REVIEW OF CPU AND GPU FADDEEVA IMPLEMENTATIONS

A. Oeftiger*, R. De Maria, L. Deniau, K. Li, E. McIntosh, L. Moneta, CERN, Geneva, Switzerland
S. Hegglin, ETH, Zürich, Switzerland
A. Aviral, BITS, Pilani, India

Abstract

The Faddeeva error function is frequently used when computing electric fields generated by two-dimensional Gaussian charge distributions. Numeric evaluation of the Faddeeva function is particularly challenging since there is no single expansion that converges rapidly over the whole complex domain. Various algorithms exist, even in the recent literature there have been new proposals. The many different implementations in computer codes offer different trade-offs between speed and accuracy. We present an extensive benchmark of selected algorithms and implementations for accuracy, speed and memory footprint, both for CPU and GPU architectures.

INTRODUCTION

The Faddeeva function $w(z)$ belongs to the family of error functions and is defined as

$$w(z) = e^{-z^2}\operatorname{erfc}(-iz) = e^{-z^2}\left(1 + \frac{2i}{\sqrt{\pi}}\int_0^z dt\, e^{t^2}\right) , \quad (1)$$

where erfc denotes the complementary error function and z a complex number. Computational algorithms to determine complex error functions often rely on the Faddeeva function, which is why it plays an important role in numeric libraries.

In electrodynamics, the Faddeeva function arises in numerical computation of the electric fields of a two-dimensional Gaussian charge distribution. In 1980, M. Bassetti and G.A. Erskine derived the corresponding expression in the context of beam-beam interaction in particle colliders [1].

A two-dimensional Gaussian charge density function

$$\rho(x,y) = \frac{Q}{2\pi\sigma_x\sigma_y}\exp\left(-\left(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2}\right)\right) \quad (2)$$

generates the electric fields [2]

$$E_u = \frac{Q}{4\pi\epsilon_0}u\int_0^\infty dt \frac{\exp\left(-\frac{x^2}{2\sigma_x^2+t} - \frac{y^2}{2\sigma_y^2+t}\right)}{(\sigma_u^2+t)\sqrt{(\sigma_x^2+t)(\sigma_y^2+t)}} \quad (3)$$

for $u = x, y$. M. Bassetti and G.A. Erskine proposed a substitution which allows to express the electric fields in terms of the Faddeeva function,

$$E_y + i\,E_x = \frac{Q}{2\epsilon_0\sqrt{2\pi(\sigma_x^2-\sigma_y^2)}}\left[w\left(\frac{x+iy}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right) - \exp\left(-\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right) w\left(\frac{x\frac{\sigma_y}{\sigma_x} + iy\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)\right] \quad (4)$$

given $\sigma_x > \sigma_y$.

In beam dynamics, numerical evaluation of Eq. (4) plays a central role for modelling collective effects such as beam-beam and beam-electron cloud interactions as well as non-self-consistent direct space charge. Therefore, one encounters implementations of algorithms to evaluate $w(z)$ in numerous simulation codes, e.g. SixTrack [3], MAD-X [4], PyORBIT [5], PyECLOUD [6, 7] and PyHEADTAIL [6, 8].

This paper aims to establish a benchmark reference as it is relevant to beam dynamics applications. We compare six implementations of numerical algorithms to determine $w(z)$ in terms of accuracy, speed and memory footprint. As various simulation codes are being developed in the Python language, we choose to interface the different implementations to Python where we evaluate the benchmarks.

IMPLEMENTATIONS

The six implementations to be compared are

1. the function `SciPy.special.wofz` provided by the Python library `SciPy` (version 0.14 and higher) which wraps the Faddeeva package by MIT [9],
2. the CERN library Fortran 90 implementation as a modification of the original code written by K. Koelbig – it is used in slightly differing variants in SixTrack, MAD-X, PyECLOUD and PyHEADTAIL,
3. the CERN library F90 implementation ported to C, as used in PyOrbit,
4. the CERN library F90 implementation ported to CUDA, as used in PyHEADTAIL,
5. the ROOT implementation [10] extracted into a C standalone file, and
6. a C port of the recent Matlab implementation by S.M. Abrarov and B.M. Quine [11, 12].

In addition we compare a variant used in SixTrack which was provided by the late Dr. G.A. Erskine. It sacrifices accuracy and memory for performance including vectorisation/pipelining and parallelisation with openMP [13].

All implementations including the benchmarking suite have been gathered on a `github.com` repository [11]. The C implementations have been interfaced via `Cython` [14] compiled with `gcc` version 4.8.4. Fortran 90 has been interfaced using `NumPy`'s `f2py` technology (also using the GNU compilers `gfortran` version 4.8.4). The CUDA kernel has been interfaced using the library `PyCUDA` [15].

* adrian.oeftiger@cern.ch, also at EPFL, Lausanne, Switzerland

ACCURACY BENCHMARK

In order to assess the accuracy of each implementation, we evaluate $w(z)$ via the arbitrary floating-point precision Python library `mpmath` [16] by employing Eq. (1). We fix a precision of `mpmath.mp.dps = 50` significant digits.

The nature of the aforementioned collective effects requires a wide input range for the Faddeeva function. Therefore we compare the accuracy of each implementation across the domain $T \in \mathbb{C}$ defined by $T = \{x + iy | 10^{-8} \leq (x, y) \leq 10^{8}\}$. Figure 1 shows for each implementation the relative deviation from the `mpmath` reference value. Here, the inputs sample T with 101 logarithmically uniformly spaced values for the real and the imaginary part, respectively. The maximal error values over T are listed in Table 1.

TIMING BENCHMARK

Timings have been performed for the same domain sampling of T like the accuracy benchmark using `time.time`. Each timing is repeated ten times, the median determines the result (this is less prone to cache misses in the timing). A given point in T is evaluated for an array of 5000 equal entries: this approach reduces the overhead from the python function call to the interfaced implementation to less than one part in a thousand. In the case of the GPU, we have used 10^6 equal array entries for the CUDA timing.

The six enumerated implementations feature consistently larger timings in the region $\{x+iy | x, y \lesssim 10\}$ than elsewhere. Table 2 lists the timings averaged over the whole domain T.

Timing for the fast SixTrack table Fortran 90 version was performed separately using `gfortran` directly instead of interfacing to Python via `f2py`. The multi-threaded version with 10'000 array entries per call gives speedups of 1.98, 2.98 and 3.72 on 2, 3 and 4 processors with `gfortran`. The same multi-threaded test with the `ifort` compiler even gives 2.09, 3.25 and 4.16, respectively.

MEMORY FOOTPRINT

The estimates of memory usage are reported in Table 3 by counting the number of tabulated constants in look-up tables used by series expansions or interpolations in the various implementations. In particular, code size and other internally used memory in math function calls such as `pow`, `exp` and similar are neglected.

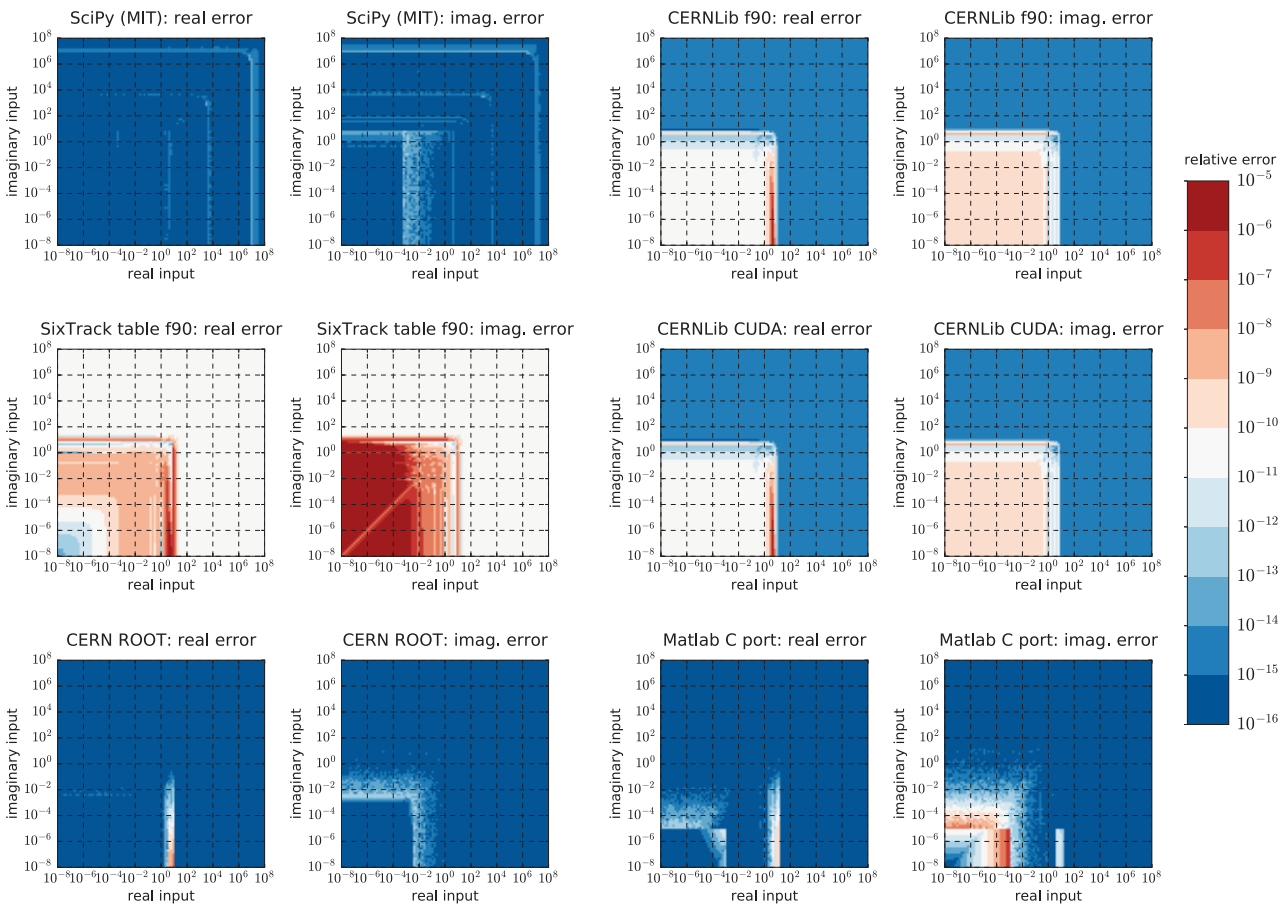

Figure 1: Relative accuracy of Faddeeva implementations w.r.t. the exact value (via Python's `mpmath`) for complex inputs ranging over 16 orders of magnitude. Large relative errors of the imaginary part in the left/bottom corner are affected by the function value approaching 0. For instance, SixTrack imaginary relative errors approach 1, while the absolute error remains below 10^{-8}. The C version of the CERNLib algorithm yields identical results as the Fortran and CUDA versions.

ISBN 978-3-95450-147-2

DISCUSSION

The `SciPy` implementation is clearly the most accurate implementation, followed by the ROOT one. The CERN library implementations in Fortran 90, CUDA and C feature equivalent error patterns confirming that they implement the same algorithm correctly. However, it is considerably less accurate than the other algorithms. The Fortran 90 timing seems to suffer from the `f2py` compilation, which is known to considerably slow down the bare Fortran performance.

The SixTrack table version trades accuracy (remaining below 10^{-8} absolute deviation) with speed. This is achieved by using a table of values computed by the CERNLib Fortran 90 version and interpolated using a third order divided-difference interpolation in the rectangle spanned from $(0, 0)$ to $(7.77, 7.46)$. For arguments outside the rectangle a two-term rational approximation is used. The speed-up is estimated to be 20-fold with respect to the CERNLib Fortran 90 version. While the loss of precision is substantial but always less than 10×10^{-8} in absolute terms it is adequate for SixTrack and for MADX-SC [17]. The Matlab version from 2014 [12] is reported to be the most accurate Faddeeva implementation [18]. Our C port features at most 1×10^{-14} difference to the Matlab values for the domain T. However, we cannot reproduce the claimed accuracy, specifically in the imaginary plane we obtain rather large relative deviations for very small input magnitudes. The respective source code and evaluation data can be found in our `github.com` repository [11].

CONCLUSION

Several algorithms and implementations of the Faddeeva functions have been benchmarked for accuracy, speed and memory footprint. The Faddeva package distributed in `SciPy` is the best in terms of accuracy and speed, however a version for GPU does not exist yet and the porting might introduce issues due to global memory latency with the large tables needed. The same reasoning applies to the SixTrack table. The ROOT version appears to be a better starting point provided a few improvements could be implemented.

Table 1: Accuracy Results for Faddeeva Implementations as Maximal Deviation From Reference Value Over the Domain T

implementation	real error	imag. error
SixTrack table	3.249×10^{-5}	2.715×10^{-1}
CERNLib F90	4.209×10^{-5}	5.165×10^{-9}
CERNLib C / CUDA	4.208×10^{-5}	5.165×10^{-9}
ROOT	1.870×10^{-7}	8.257×10^{-13}
Matlab C port	4.552×10^{-10}	2.773×10^{-7}
SciPy / MIT	2.046×10^{-14}	2.976×10^{-13}

Table 2: Timing Results for Faddeeva Implementations as Averages Over the Domain T. A call refers to evaluating a single array entry. The timing of the table based SixTrack Fortran 90 version is estimated to be about 20 faster than CERNLib F90, it has been timed in pure Fortran though as opposed to the CERNLib F90 (via `f2py`).

CUDA	0.005 µs/call
SixTrack table	0.031 µs/call
SciPy / MIT	0.132 µs/call
ROOT	0.172 µs/call
Matlab C port	0.278 µs/call
CERNLib C	0.379 µs/call
CERNLib F90	0.655 µs/call

Table 3: Memory Footprint Estimates for Faddeeva Implementations

Matlab C port	<1 KByte
CERNLib C / F90 / CUDA	<1 KByte
ROOT	2.6 KBytes
SciPy / MIT	11.5 KBytes
SixTrack table	1800 KBytes

APPENDIX

Machine Specifications

The machine specifications are outlined in the following table.

Table 4: Relevant CPU and GPU Specifications

CPU	2× Intel Xeon E5-2630 (v1)
CPU cores	2 × 6
RAM	256 GB DDR3
CPU clock rate	2.30 GHz
CPU L3 cache	15 MB
instruction set	Intel AVX
FP32 performance	0.1 TFLOPS
GPU	NVIDIA Tesla C2075
CUDA cores	448
RAM	5.3 GB DDR5
GPU clock rate	1.15 GHz
CUDA computing capability	2.0
FP32 performance	1.0 TFLOPS

The server machine runs the Linux distribution Ubuntu under version 14.04.

REFERENCES

[1] M. Bassetti and G.A. Erskine, "Space Charge Effects and Limitations in the CERN Proton Synchrotron", in *Proc. 4th Int. Particle Accelerator Conf. (IPAC'13)*, Shanghai, China, May 2013, paper WEPEA070, pp. 2669-2671.

[2] H. Wiedemann, "Statistical and Collective Effects", in *Particle Accelerator Physics*, 3rd ed. New York: Springer, 2015, p. 644.

[3] SixTrack, 6D Tracking Code, Accelerator Beam Physics Group, CERN, Switzerland, 2016, http://cern.ch/sixtrack/.

[4] MAD-X, Methodical Accelerator Design, Accelerator Beam Physics Group, CERN, Switzerland, 2016, http://cern.ch/madx/.

[5] A. Shishlo, S. Cousineau, J. Holmes, and T. Gorlov, "The Particle Accelerator Simulation Code PyORBIT", in *Proc. Int. Conf. on Computational Science (ICCS'15)*, 2015, vol. 51, p. 1272-1281.

[6] PyCOMPLETE, Python Collective Effects Library, Accelerator Beam Physics Group, CERN, Switzerland, 2016, http://github.com/PyCOMPLETE/.

[7] G. Iadarola and G. Rumolo, "PyECLOUD and Build-up Simulations at CERN", in *Proc. 5th Workshop on Electron-Cloud Effects (ECLOUD'12)*, La Biodola, Italy, June 2012, CERN Yellow Report CERN-2013-002, pp. 189-194.

[8] E. Metral *et al.*, "Beam Instabilities in Hadron Synchrotrons", in *IEEE Transactions on Nuclear Science*, vol. 63, no. 2, Apr. 2016, pp. 1001-1050.

[9] Faddeeva Package, Massachusetts Institute of Technology, Boston, USA, http://ab-initio.mit.edu/wiki/index.php/Faddeeva_Package

[10] R. Brun and F. Rademakers, "ROOT - An Object Oriented Data Analysis Framework", in *Nucl. Inst. & Meth. in Phys. Res. A* 389, 1997, pp. 81-86. See also http://root.cern.ch/.

[11] PyCOMPLETE Faddeevas, Benchmarking suite and Faddeeva implementation sources, Accelerator Beam Physics Group, CERN, 2016, http://github.com/PyCOMPLETE/faddeevas/.

[12] Matlab Central, file ID: #47801, submitted on Sept. 10, 2014.

[13] N. D'Imperio *et al.*, "Experience with OpenMP for MADX-SC", in CERN-ACC-2014-0075 and BNL C-A/AP/515, July 2014.

[14] S. Behnel *et al.*, "Cython: The Best of Both Worlds", in *Computing in Science Engineering*, vol. 13, no. 2, 2011, pp. 31-39. See also http://cython.org/.

[15] A. Klöckner *et al.*, "PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation", in *Parallel Computing*, vol. 38, no. 3, Mar. 2012, pp. 157-174. See also http://documen.tician.de/pycuda/.

[16] F. Johansson *et al.*, "mpmath: a Python library for Arbitrary-precision Floating-Point Arithmetic (version 0.19)", 2016, http://mpmath.org/.

[17] E. McIntosh, R. De Maria, and M. Giovannozzi, "Investigation of Numerical Precision Issues of Long Term Single Particle Tracking", in *Proc. of Int. Particle Accelerator Conf. (IPAC2013)*, Shanghai, China, MOPWO026, 2013.

[18] S.M. Abrarov and B.M. Quine, "Accurate Approximations for the Complex Error Function with Small Imaginary Argument", in *Journal of Mathematics Research*, vol. 7, no. 1, 2015, pp. 44-53.

ISBN 978-3-95450-147-2