Performance of the dot product function in radiative transfer code SORD

Sergey Korkin*[a,b], Alexei Lyapustin[b], Aliaksandr Sinyuk[c,b], and Brent Holben[b]
[a]USRA GESTAR, 7178 Columbia Gateway Drive, Columbia, MD, USA 21046; [b]NASA GSFC, 8800 Greenbelt Rd., Greenbelt, MD, USA 20771; [c]Sigma Space Corp., 4600 Forbes Blvd., Lanham-Seabrook, MD USA 20706

ABSTRACT

The successive orders of scattering radiative transfer (RT) codes frequently call the scalar (dot) product function. In this paper, we study performance of some implementations of the dot product in the RT code SORD using 50 scenarios for light scattering in the atmosphere-surface system. In the dot product function, we use the unrolled loops technique with different unrolling factor. We also considered the intrinsic Fortran functions. We show results for two machines: ifort compiler under Windows, and pgf90 under Linux. Intrinsic `DOT_PRODUCT` function showed best performance for the ifort. For the pgf90, the dot product implemented with unrolling factor 4 was the fastest.

The RT code SORD together with the interface that runs all the mentioned tests are publicly available from ftp://maiac.gsfc.nasa.gov/pub/skorkin/SORD_IP_16B (current release) or by email request from the corresponding (first) author.

Keywords: polarized radiative transfer, successive orders of scattering, open-source software, dot product

1. INTRODUCTION

Successive orders of scattering is a method of numerical simulation of light scattering in the atmosphere-surface system[1,2]. The core of the method is iterative computation of the next scattering order from the current one. Iterations start from the single scattering approximation, which has a simple analytical form. At present, the method is well developed, implemented in several RT codes[3-6], and widely used in atmospheric and terrestrial applications because of relative simplicity of coding.

In atmospheric and terrestrial applications, RT code works as part of retrieval algorithm[7,8]. Such algorithms attempt to minimize the least squared deviation between the measured and simulated signals by varying input (retrieved) parameters[9]. In case of several input parameters, one deals with multidimensional minimization. For efficient minimization, the algorithm computes derivatives of the signal over retrieved parameters (the Jacobian matrix) also using RT code (one run of the code per each parameter to be retrieved). Thus, the retrieval algorithm invokes RT code many times. Reprocessing of huge amount of existing data or real-time processing requires fast minimization. This in its turn impose severe requirement on the RT code run time.

The known monochromatic vector radiative transfer equation[10] (RTE) provides theoretical basis for the method of successive orders. Boundary conditions imposed on the RTE define irradiance on top of atmosphere and reflectance from the bottom. The RTE has one derivative over dimensionless optical thickness, and two integrals, one over cosine of the zenith angle, and one over azimuth. Fourier expansion of the RTE solution allows for analytical integration over azimuth. For the zenith integral, Gauss quadrature is used. Integration over optical depth is performed numerically as well. Thus, there are four important input parameters to define accuracy and run time for the method. They are the number of Fourier moments, M, for integration over azimuth; the order of Gauss quadrature, N, for the zenith integration; the number of layers, L, for vertical integration over optical depth; and the number of scattering orders, S, taken into account.

In the framework of Gauss technique, the integral over cosine of the zenith angle, μ, is substituted with a summation over all N nodes. Each element of the sum is a product of the weighted scattering law, $w \cdot p(\mu_i, \mu_j)$, and the intensity, $I(\mu_j)$, at some Gauss node, j. Thus, the Gauss integration is expressed as the dot (or scalar) product

*sergey.v.korkin@nasa.gov

$$\int_{-1}^{1} p(\mu_i,\mu) I(\mu) d\mu \approx \sum_{j=1}^{N} w_j p(\mu_i,\mu_j) I(\mu_j) = \mathbf{P}\cdot\mathbf{I} = dot\_product(\mathbf{P},\mathbf{I}) \cdot \quad (1)$$

In Eq.(1), $\mathbf{P} = \left[w_1 p(\mu_i,\mu_1) \quad w_2 p(\mu_i,\mu_2) \quad ... \quad w_N p(\mu_i,\mu_N) \right]$ and $\mathbf{I} = \left[I(\mu_1) \quad I(\mu_2) \quad ... \quad I(\mu_N) \right]$, and *i* is the direction of observation (scattering).

Now we estimate the number of calls of the dot product function in a typical case. An atmosphere loaded with moderate amount of aerosol has a total (aerosol and Rayleigh) optical thickness of about 1.0. For integration over optical depth, τ, the $d\tau = 0.01$ works well[4] meaning that such a case requires about $L = 100$ layers for vertical integration. Depending on aerosol particles, the number N takes values from ~10 (fine aerosol fraction) to ~100 (dust). We assume $N = 50$ as a typical value. Also, the number of Fourier moments, M, is about 20-50, and about $S = 10$ scattering orders are needed for computation. To account the fact that height scattering orders does not require height Fourier moments, we assume $M = 10$ for all S. Further, for polarization, the dot product is used for each component of the Stokes vector, [*I*, *Q*, *U*, *V*][1,10]. Even if one ignores *V*, he has to use the dot product 9 times more in vector case (3-by-3 matrices) compared to the scalar case (only *I*). Thus, we have L x N x M x S x 9 = 100 x 50 x 10 x 10 x 9 = 4.5E+6 (4.5 million times!). This is an estimation of the amount of dot product calls per each solar angle and band and only one invoke of the RT code. Look-up tables usually contain dozens of solar angles, thus the amount of calls reaches tens of millions per band. Finally, in case of several bands, the total number of calls of the dot product exceeds 8 orders in magnitude, not including computation of derivatives, for a not very complicated atmospheric scenario. The number of calls in case of clouds and hyperspectral remote sensing systems easily overcomes the estimation. The RT code must use the best available implementation of the dot product function. In our next chapter, we discuss different ways of implementation of the dot product, and illustrate how these approaches work in our particular set of tests.

2. METHODOLOGY

Our RT code SORD[6] (Successive ORDers) is an open source software. Because of that, in this paper we test only publicly available implementations of the dot product function: our own implementations of the dot product, the BLAS dot-product function[11], and the intrinsic Fortran functions. Optimized libraries, such as Intel® Math Kernel Library (MKL)[12], Numerical Algorithms Group (NAG)[13], and other commercial software remain beyond our discussion.

Direct implementation of the dot product of two vectors, `X1` and `X2`, with `N` elements in each is

```
S = 0.0
DO IX = 1, N
      S = S + X1(IX)*X2(IX)
END DO
```

In this paper, we focus on the known loop unrolling technique[14] (see p.188 “Basic Loop Unrolling”). The BLAS dot product function `_DOT`[11] uses this technique. Further, we will refer to the double precision of the `_DOT` function, `DDOT`. The `DDOT` uses loop unrolling to reduce loop overhead (change index and check if it still within the boundaries). A simplified (compared to the BLAS `DDOT`) example with unrolling factor 5 is given below. The preconditioning loop

```
DDOT = 0.0
M = MOD(NX, 5)
DO IX = 1, M
      DDOT5 = DDOT5 + X1(IX)*X2(IX)
END DO
```

is followed by the main loop

```
M1 = MX+1
DO IX = M1, NX, 5
      DDOT5 = DDOT5 + X1(IX)*X2(IX) + X1(IX+1)*X2(IX+1) + &
            X1(IX+2)*X2(IX+2) + X1(IX+3)*X2(IX+3) + X1(IX+4)*X2(IX+4)
END DO
```

The preconditioning loop takes extra time, which is the obvious negative side of this technique. Note that unlike in the BLAS `DDOT`, we do not check the value of `M` using `IF (M.NE.0)` condition. The `DO`-construct will check the index value and, if it is beyond the range, will do nothing.

The BLAS `DDOT` function has used the unrolling factor 5 for years[11]. In our study, we have considered dot products with the unrolling factor 5 in two forms. First is the original `DDOT` from the BLAS. It allows for arbitrary increment over elements of input vectors. Second – simplified form of the BLAS function. Since we know that in our case both increments are equal 1, we removed the piece of code for unequal increments or equal increments other than 1 from. We also omitted unnecessary `IF`-statements as mentioned above. Below we refer to our function as ***ddot 5***, while the ***blas*** corresponds to the `DDOT` function from the BLAS. We considered the following unrolling factors: 1 (no unrolled loops), 2, 4, 8, 16 (only even factors, because the Gauss order is always even; the precondition loop will take no time). To some degree this our analyses is identical to a procedure performed by the Automatically Tuned Linear Algebra Software (ATLAS) libraries[15]. We also included the intrinsic Fortran functions `DOT_PRODUCT` (***dot_prod*** in the figures below), and `SUM(A*B)`. Thus, we have 9 different implementations of the dot product function.

In our study, we used these two machines:

- **Machine 1:** Intel® i7-2720QM CPU, 2.2GHz, Windows 7 64 bit; Intel® Visual Fortran Compiler 11.0.072 integrated with Microsoft Visual Studio 2008. In Visual Studio, we enabled the "Maximize Speed" option located under `Configuration Properties\Fortran\Optimization`. Further, we refer to this machine by the compiler name, "***ifort***". The RT code SORD was developed on this Machine 1.
- **Machine 2:** Intel® Xeon E7-4890 v2 CPU, 2.8 GHz, Linux 2.6 64 bit; The Portland Group Fortran 90/95 compiler 7.1-4 with the following compiler keys: `-O3 -Mipa=fast, inline = Msmartalloc`. We refer to this machine as "***pgf 90***". The NASA Goddard Space Flight Center (GSFC) AERONET[7] team uses Machine 2 for data processing and research.

In each test, the run time was measured using the intrinsic Fortran function `CPU_TIME` on both machines. In addition to that, we have confirmed that the CPU_TIME readings correspond (with negligible difference) the result of the Linux *time* command (Machine 2).

In order to avoid influence from such factors as system updates, antivirus and remote security checks, etc. we ran the tests several times to confirm that the run time in different runs is close and deviation is negligible. In addition to that, we unplugged Machine 1 (laptop) from all networks during the tests.

In our recent SPIE paper[6, 16], we report results of validation of the RT code SORD against 50 published benchmarks. These scenarios cover a wide range of possible atmospheric conditions, such as the total optical thickness, scattering by clear atmosphere, small and large aerosol particles, and clouds, reflection from land, ocean, or black surface. In this paper, we only note that both the total number of ordinates (whole sphere) for numerical integration over the zenith), and the number of layers for vertical integration over optical thickness range from 0 (single scattering) to approximately 500 (cloud scenarios, for which the method of successive orders is usually not the best choice). Computation burden is directly proportional to first power of the number of layers, L, and second power of the number of ordinates, N. In the Figures below, we show these parameters in the form of **`NB_NH_NT`**. In this notation, **`NB`** is the number of benchmark test (for reference), **`NH`** is number of ordinates per hemisphere ($N = 2 \cdot$ **`NH`** is the total number of ordinates), and **`NT`** is the number of layers used for numerical integration over optical thickness of atmosphere, τ. For example, **`29_256_500`** means that test No.29 was computed with 256 ordinates per hemisphere (512 total), and 500 layers for vertical integration. Solar and view geometry vary in a large range as well. All these facts allow us to claim that our set of tests is representative and provides trustable estimation of run time.

3. RESULTS AND DISCUSSION

Figures 1 and 2 show run time results for the ***ifort*** and ***pgf90*** machines, respectively, and all 9 implementations of the dot product function. One bar correspond to one implementation of dot product. One color correspond to one particular test. The "hight" of a piece of a given color in a bar gives the run time for the corresponding test using corresponding implementation of the dot product. For example, in the top-right chart of Figure 1, the 13-th test (light-blue) takes about 0.6 seconds (2.6-2.0 seconds – "height" of the light-blue piece) if either dot 5 (dot product with unrolling factor 5) or the BLAS `DDOT` are used. The total height of each bar from the 0-level corresponds shows combined run time for several

tests (e.g., 5 in the example just discussed). The total time shows performance in several scenarios and because of that allows to judge efficiency of each implementation of the dot product function with more certainty compared to one particular test.

In Figure 1, we divided results into 5 groups for convenience of reading only. The first (top-left) group shows tests with run time less than 0.1 seconds in each test. No legend is shows because the chart combines results for 32 tests (32 color stripes in each bar). The second (top-right) group shows results for tests that take some fraction of a second to run. The middle row contain results for scenarios that run for a few seconds (left) and tens of seconds (right). Finally, the bottom chart show results for the tests that run hundreds of seconds each. We organized Figure 2 identically.

Figure 1. Performance of the dot product functions on the ***ifort***-machine (Windows) grouped by run time.

For the ***ifort*** machine, the intrinsic Fortran functions `DOT_PRODUCT(A, B)` and `SUM(A*B)` show best performance in all kinds of tests. Performance of the dot product without unrolling, ***dot 1***, is comparable with the intrinsic function. Performance of the `DDOT` function from BLAS and dot product with unrolling factor 5, ***dot 5***, are close. Both are slow

for all kinds of tests. Noteworthy that all other unrolling factors show performance better than the factor of 5. Thus, the BLAS dot product function is not preferable for the RT simulations on machines, similar to our ***ifort***.

On the contrary, performance of the dot product functions on the ***pgf90*** machine was different in different scenarios. For typical aerosol cases shown in the top left chart of Figure 2, performance of the BLAS `DDOT` function is the worst, while the identical ***dot 5*** is the fastest of all, except for dot product with unrolling factor 4 (***dot 4***). The BLAS showed almost the best performance in the next group with tests running about a fraction of a second each.

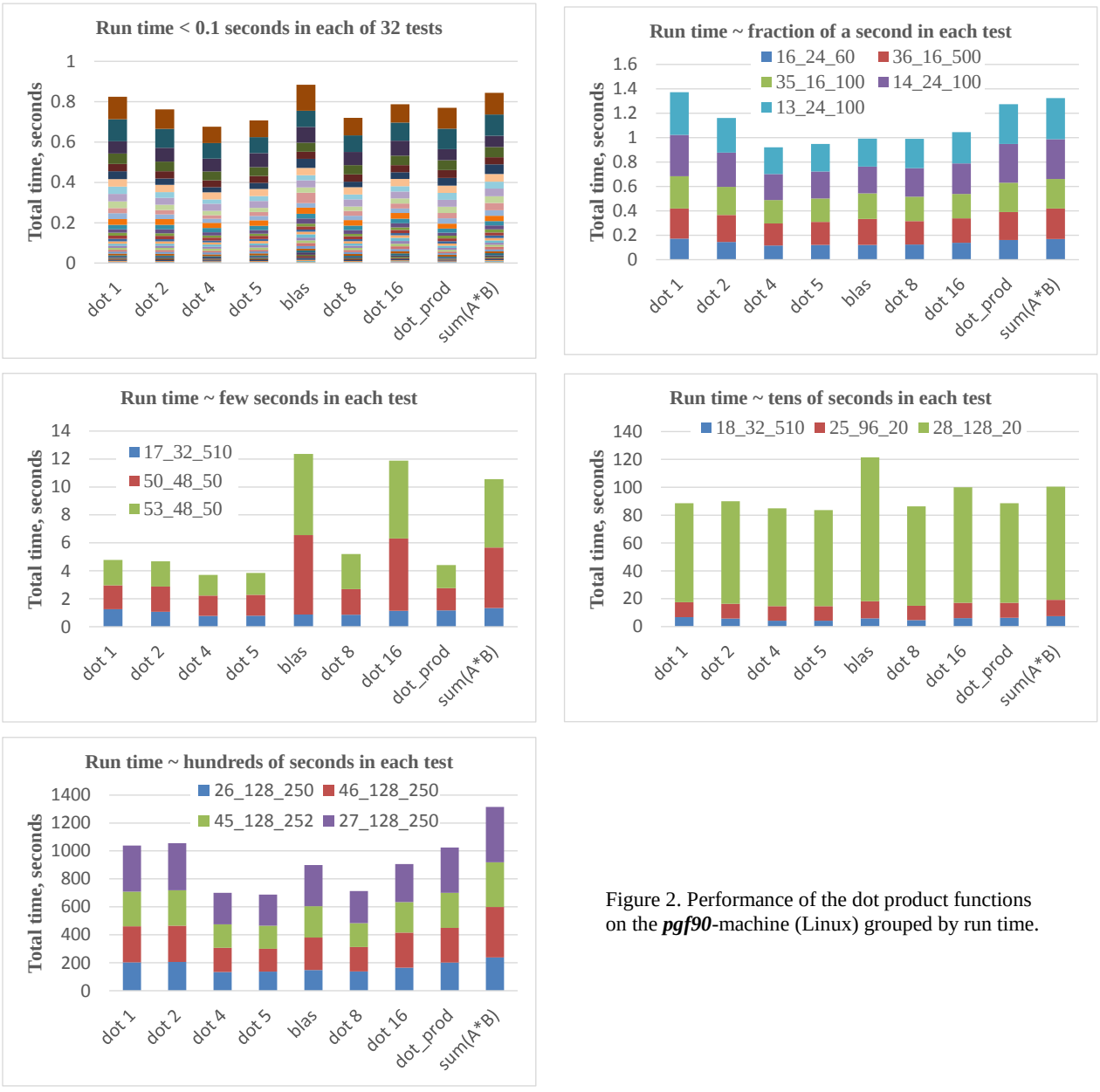

Figure 2. Performance of the dot product functions on the ***pgf90***-machine (Linux) grouped by run time.

In all other groups, the BLAS `DDOT` failed to show good performance similar to the ***ifort*** machine. The intrinsic Fortran functions, `DOT_PRODUCT(A, B)` and `SUM(A*B)`, were not the best either. Interestingly, simplified version of the BLAS `DDOT` function, ***dot 5***, was always faster compared to the original BLAS function. Dot product with unrolling factor 4 and 5, ***dot 4*** and ***dot 5*** respectively, showed the best performance in all the groups on the ***pgf90***-machine.

We also note that for the ***ifort*** machine distribution of run time over implementations of dot product is identical for all the groups: dot product from BLAS and the one with unrolling factor 5 are the slowest, while direct implementation of dot product and intrinsic Fortran functions are the fastest. The ***pgf90*** machine, on the contrary, sometimes shows random performance. However, **dot 4** and **dot 5** are always the best, while the BLAS `DDOT` is often among the worst. Based on this analysis, we recommend using the dot product with unrolling factor 4 in all cases when RT code SORD is used on the ***pgf90*** machine.

In addition to reduction of the loops overhead, the unrolling also helps increase parallelism[14] (see p.188). In the example above,

```
DDOT5 = DDOT5 + X1(IX)*X2(IX) + X1(IX+1)*X2(IX+1) + &
      X1(IX+2)*X2(IX+2) + X1(IX+3)*X2(IX+3) + X1(IX+4)*X2(IX+4),
```

there are five independent multiplications, which can be done in parallel. The compiler may or may not recognize this fact. In order to emphasize independent multiplications, the "Reductions" section[14] (see p.202) recommends the following technique (we omit the preconditioning loop for simplicity)

```
SUM1 = 0.0
…
SUM5 = 0.0
DO IX = 1, N, 5
      SUM1 = SUM1 + X1(IX)*X2(IX)
      …
      SUM5 = SUM5 + X1(IX+4)*X2(IX+4)
END DO
SUM = SUM1 + SUM2 + SUM3 + SUM4 + SUM5
```

In this example, all lines in the loop body are independent and hence can be run in parallel. However, it also has two disadvantages: 1) reference to 5 temporary variables; 2) the unrolling factor must not exceed the number of available CPUs or cores involved in parallel computations. Using our benchmarks, we will check this approach elsewhere.

4. CONCLUSION

As expected, different implementation of dot product showed different performance. Which implementation is the best depends on hardware and software. On the ***ifort*** machine, the intrinsic Fortran functions `DOT_PRODUCT(A, B)` and `SUM(A*B)` showed equally good performance. In many cases, direct implementation of dot product without unrolled loops (unrolling factor 1) showed comparable performance and can be harmlessly used. The BLAS `DDOT` showed the longest run time and hence is not recommended for the RT code SORD, if the ***ifort*** or similar machine is used.

For the ***pgf90***, on the contrary, the intrinsic Fortran functions were not the best. However, the BLAS dot product was not good either. Unrolling by a factor of 4 showed the best overall performance for the ***pgf90*** machine and we recommend this implementation for the current NASA GSFC AERONET server.

In general, we conclude that it is not only desirable, but absolutely essential to run variety of tests and only after that select the best implementation of the dot product, or at least avoid the worst one. With more than 50 benchmarks, open source policy, and (we hope) clarity of coding, the RT code SORD provides such an opportunity to a user. In the nearest future, we plan to test optimized commercial libraries, such as Intel® MKL, the automatically tuned ATLAS library, and modification of the unrolled loops as described in the end of Section 3. We plan to report the results elsewhere.

In the end of the paper we note once again that the RT code SORD and an interface that runs all the above-mentioned tests are publicly available from ftp://maiac.gsfc.nasa.gov/pub/skorkin/SORD_IP_16B (current release) or by email request from the corresponding (first) author.

ACKNOWLEDGEMENTS

This research is supported by the NASA ROSES-14 program "Remote Sensing Theory for Earth Science" managed by Dr. Lucia Tsaoussi, grant number NNX15AQ23G. Sergey Korkin thanks James Limbacher (SSAI and NASA GSFC) for useful discussions on high performance computing.

REFERENCES

[1] van de Hulst, H. C., [Multiple light scattering. Tables, formulas, and applications], Academic Press, New York (1980). Volume 1, Section 4.3, p.46.
[2] Lenoble, J. (Ed.), [Radiative Transfer in Scattering and Absorbing Atmospheres: Standard Computational Procedures], A. Deepak Publishing, Hampton VA (1985), Section 3.7, p.46.
[3] Min, Q. and Duan, M., "A successive order of scattering model for solving vector radiative transfer in the atmosphere", J. Quant. Spect. Rad. Trans. 87, 243-259 (2004).
[4] Lenoble, J., Herman, M., Deuzé, J L., Lafrance, B., Santer, R., and Tanré, D., "A successive order of scattering code for solving the vector equation of transfer in the earth's atmosphere with aerosols", J. Quant. Spect. Rad. Trans. 107, 479-507 (2007).
[5] Zhai, P.-W., Hu, Y., Trepte, C. R., and Lucker, P. L., "A vector radiative transfer model for coupled atmosphere and ocean systems based on successive order of scattering", Opt. Exp. 17(4), 2057-2079 (2009).
[6] Korkin, S., Lyapustin A., Sinyuk, A., and Holben, B., "A new code SORD for simulation of polarized light scattering in the Earth atmosphere", Proc. SPIE 9853, Polarization: Measurement, Analysis, and Remote Sensing XII, 985305, DOI: 10.1117/12.2223423. http://spie.org/Publications/Proceedings/Paper/10.1117/12.2223423.
[7] Holben, B. N., Eck, T. F., Slutsker, I., Tanré, D., Buis, J. P., Setzer, A., Vermote, E., Reagan, J. A., Kaufman, Y. J., Nakajima, T., Lavenu, F., Jankowiak, I., and Smirnov, A., "AERONET-A Federated instrument Network and Data Archive for Aerosol Characterization", Rem. Sens. Env. 66, 1–16 (1998).
[8] Dubovik, O., Lapyonok, T., Litvinov, P., Herman, M., Fuertes, D., Ducos, F., Lopatin, A., Chaikovsky, A., Torres, B., Derimian, Y., Huang, X., Aspetsberger, M., and Federspiel, C., "GRASP: a versatile algorithm for characterizing the atmosphere", SPIE: Newsroom, DOI:10.1117/2.1201408.005558, Published Online: September 19, 2014. http://spie.org/x109993.xml (16 March 2016).
[9] Dubovik, O., Herman, M., Holdak, A., Lapyonok, T., Tanré, D., Deuzé, J. L., Ducos, F., Sinyuk, A., Lopatin, A., "Statistically optimized inversion algorithm for enhanced retrieval of aerosol properties from spectral multi-angle polarimetric satellite observations", Atmos. Meas. Tech. 4, 975–1018 (2011).
[10] Chandrasekhar, S., [Radiative transfer], Oxford University Press, London (1950).
[11] http://www.netlib.org/lapack/explore-html/d5/df6/ddot_8f.html (8 August 2016).
[12] https://software.intel.com/en-us/intel-mkl (8 August 2016).
[13] https://www.nag.com/nag-fortran-library (8 August 2016).
[14] Dowd, K., [High performance computing], O'Reilly & Associates, Inc., Sebastopol CA (1993).
[15] http://math-atlas.sourceforge.net/ (8 August 2016).
[16] Korkin, S., Lyapustin A., Sinyuk, A., and Holben, B., "Accuracy of RT code SORD for realistic atmospheric profiles", Proc. SPIE Remote Sensing 2016, Edinburgh UK, Paper No.10001-10 (submitted).