TT-Toolbox 2.2: Fast multidimensional array operations in TT-format

I.V. Oseledets

Corrections for the github version are written in annotations like this

2009-2012

1 Contributors

- Ivan Oseledets
- Sergey Dolgov
- Vladimir Kazeev
- Olga Lebedeva
- Thomas Mach

2 What is the TT-format for tensors

The goal of this note is to get a quick start into the TT-Toolbox. For more details, please refer to the HTML documentation, located at the directory doc/ and read papers on the TT-format at http://spring.inm.ras.ru/osel. These papers can be obtained from

Tensor $\mathbf{A}$ is said to be in the TT-format, if

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

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 $n_1 \ldots n_d$ 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 $i_1, \ldots, i_d, j_1, \ldots, j_d$, where multiindex $i_k, k = 1, \ldots, d$ corresponds to rows, and $j_k, k = 1, \ldots, d$ — to columns of the matrix. The matrix M is said to be in the TT-format if

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

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

3 `tt_tensor` and `tt_matrix` storage scheme

TT-Toolbox 2.2has 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.

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

4 Quick start

One has to "prepare" matrices and tensors in the format. There are several possibilities how to do it. There are several standard tensors, available in the Toolbox: `tt_ones` generates a rank-1 tensor of all ones `tt_x` generates a tensor, obtained from a vector $0 : n^d$.

```
 >> d=5; rhs=tt_ones(2,d) %Generate a d-dimensional
                         %tensor of all ones
rhs is 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
```

As for matrices, one generate the identity matrix very easily by calling.

```
>> e=tt_eye(2,d); %Generate 2^d x 2^d identity matrix
                  %in TT-format
e 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 TT-Toolbox contains efficient low-parametric representations of simplest discretizations of Laplace operator. For example, `tt_qlaplace_dd` generates a discretization of a f-dimensional Laplace operator, discretized on a 2^d grid in each direction.

```
>> lp=tt_qlaplacex_dd([d,d,d])
lp is a 15-dimensional TT-matrix, ranks and mode sizes:
r(1)=1   n(1)=2 m(1) = 2
r(2)=5   n(2)=2 m(2) = 2
r(3)=5   n(3)=2 m(3) = 2
r(4)=5   n(4)=2 m(4) = 2
r(5)=5   n(5)=2 m(5) = 2
r(6)=1   n(6)=2 m(6) = 2
r(7)=5   n(7)=2 m(7) = 2
r(8)=5   n(8)=2 m(8) = 2
r(9)=5   n(9)=2 m(9) = 2
r(10)=5      n(10)=2 m(10) = 2
r(11)=1      n(11)=2 m(11) = 2
r(12)=5      n(12)=2 m(12) = 2
r(13)=5      n(13)=2 m(13) = 2
r(14)=5      n(14)=2 m(14) = 2
```

```
r(15)=5       n(15)=2 m(15) = 2
r(16)=1
```

Now we are able to solve a nontrivial problem. Let us try to approximately solve the three-dimensional Laplace equation with right-hand side equal to one everywhere. It can be done easily.

```
>> d=7; lp=tt_qlaplace_dd([d,d,d]);
>> rhs=tt_ones(2,3*d); sol=dmrg_solve2(lp,rhs,1e-6)
```

`dmrg_solve2` is the name of the solver, and $1e-6$ is the required accuracy. The output on my machine is

```
=dmrg_solve2= Sweep 1, dx_max: 2.246e+00, res_max: 4.614e+05, erank: 5.30409
=dmrg_solve2= Sweep 2, dx_max: 3.254e-01, res_max: 8.606e-02, erank: 5.40987
=dmrg_solve2= Sweep 3, dx_max: 9.813e-15, res_max: 2.693e-14, erank: 4.50185
=dmrg_solve2= Sweep 4, dx_max: 0.000e+00, res_max: 1.107e-14, erank: 1.93218
=dmrg_solve2= Sweep 1, dx_max: 2.047e+02, res_max: 1.224e+05, erank: 5.51189
=dmrg_solve2= Sweep 2, dx_max: 3.881e-01, res_max: 4.182e+00, erank: 11.2821
=dmrg_solve2= Sweep 3, dx_max: 6.940e-02, res_max: 3.962e-01, erank: 18.5293
=dmrg_solve2= Sweep 4, dx_max: 6.444e-05, res_max: 5.604e-03, erank: 24.5541
=dmrg_solve2= Sweep 5, dx_max: 5.519e-08, res_max: 7.603e-06, erank: 25.7358
=dmrg_solve2= Sweep 6, dx_max: 1.972e-10, res_max: 3.431e-08, erank: 25.8134
=dmrg_solve2= Sweep 7, dx_max: 0.000e+00, res_max: 6.507e-08, erank: 21.6135
sol is a 21-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)=8   n(4)=2
r(5)=12      n(5)=2
r(6)=16      n(6)=2
r(7)=20      n(7)=2
r(8)=12      n(8)=2
r(9)=23      n(9)=2
r(10)=38     n(10)=2
r(11)=46     n(11)=2
r(12)=45     n(12)=2
r(13)=37     n(13)=2
r(14)=23     n(14)=2
r(15)=12     n(15)=2
r(16)=23     n(16)=2
r(17)=19     n(17)=2
r(18)=15     n(18)=2
r(19)=8      n(19)=2
r(20)=4      n(20)=2
r(21)=2      n(21)=2
r(22)=1
```

Consider using also newer
amen_solve2
amen_block_solve
for the linear system solution
(see help statements in these files)

The accuracy then can be checked as

```
>> norm(lp*sol-rhs)/norm(rhs)

ans =

   4.5194e-07
```

If not the residue is required, but L2-norm approximation to the solution, then the very important *rounding* procedure should be used.

```
>> sol_appr = round(sol,1e-6)
sol_appr is a 21-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)=8    n(5)=2
r(6)=10       n(6)=2
r(7)=12       n(7)=2
r(8)=7    n(8)=2
r(9)=13       n(9)=2
r(10)=19      n(10)=2
r(11)=21      n(11)=2
r(12)=23      n(12)=2
r(13)=21      n(13)=2
r(14)=13      n(14)=2
r(15)=7       n(15)=2
r(16)=13      n(16)=2
r(17)=13      n(17)=2
r(18)=11      n(18)=2
r(19)=8       n(19)=2
r(20)=4       n(20)=2
r(21)=2       n(21)=2
r(22)=1
```

Note, that the ranks are now much smaller. The relative error of the approximation is guaranteed:

```
>> norm(sol-sol_appr)/norm(sol)

ans =

   5.6937e-07
```

But the residual is larger

```
>> norm(lp*sol_appr-rhs)/norm(rhs)

ans =

   5.8572e-04
```

It is very natural, because the Laplace operator is ill-conditioned.

As you can see, we used several standard MATLAB operations (matrix-by-vector product as "*", addition as "+", the Frobenius norm as a `norm` function). The only unfamiliar function is the `round` function, which helps to reduce the ranks while maintaining the accuracy. You can try to in-

crease/decrease the accuracy and see what is the effect. If a plot of the solution is required, it is convenient to convert it to a full tensor. Let us compute the slice of the solution in the middle of the "z" axis. The simplest way to do it is to reshape the quantized tensor into a 3D-tensor:

```
>> sol_3d=reshape(sol,[2^d,2^d,2^d],1e-8);
sol_3d is a 3-dimensional TT-tensor, ranks and mode sizes:
r(1)=1    n(1)=128
r(2)=12       n(2)=128
r(3)=12       n(3)=128
r(4)=1
```

Now we can extract the required slice using fancy indexing:

```
>> sl=sol_3d(:,:,2^(d-1)); sl=reshape(sl,[2^d,2^d],1e-8)
sl is a 2-dimensional TT-tensor, ranks and mode sizes:
r(1)=1    n(1)=128
r(2)=12       n(2)=128
r(3)=1
```

Finally, the conversion to the full format is done via the `full` function:

```
 >> mm=full(sl);
```

`full` has an optional argument, specifying the size of the result (i.e., to avoid additional reshapes). Now, `mm` is an ordinary ~~2D-array~~ 1D-array and can be plotted.

In the opposite, the conversion from a full tensor to a TT-tensor is done via `tt_tensor`class constructor.

5 Basic functions

TT-Toolbox supports several basic functions for matrices and vectors. The full list can be found in the HTML documentation. Here we give only the basic openes The following operations are supported:

You can reshape y on the fly by passing new size vectors, e.g. sz1, sz2:

tt = tt_tensor(y,eps,sz1)

with prod(sz1)=numel(y)

ttm = tt_matrix(y,eps,sz1,sz2)

with prod(sz1)=size(y,1)
prod(sz2)=size(y,2)

Matrices should be full (not sparse)

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. `y=full(tt)` — converts TT-tensor `tt` to a full array.
4. `y=full(ttm)` — converts TT-matrix `ttm` to a full square matrix.

5. `tt=round(tt,eps)` — approximates given TT-tensor with another TT-tensor with smaller ranks but with prescribed accuracy `eps`.

6. 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.

7. `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.

8. `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.

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

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

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

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

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

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

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

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

17. `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`.

6 More functions

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

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

a=tkron(b,c)

19. ~~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.

20. `tt=tt_random(n,d,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.

21. `tt=tt_ones(n,d)` generate tensor with mode sizes n, dimension d of all ones

22. `tt=tt_eye(n,d)` 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.

23. `tt=tt_qlaplace_dd(d)` – generate Laplacian operator with Dirichlet BC on a grid with $2^{d_1} \times 2^{d_2} \times \ldots$.

24. `tt=tt_x(n,d)` — returns QTT representation of vector $1 : n^d$

25. `reshape` of a TT-tensor

7 Advanced routines

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

1. Linear system solution `y=dmrg_solve2(a,x,eps,options)` extensive rank growth).
2. Eigenvalue solver ~~`y=dmrg_eigb2(a,x,eps,options)`~~ y = dmrg_eig(A,eps,options)
3. Cross approximation of a black-box tensor `y=dmrg_cross(d,n,fun,eps,options)`
4. Construction of the WTT filters (`filters_wtt, wtt, iwtt`)
5. Fast evaluation of pointwise function of a TT-tensor: `y=funcrs2(tt,...)`

Consider using also amen_cross and greedy2_cross

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! Also, the most brave ones can try one cross approximation algorithm based on element evaluation, contained in the subdirectory `cross`. However, right now they are not as fast as they can be, due to MATLAB indexing (well, they are fast, but not as fast as I think they should be).

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_x(2, d)*h;
fun = @(x) sqrt(x);
tic;
tt=funcrs2(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(2,d); p=tt_tensor(p); dot(p,tt)*h-2/3
ans =
  -2.3648e-14
```

8 Conclusion

More detailed information should be found in the documentation (see index.html in the doc/ directory). This software is not free from bugs, so please feel free to write and ask, what is going on.