Accelerated approximation of gyroaveraging operators

Oren Bassik

Abstract. In solving the gyrokinetic Vlasov-Poisson system, it is necessary to apply the gyroaverage operator. We benchmark various schemes to numerically approximate this operator: using both equispaced nodes, chebyshev nodes, and local spline approximation vs global spectral methods. In particular we quantify the impact of shared-memory parallelism and the use of GPU accelators to speed up calculations, as well as the trade-off of accuracy vs computational cost.

In solving the gyrokinetic Vlasov-Poisson system, at every time step one must calculate a gyroaverage. Given a function $f(x, y, \rho) : \mathbb{R}^3 \to \mathbb{R}$ we define the *gyroaverage operator:*

$$G(f)(x_c, y_c, \rho) = \int_0^{2\pi} f(x_c + \rho\sin(\gamma), y_c - \rho\cos(\gamma), \rho)\mathrm{d}\gamma$$

1. APPROXIMATION METHODS We outline several calculation methods. First note that if we have access to the function f itself, we can use any of various quadrature methods to calculate $G(f)$ naively from its definition, at any point we wish to do so. Indeed this is our benchmark for accuracy. In our case we use an adaptive trapezoid rule from the Boost C++ library, as our integrand is periodic.

In a numerical solver, we must represent our function via some discrete data and an interpolation scheme. We have chosen to fully parallelize the dependence of f on ρ; we pick a discrete set of ρ and on each value we view f as a bivariate function. We then sample $f(\cdot, \cdot, \rho)$ at either

1. equispaced nodes
2. tensor-product chebyshev nodes, i.e. $\left(\cos\left(\frac{j\pi}{N-1}\right), \cos\left(\frac{k\pi}{N-1}\right)\right)\ 1 \leq j, k \leq N$

Given equispaced nodes, we have implemented a variety of interpolation schemes. These include

1. Bilinear interpolation: the function is represented on each rectangle $(x_i, x_{i+1}) \times (y_j, y_{j+1})$ as a function of the form $\sum_{i,j=0}^{1} a_{ij}x^i y^j$.
2. Bicubic interpolation: the function is represented on each rectangle $(x_i, x_{i+1}) \times (y_j, y_{j+1})$ as a function of the form $\sum_{i,j=0}^{3} a_{ij}x^i y^j$ (so 16 coefficients). We approximate derivatives using 4th order finite differences where possible, and the resulting representation is C^1 everywhere.
3. Trigonometric interpolation: the function is represented globally by an expression of the type

$$\sum_{p,q=0}^{N} a_{pq}\cos\left(\frac{\pi p(2x+1)}{2N}\right)\cos\left(\frac{\pi q(2y+1)}{2n}\right)$$

Given Chebyshev nodes, we use tensor product Chebyshev interpolation. That is to say, our function is approximated globally by an expression of the form

$$\sum_{p,q=0}^{N} a_{pq} T_p(x) T_q(y).$$

Where $T_n(x)$ is the Chebyshev polynomial of the first kind.
Note that for both Chebyshev and Fourier interpolation, the calculation of the a_{pq} can be accomplished with an FFT, as can the inverse transformation (from coefficients to values at nodes). (We use the real-to-real DCT code in the FFTW library).

2. INTEGRATION METHODS Each method of approximation lends itself to different gyroaveraging schemes:

Bilinear interpolation The simplest scheme we have implemented is to feed our bilinear interpolated function into an adaptive trapezoid rule integrator; this is mainly to test the accuracy of the following scheme. The below gives orders of magnitude speed-up for this fairly simple method (which is however not very accurate).

The scheme is as follows; we suppress the third parameter of f for clarity:

1. Seperate the integral

$$G(f)(x_c, y_c) = \int_0^{2\pi} f(x_c + \rho\sin(\gamma), y_c - \rho\cos(\gamma))\mathrm{d}\gamma$$

   into arcs wherever the circle of radius ρ passes through a grid line (the border of a rectangle on across which the bilinear interpolation parameters change.).
2. On each of these arcs, f is approximated as bilinear, i.e.

$$G(f)(x_c, y_c) = \int_{\gamma_k}^{\gamma_{k+1}} \sum_{i,j=0}^{1} a_{ij}(x_c + \rho\sin(\gamma))^i (y_c - \rho\cos(\gamma))^j \mathrm{d}\gamma$$

3. This is analytically integrated, and the contribution of this arc to $G(f)(x_c, y_c)$ is linear in the a_{ij} parameters.
4. The dependence of the a_{ij} parameters on the values of f sampled at the 4 nearest node points is indeed linear as well. This is composed with the above linear operation so that $G(f)(x_c, y_c)$ is a linear combination of values of f sampled near the circle of radius ρ around (x_c, y_c) and nowhere else.
5. The above is stored in a sparse matrix,and $G(f)$ is thus represented as a sparse matrix multplication (applied straight to the sampled values of f at equispaced nodes).

The sparse matrix above is large but can be precomputed for any given set of nodes and values for ρ. This method is also easily accelerated by any speedup for sparse matrix multiplication; indeed we use both OpenMP for multi-core CPU acceleration and the ViennaCL library for GPU-accelerated sparse matrix multiplication in our benchmarks.

Bicubic interpolation This is similiar to the above, with a couple of extra steps:

1. First, we approximate at each node point the f_x, f_y, and f_{xy}. derivatives, using 5-point stencils and 16-point stencils for fourth-order accuracy.

2. Next, we solve on each rectangle for 16 coefficients a_{ij} to match f, f_x, f_y, and f_{xy} at each corner.
3. As above, we seperate each gyroaverage calculation into piecewise bicubic arcs, and analytically integrate

$$G(f)(x_c, y_c) = \int_{\gamma_k}^{\gamma_{k+1}} \sum_{i,j=0}^{3} a_{ij}(x_c + \rho\sin(\gamma))^i (y_c - \rho\cos(\gamma))^j \mathrm{d}\gamma$$

4. The full matrix of gyroaverage values $G(f)(x_i, y_j)$ is again a sparse matrix multiplication applied to the collection of bicubic coefficients.

In terms of the computational cost of calculating a gyroaverage, for any given set of nodes and radii we can precompute and store the above sparse matrix. The calculation of finite differences and bicubic coefficients needs to be redone at every step, and this incurs a storage penalty of 16x the size of storing just the values of f. The sparse matrix is also quite a bit larger than in the bilinear case. As above we are able to accelerate the linear algebra using either CPU or GPU parallelism.

Fourier interpolation This is a global method, and we expect spectral accuracy. Indeed, if f were periodic, this method would unquestionably beat all others. This is for two reasons. One, we can rely on the FFT (DCT and IDCT in our case) to very quickly move between Fourier coefficients and sampled function values. Two, gyroaveraging is a form of convolution, and so takes a very simple form in frequency space. Indeed, if we gyroaverage a generic bivariate cosine transform basis function, we find

$$\int_0^{2\pi} \left(\cos(a + b\sin(\gamma))\right)\left(\cos(c + d\cos(\gamma))\right) \mathrm{d}\gamma \; = 2\pi\cos(a)\cos(c) J_0(\sqrt{b^2 + d^2})$$

where J_0 denotes the Bessel function of the first kind. This is to say that we can calculate a gyroaverage by

1. Take a forward DCT of f sampled on equispaced nodes
2. do a coefficient-wise multiply by a matrix of precomputed values $J_0(\cdot)$ above
3. perform an inverse DCT

All of the above calculations are very fast; we use the mature FFTW library which provides all of the relevant transforms and takes full advantage of multi-core CPUs. PCIe memory transfer limits the benefits of GPUs for "small" FFTs.

For non-periodic functions, this method will fail at any point where the gyroaveraging circle will cross outside the original periodicity bounds. Indeed, even if we have a function which goes to 0 everywhere on the boundary but fails to be smooth there (as a periodic function), we expect Fourier interpolation to fail (to converge). However, this is easily overcome by expanding our domain and padding with zero values so that at our max ρ we still do not cross the new boundary. This indeed works, at least for functions smooth on our original domain and which approach 0 on the boundary; however there is an obvious cost in both memory usage and computational cost.

Chebyshev interpolation This is a global method, and with Chebyshev interpolation we are not limited to periodic functions nor do we need any padding. Very simply, we

precompute for each 2d basis function $T_p(x)T_q(y)$ the full gyroaverage on the whole target grid:

$$G(f_{pq})(x_c, y_c) = \int_0^{2\pi} T_p(x_c + \rho\sin(\gamma))T_q(y_c + \rho\cos(\gamma))\mathrm{d}\gamma$$

and we store this as a row in a dense matrix. This matrix has N^4 entries for each value of ρ. The precomputation is fairly expensive, but since our basis functions are smooth we can actually re-use the padded FFT method above to calculate it efficiently and scalably. This method is quite general and we can choose our target matrix independent of initially sampling chebyshev nodes: indeed we target equispaced nodes so that we can fairly compare accuracy vs all the other methods. Also, this method is amenable to substitution of Chebyshev polynomials for some other set of basis functions, in particular other classical orthogonal polynomials. After precomputation, each gyroaverage is just

1. a DCT to calculate coefficients for a 2D Chebyshev approximant
2. a dense matrix-vector multiplication

3. DATA For our initial set of benchmarks, we are using a smooth tractable function which happens to have an analytic gyroaverage. To be specific, we are using

$$f(x, y, \rho) = Ne^{-A(x^2+y^2)}e^{-B\rho^2}$$

with $N = 50$, $A = 24$, $B = 1.1$, and ρ taking 20 values in a range from 0 to 0.8625, on the domain $[-1, 1]^2$. For this function, we have calculated relative error, precomputation time, and run time for each of our methods, using both fp32 and fp64 data types. These were run on a machine ("cuda4" at NYU)with 48 2.3ghz CPU cores, and two Titan X GPUs (though our code currently only uses 1!) with 12gb GPU memory. The data is currently in an attached Excel spreadsheet.