AcousticLev Manual

Alexander Martinez-Marchese

August 22, 2016

Contents

1 Introduction

AcousticLev is a C++ header only library to calculate ultrasonic phase array/s settings to levitate particles using automatic differentiation (AD).

The library is an implementation of the algorithm found in [1]. All that is needed to use the library is to put the folder `acousticHologram` in the same folder with the C++ file that uses it. An example of code using the library is `particleLev.cpp`. One can use the python file `showData.py` to see the results of the calculations. The python file also writes a `.vtk` file that can be viewed in ParaView [2], that shows the magnitude of the pressure, the potential and the force field due to the phased array/s.

The library uses the following dependencies:

- Eigen [3] (for matrix operations)
- NLopt [4] (for optimization)
- MayaVi [5] (in the python file for visualization)

The library uses forward mode AD to quickly compute the pressure field at any point due to the phased array/s without any approximation errors. It uses modified FADBAD++ [6] files that include Bessel functions to model the sound transducer's pressure field.

2 How to use the library

The following steps are carried out in the `particleLEv.cpp` file, using functions in the folder `acousticHologram`.

The first step is to declare all the geometric, physical and optimization constants used for the calculation. The variable `sf` is used to scale the physical values so that they are all closer in magnitude. The suggested value for `sf` is 1000, so that all values in SI units are in mm instead of m. The variable `opt_scale` is used to scale the optimization function used by NLopt so that the solution converges faster. The Eigen matrix `w` is used to weight different directions (x,y or z) so that the shape of the sound field around the levitation point is adequate (see [1]). When using one phased array, `w` should be weighted towards the direction the array is pointing to.

The second step is to declare rectangular phased array objects. When the arrays are declared, their center is at the origin with the each of the sound transducers in the array pointing in the same direction as the z-axis. Each declared array can be translated or rotated. After moving all the needed arrays to their desired locations and there is more than one array, one adds them

together (using the overloaded + sign) to form a single fixed array. This new array now contains all the transducer locations and normal vectors which are used to calculate the transducer settings (see [1]). The class declaration is in the file `transducerArray.hpp`.

The third step is to declare the desired levitation points, and the Eigen matrix `Mj`. The `Mj` matrix stores all the sound pressure derivatives (up to third order if needed) with respect to a fixed levitation point, for each of the transducers in the final single phased array. These values are used throughout the calculations needed to optimize the array settings and to compute field values (pressure, potential or force) at a given point (see [1]). The values are exact and they are calculated using forward mode AD using an analytic model of the sound field for each transducer and FADBAD++. The functions doing this are in the file `transducerDerivatives.cpp`.

The final step is to call NLopt and the optimization function and its gradient in `optFunction.hpp` to optimize the phases of each of the transducers. This function is given as U_{aa} (eq. 10 in [1]). This function should have a 2 in front of the second term and p_n instead of p_a inside the summation. And was derived by using $|p|^2 = p \cdot p$ instead of $|p| = p \cdot p$ (eq. 11) as shown in [1]. The corrected function is given by eq. 2.

For post-processing, one can calculate the imaginary pressure, the potential and force due to the pressure field produced by all of the transducers to check that all the forces are pointing towards the desired levitation point. This data is accessed by the `showData.py` script to produce a `.vtk` file. The functions that can do this are in the file `soundField.hpp`.

Also one can compute particle paths that a particle would take from a given point to a levitation point. This includes air resistance. This is done using a 4th order Rugge-Kutta (RK4) scheme [7] with a given fixed time step. The functions doing this are found in `particlePath.hpp`.

The following sections go into more detail on how the library is designed and what equations are used. One does not need to read them in order to use the library.

3 How the AD is done

The automatic differentiation is done in forward mode three times to get the third order derivatives needed for the optimization function, using the forward mode files of FADBAD++ (`fadiff.h` together with the main header file `fadbad.h`). **The advantage of this method compared to finite differences is that the result is exact up to machine precision, and there is no error due to the step size or the frequency of the sound from the transducer being modelled**.

In theory, one should perform the differentiation in reverse mode the first time since the number of dependent variables (1) is lower than the number of independent variables (3). That is the behaviour for a large number of independent variables. However, when one programs both alternatives, using the reverse mode AD once first and then forward mode AD once for calculating forces, the run time is 35% slower than using forward mode AD twice. This is due to the more complex implementation needed for reverse mode AD, and the fact that the ratio between dependent and independent variables is low. **This illustrates the fact that even if the time complexity in big O notation is lower for one of two algorithms, one should still run both algorithms if the number of variables is low, because this type of time complexity excludes constants and lower order terms**.

By declaring the number of variables being differentiated each time forward mode AD is used (using the stack-based forward method implementation of FADBAD++), the run time was reduced a further 33% (again for calculating forces). The real and complex components of the pressure field are differentiated separately so one can easily use FADBAD++, since the transducer sound field (eq. 7 in [1]) can be separated into real and complex terms with type `double`.

4 Details on how the equations in [1] and in the code are derived

The U_{aa} function can be derived from the Gorkov potential U (eq. 3 in [1]), by noting that $|p|^2 = p \cdot p$, and by applying the product rule on eq. 11 in [1]. This yields for the derivative with respect to variable c:

$$(a \cdot b)_c = (a \cdot b_c) + (a_c \cdot b) \tag{1}$$

Substituting this equation twice in eq. 3 in [1] yields:

$$U_{aa} = 2\Big[K_1(p_a \cdot p_a + p \cdot p_{aa}) + K_2 \sum_n (p_{na} \cdot p_{na} + p_n \cdot p_{naa})\Big] \tag{2}$$

With $n = x, y, z$. Eq. 2 has an extra factor of 2 in front of the second term and p_n instead of p_a inside the summation compared to eq. 10 in [1].

To find the gradient equation (eq. 13 in [1]) used by NLopt, one again uses the product rule on eq. 12 in [1]. Note that the sound field equation for each transducer can be separated into a real and an imaginary component using Euler's formula ($e^{ix} = cos(x) + isin(x)$). The terms are in the form $kcos(\varphi^j)$ and $ksin(\varphi^j)$. For example, since $(cos(x))_x = -sin(x)$, when one computes $\Re(P_g)_{\varphi^j}$ one gets a term equal to the negative imaginary component, $-\Im(P_g^j)$, hence the negative terms in eq. 13 in [1]. No complex calculus needs to be used.

A derivation of the Gorkov potential U found in [1] is given in [8].

The force at a point can be derived by using the fact that $\vec{F} = -\nabla U$ and eq. 1. The force components with $a = i, j, k$ are the following:

$$\vec{F} \cdot a = -2\Big[K_1(p \cdot p_a) + K_2 \sum_n (p_n \cdot p_{na})\Big] \tag{3}$$

5 Library architecture

The library was designed to be accurate but also fast, hence the decision to use AD. The library was also designed to be easy to read. This was done by naming the data types in a similar way as the variables in the paper, and using the function `inline id()` found in `common.hpp`.

The Eigen matrix `Mj` for example, is based on the variable M^j in the paper, and it contains all the transducer pressure derivative terms (without the $e^{i\varphi^j}$ factor), where the rows are for each transducer and each column is for each derivative. The function `id()` is used in the code and with `Mj` in particular as follows: when `Mj` is filled with the output of the functions using FADBAD++ (in `transducerDerivatives.hpp`) the function `id(a,b,c)` corresponds to the location to store the first derivative (a) with respect to variable x,y,z or none, the second derivative (b) with respect to the same variables and so on. So for example, if one wants to get or write the value of the pressure for transducer j (M^j in the paper), one uses `Mj(j, id(0,0,0))`. If one wants M^j_{yx}, one uses `Mj(j, id(2,1,0))` and if one wants M^j_{zzz}, one uses `Mj(n, id(3,3,3))`.

When one needs to calculate only the pressure, the potential and force at a given point, one only needs `Mj` values up to the second derivative. Therefore there are two functions to fill `Mj`, one for the optimization function used by NLopt that fills `Mj` up to third derivatives using forward mode AD three times (`getThirdDerivsMjs()`), and one for calculating the pressure, potential and force at a given point that fills `Mj` up to second derivatives using forward mode AD two times (`getSecondDerivsMjs()`). Both functions are found in `transducerDerivatives.hpp`.

For calculating pressure, potential and force, there is a common initial loop that sums all the pressures from each transducer, however one only needs to sum up to second order derivatives for calculating the force (see force equation), up to first order derivatives for the potential (see eq. 3 in [1]) and only the pressure terms for the pressure. This loop can be taken out of the three corresponding functions (it is the function `inline pressureDerivatives()` in `transducerDerivatives.hpp`) and when summing over pressure terms, their first, or their second derivatives, the derivative summation will be over `id() = 0`, `id() = 0 to 3` and `id() = 0 to 15`, since the function `id(a,b,c)` returns `a + b * MAX_ID + c * MAX_ID * MAX_ID` with `MAX_ID = 4`. This can be done because the right derivatives are stored contiguously in the columns of the `Mj` matrix.

References

[1] Asier Marzo, Sue Ann Seah, Bruce W Drinkwater, Deepak Ranjan Sahoo, Benjamin Long, and Sriram Subramanian. Holographic acoustic elements for manipulation of levitated objects. *Nature communications*, 6, 2015.

[2] James Ahrens, Berk Geveci, Charles Law, CD Hansen, and CR Johnson. Paraview: An end-user tool for large-data visualization, 2005.

[3] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[4] Steven G. Johnson. The nlopt nonlinear-optimization package. http://ab-initio.mit.edu/nlopt.

[5] P. Ramachandran and G. Varoquaux. Mayavi: 3D Visualization of Scientific Data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[6] Ole Stauning. Fadbad++: Flexible automatic differentiation using templates and operator overloading in c++. http://http://www.fadbad.com.

[7] J.Douglas Faires and Richard Burden. *Numerical Methods*. Brooks/Cole, 2nd edition, 1998.

[8] Henrik Bruus. Acoustofluidics 7: The acoustic radiation force on small particles. *Lab on a Chip*, 12(6):1014–1021, 2012.