
            }

            /* if this is first row, then Gamma(k,t) is minimal value */
            if(k==0){
                mink=0; /* index k with min value of g(:,t) */
                Gamma[t*K+k] = 1;
            } else {
                /* is this smaller value then previous one? */
                if(g[t*K+k] < g[t*K+mink]){
                    /* old one is not min, set it equal to zero */
                    Gamma[t*K+mink] = 0;
                    mink=k;
                    Gamma[t*K+k] = 1;
                } else {
                    /* it is not min */
                    Gamma[t*K+k] = 0;
                }
            }
        }

    }
}
```

nvcc -ptx mykernel.cu


one iteration
1st implementation
vectorization
gpu
gpu, user-kernel
time [s]
T


Γ computation
1st implementation
vectorization
gpu
gpu, user-kernel
time [s]
T


Θ computation
1st implementation
vectorization
gpu
gpu, user-kernel
time [s]
T


L computation
1st implementation
vectorization
gpu
gpu, user-kernel
time [s]
T