TT-toolbox 2.1: Fast multidimensional array operations in TT format

I.V. Oseledets

May 2011

1 Contributors

The toolbox is written by Ivan Oseledets (Institute of Numerical Mathematics, Moscow, Russia), with contributions from Olga Lebedeva, Sergei Dolgov,Vladimir Kazeev, Dmitry Savostyanov (all are from INM RAS Moscow), and ideas from

2 What is the TT-format for tensors

Tensor **A** is said to be in the TT-format, if

$$A(i_1, i_2, \dots, i_d) = G_1(i_1) \dots G_d(i_d),$$

and $G_k(i_k)$ is a $r_{k-1} \times r_k$ matrix, and $r_0 = r_d = 1$. The approximation in this format is known to be stable and can be based on QR and SVD decompositions. In linear algebra the most important operation is probably matrix-by-vector product. Thus, matrices have to be also represented in TT-format. Vector of length $n_1 \dots n_d$ is said to be in the TT-format, if it has low TT-ranks considered as d-dimensional array (in MATLAB it is just a single call the the `reshape` function. Matrices acting on such vectors have size $M \times N$, where $N = \prod_{k=1}^{d} n_k$. For simplicity assume they are square, then each element of such matrix can be indexed by $i_1, \dots, i_d, j_1, \dots, j_d$, where multiindex $i_k, k = 1, \dots, d$ corresponds to rows, and $j_k, k = 1, \dots, d$ — to columns of the matrix. The matrix M is said to be in the TT-format if

$$M(i_1, \dots, i_d, j_1, \dots, j_d) = M_1(i_1, j_1) M_2(i_2, j_2) \dots M_d(i_d, j_d),$$

and $M_k(i_k, j_k)$ is a $r_{k-1} \times r_k$ matrix.

3 What is new in version 2.0

TT Toolbox version 2.0 introduces several major innovations compared to version 1.0.

1. New classes `tt_tensor` and `tt_matrix`, which now represent TT-tensors and TT-matrices.
2. Object-oriented approach allowed to overload many standard MATLAB functions, including addition, subtraction, multiplication by number, scalar product, norm, Kronecker products, matrix-by-vector product etc. For details see description below.
3. Complex arithmetic is now fully supported
4. New advanced subroutines are introduced, based on the QTT-DMRG approach: fast approximate matrix-by-vector product, computation of functions of TT-tensors using cross-DMRG algorithm, solution of linear systems via DMRG-solve algorithm and solution of eigenvalue problems via classical DMRG algorithm.
5. Several subroutines to generate basic TT-tensors and TT-matrices: matrices of all ones, identity matrix, random TT-tensors with fixed core sizes

4 Illustration of basic functionality

TT-Toolbox 2.0 has two main classes: `tt_tensor` and `tt_matrix`. The first is a TT-representation of a d-dimensional array in TT-format, and the second — of d-level matrix in TT-format. From old cell-array representations, used in TT-Toolbox 1.0, these representations can be obtained via calls to `tt_tensor` and `tt_matrix` constructors:

```
 >> d=5; A=tt_ones(2,d) %Generate a d-dimensional
                        %tensor of all ones
A =
    [2x1 double]
    [2x1 double]
    [2x1 double]
    [2x1 double]
    [2x1 double]
```

And conversion to new format is done via command

```
>> B=tt_tensor(A)
B a 5-dimensional TT-tensor, ranks and mode sizes:
r(1)=1  n(1)=2
r(2)=1  n(2)=2
r(3)=1  n(3)=2
r(4)=1  n(4)=2
r(5)=1  n(5)=2
r(6)=1
```

For matrices in the TT-format situation is analogous, for example,

```
>> e=tt_eye(d,2); %Generate 2^d x 2^d identity matrix
                  %in TT-format
>> C=tt_matrix(e)
C is a 5-dimensional TT-matrix, ranks and mode sizes:
r(1)=1  n(1)=2  m(1)=2
r(2)=1  n(2)=2  m(2)=2
r(3)=1  n(3)=2  m(3)=2
r(4)=1  n(4)=2  m(4)=2
r(5)=1  n(5)=2  m(5)=2
r(6)=1
```

The conversion operations in the opposite direction are realized via `core` function:

```
>> e=core(C);
>> A=core(B);
```

This helps to keep compatibility with previously developed algorithms. All previous functions are kept in the TT Toolbox 2.0 and can be applied. The `core` command is also overloaded to get a specific core of the TT-decomposition (in the new ordering of indices, it returns a tensor of size $r_{k-1} \times n_k \times r_k$:

```
>> cr=core(B,3); %Get core number 3
>> size(cr)
ans =
     5     2     5
```

For `tt_tensor` and `tt_matrix` classes basic arithmetic operations are defined:

```
 >> d=7;
 >> v=tt_random(2,d,5); %Generate a random
```

```
                               %d-dimensional tensor
                               %with mode sizes 2 and ranks 5
 >> v1=tt_random(2,d,5); %Generate a random
                          %d-dimensional tensor
                          %with mode sizes 2 and ranks 5
 >> v=tt_tensor(v); %Make it a TT-tensor
 >> v1=tt_tensor(v1);
 >> v2=v+2*v1 %Linear combinations

v2 is a 7-dimensional TT-tensor, ranks and mode sizes:
 r(1)=1   n(1)=2
r(2)=10   n(2)=2
r(3)=10   n(3)=2
r(4)=10   n(4)=2
r(5)=10   n(5)=2
r(6)=10   n(6)=2
r(7)=10   n(7)=2
 r(8)=1
```

Note that ranks are added, as expected. The same can be done for matrices with the same syntax.

The computations with TT-tensors are impossible without *TT-rounding*. The tensor is approximated by another TT-tensor with smaller TT-ranks but with prescribed accuracy ε. It is implemented in the `round` command (old syntax is the `tt_compr2` subroutine). For example,

```
 >> d=7; v=tt_random(2,d,5); v=tt_tensor(v); v=v+v
v2 is a 7-dimensional TT-tensor, ranks and mode sizes:
 r(1)=1   n(1)=2
r(2)=10   n(2)=2
r(3)=10   n(3)=2
r(4)=10   n(4)=2
r(5)=10   n(5)=2
r(6)=10   n(6)=2
r(7)=10   n(7)=2
 r(8)=1
```

but after rounding ranks are reduced:

```
 >> w=round(v,1e-12)
w is a 7-dimensional TT-tensor, ranks and mode sizes:
r(1)=1 n(1)=2
r(2)=2 n(2)=2
```

```
r(3)=4 n(3)=2
r(4)=5 n(4)=2
r(5)=5 n(5)=2
r(6)=4 n(6)=2
r(7)=2 n(7)=2
r(8)=1
```

Note that some ranks are reduced to 4 (maximal possible ranks). To check accuracy, overloaded functions `minus` and `norm` can be used:

```
>> norm(w-v)/norm(w)
ans =
    1.4258e-14
```

One can also check particular elements of the tensors w and v. This can be done either via direct indexing:

```
>> em1=w(1,1,2,1,1,1,2)
em1 =
    93.4955
>> em2=v(1,1,2,1,1,1,2)
em2 =
    93.4955
>> em1-em2
ans =
    1.4211e-14
```

This is an indicator of the fact that v and w differ by machine precision. The second options is to use linear index array:

```
>> ind=[1,1,2,1,1,1,2]; em1=w(ind)
em1 =
    93.4955
```

which has to be used if the array is of large or variable dimension, and direct indexing is not possible. Also, taking subsets is also acceptable, for example command

```
>> w1=w(1,:,:,:,:,:,:);
```

will return TT-tensor $w1$ with the first mode size equal to 1. The `tt_tensor` and `tt_matrix` classes have several constructors. One of the most important is the conversion from the full array, which can be used to detect TT-structures in given tensors. For example, to get a QTT-representation of a function $\sqrt{x}$ on $[0,1]$, the following code can be used:

```
>> d=10; n=2^d; h=1.0./(n-1); x=(0:n-1)*h; %Get x
>> x=sqrt(x); x=reshape(x,2*ones(1,d)); %Quantization
>> tt=tt_tensor(x,1e-8)
tt is a 10-dimensional TT-tensor, ranks and mode sizes:
 r(1)=1  n(1)=2
 r(2)=2  n(2)=2
 r(3)=4  n(3)=2
 r(4)=6  n(4)=2
 r(5)=6  n(5)=2
 r(6)=6  n(6)=2
 r(7)=6  n(7)=2
 r(8)=6  n(8)=2
 r(9)=4  n(9)=2
r(10)=2 n(10)=2
r(11)=1
```

To check accuracy, conversion back from TT-format to full format can be used:

```
>> y=full(x); %The result is a d-dimensional tensor
>> y=y(:); norm(y-x(:))/norm(y)
ans =
   7.9424e-10
```

Thus, it is a very accurate approximation. Setting parameter ε to larger values will result in smaller ranks, but lower accuracy.

5 tt_tensor and tt_matrix storage scheme

Class tt_tensor contains the following fields:

1. tt.core — cores of the TT-decomposition stored in one "long" 1D array
2. tt.d — dimension of the array
3. tt.n — mode sizes of the array
4. tt.r — ranks of the decomposition
5. tt.ps — markers for position of the k-the core in array tt.core: if ps=tt.ps, then k-core can be obtained as

```
>> cr=tt.core; ps=tt.ps; corek=cr(ps(k):ps(k+1)-1);
```

Class `tt_matrix` contains three fields:

1. `ttm.n` — sizes of row indices
2. `ttm.m` — sizes of column indices
3. `ttm.tt` — TT-tensor of the vectorized TT-representation of the matrix

6 Basic functionality

TT-Toolbox supports several basic functions for matrices and vectors. The following operations are supported:

1. `tt=tt_tensor(y,eps)` — construct TT-tensor from full array y with accuracy `eps`.
2. `ttm=tt_matrix(y,eps)` — construct TT-matrix from full array of dimension $n_1 \times n_2 \ldots \times n_d \times m_1 \ldots \times m_d$ with accuracy `eps`.
3. `tt=tt_tensor(TT)` — constructs TT-tensor from the TT-Toolbox 1.0 format.
4. `ttm=tt_matrix(TTM)` — constructs TT-matrix from the TT-Toolbox 1.0 format.
5. `tt=core(tt,k)`, `ttm=core(ttm,k)` — constructs TT-Toolbox 1.0 format from TT-Toolbox 2.0 format, if parameter k is not specified, and k-th core of the decomposition, if it is specified.
6. `y=full(tt)` — converts TT-tensor `tt` a full array.
7. `y=full(ttm)` — converts TT-matrix `ttm` to a full square matrix.
8. `tt=round(tt,eps)` — approximates given TT-tensor with another TT-tensor with smaller ranks but with prescribed accuracy `eps`.
9. Binary operations `tt1`$\cdot$`tt2`, where $\cdot$ can be any operation from the set $\{+, -, .*\}$ (plus,minus, elementwise product). They are implemented when both iterands are either TT-tensors or TT-matrices.
10. `r=rank(tt,k)`, `r=rank(ttm,k)` — returns all ranks of the TT-decomposition if k is not specified, and the k-th rank, if it is given.
11. `sz=size(tt)`, `sz=size(ttm)` — returns the size of the array. For the TT-tensors, it returns d integers, for TT-matrix it returns $d \times 2$ array of interegers.

12. `mm=mem(tt), mm=mem(ttm)` — memory required to store the TT-tensor

13. `er=erank(tt)` — effective rank of the TT-tensor.

14. Matrix-by-vector product: A$\star$b, where A is a `tt_matrix` and b is a `tt_tensor` of appropriate sizes;

15. Matrix-by-matrix product: A$\star$B, where A is a `tt_matrix` and B is a `tt_matrix`.

16. Matrix by full vector product: A$\star$b, where A is a `tt_matrix` and b is a full vector of size $\prod_{k=1}^{d} m_k$.

17. `p=dot(tt1,tt2)` — dot (scalar) product of two TT-tensors

18. `p=norm(tt)` — Frobenius norm of the TT-tensor.

19. `elem=tt(i1,i2,...,id)` — computes element of the TT-tensor in position `i1,i2,...,id`.

20. `elem=tt(ind)`, where `ind` is an integer array of length d return element of the TT-tensor in the position specified by multiindex `ind`.

7 More functions

There are several functions that are helpful in working with TT-tensors and TT-matrices.

21. `ttm=diag(tt)`, `tt=diag(ttm)` — constructs either diagonal TT-matrix from TT-tensor, or takes diagonal of a TT-matrix.

22. `a=kron(b,c)`. For b, c being TT-tensors, it computes outer product of them with number of dimensions equal to the sum of the number of dimensions of b and c. For TT-matrices it computes their Kronecker product in TT-format.

23. `tt=tt_random(d,n,r)` — generate random TT-tensor with dimension d, mode size n, ranks r. n and r can be either numbers (then all dimensions and ranks are the same) or arrays of integers. NOTE: the `tt` is in the OLD format to ensure compatibility. To convert to the new format use `tt_tensor` constructor. In future releases this function will return TT-tensor object.

24. `tt=tt_ones(n,d)` generate tensor with mode sizes n, dimension d of all ones. Returns result in the old format

25. `tt=tt_eye(d,n)` generate identity matrix with dimension d and mode size n. n can be either a number, or integer array. Returns result in the old format.

26. `tt=tt_qlaplace_dd(d)` – generate Laplacian operator with Dirichlet BC on a grid with $2^{d_1} \times 2^{d_2} \times \ldots$. Returns result in the old format.

27. `tt=tt_x(d,n)` — returns QTT representation of vector $1 : n^d$ (in the OLD format).

Using these subroutines one can easily implement, say, iterative methods for matrix problems. For example, to implement power method for the maximal eigenvalue for 2D-Laplacian, the following code can be used:

```
%power_iter.m --- simple power iteration
d=10;
mt=tt_qlaplace_dd(d,d);
mt=tt_matrix(mt); %Create 2D Laplacian
                  %on 2^d × 2^d grid with Dirichlet BC
v=tt_ones(2*d,2);
v=tt_tensor(v); %Create vector of all ones
niter=4000; eps=1e-6;
tic;
for i=1:niter
  v=v/norm(v); v1=mt*v; ev=dot(v1,v);
  v=round(v,eps); %Round to avoid rank growth
  if ( ¬mod(i,400) )
     fprintf('iter=%d ev=%d rank=%3.1f \n',i,ev,erank(v))
  end
end
toc
```

This is of course not the best way to compute the maximal eigenvalue and is presented only to illustrate how the Toolbox works. Actual results are

```
>> power_iter
iter=400 ev=7.969962e+00 rank=4.9
iter=800 ev=7.984991e+00 rank=4.6
iter=1200 ev=7.989996e+00 rank=4.6
iter=1600 ev=7.992498e+00 rank=4.6
iter=2000 ev=7.993998e+00 rank=4.4
iter=2400 ev=7.994999e+00 rank=4.4
iter=2800 ev=7.995714e+00 rank=4.5
iter=3200 ev=7.996249e+00 rank=4.7
```

```
iter=3600 ev=7.996666e+00 rank=4.7
iter=4000 ev=7.997000e+00 rank=4.6
Elapsed time is 33.076830 seconds.
```

If we set $d = 20$ then the output is

```
>> power_iter
iter=400 ev=7.969963e+00 rank=4.7
iter=800 ev=7.984991e+00 rank=4.0
iter=1200 ev=7.989996e+00 rank=3.9
iter=1600 ev=7.992498e+00 rank=3.9
iter=2000 ev=7.993999e+00 rank=3.9
iter=2400 ev=7.994999e+00 rank=3.8
iter=2800 ev=7.995714e+00 rank=3.9
iter=3200 ev=7.996249e+00 rank=4.0
iter=3600 ev=7.996666e+00 rank=4.0
iter=4000 ev=7.997000e+00 rank=4.0
Elapsed time is 65.257562 seconds.
```

Timing scales only linear in d. This is the *logarithmic complexity.* The accuracy of the eigenvector can be verified by computing the residue:

```
>> v=v/norm(v); ev=dot(mt*v,v); norm(mt*v-ev*v)/ev
ans =
   2.1651e-04
>>
```

The residue is not good enough due to the very bad eigenvalue solver for this problem.

8 Advanced routines

There are several advanced subroutines for *approximate basic operations.* These include:

1. `y=mvk(a,x,eps,nswp,y,rmax)` — multiplies TT-matrix `A` by a TT-vector `x` with accuracy `eps`. TT-tensor `y` can also be an initial approximation to Ax, `nswp` is the number of DMRG sweeps and `rmax` is the maximal rank of the result (sometimes needed to avoid extensive rank growth).

2. `y=funcrs(tt,fun,eps,y,nswp)` — computes function of a TT-tensor with accuracy `eps` using cross-DMRG method.

‘ Note, that these routines are only efficient for small mode sizes, i.e. for the QTT case ($n = 2$ or sometimes $m = 4$). There are several other subroutines in the subdirectory `exp`, you can try them out!

As an example, consider computation of the QTT-approximation of $\sqrt{x}$ defined on $[0, 1]$ on a very fine grid, one can use the following code:

```
%funcrs.m: Example of cross-DMRG method
%for computing functions of TT-tensors
d=70; n=2^d; h=1.0./(n-1);
x=tt_x(d,2); %QTT-representation of x with ranks 2
x=tt_tensor(x); %Make it TT-tensor
fun = @(x) sqrt(x);
tic;
tt=funcrs(x,fun,1e-12,x,8);
toc;
```

The computation results are

```
>> fcrs
sweep=1, er=9.34e-02 er_nrm=1.00e+00
sweep=2, er=5.89e-09 er_nrm=3.49e-08
sweep=3, er=2.75e-13 er_nrm=2.98e-08
sweep=4, er=2.75e-13 er_nrm=5.27e-09
sweep=5, er=2.75e-13 er_nrm=2.79e-08
sweep=6, er=2.75e-13 er_nrm=2.79e-08
sweep=7, er=2.75e-13 er_nrm=2.79e-08
Elapsed time is 0.451595 seconds.
```

The convergence criteria for funcrs is not a well-developed subject, thus it is safer to perform several sweeps. The solution has very good ranks, which can be verified by calling `erank` command:

```
>> erank(tt)
ans =
    6.2815
```

To check the accuracy of this approximation on 2^{70} points, one can compute integral

$$\int_0^1 \sqrt{x} dx \approx h \sum_{k=1}^{n} f(x_k).$$

This can be realized via a scalar product of the TT-tensor with tensor of all ones. It should be compared with analytical value $\frac{2}{3}$:

```
>> p=tt_ones(d,2); p=tt_tensor(p); dot(p,tt)*h-2/3
ans =
  -2.3648e-14
```