A comparison of schemes for the numerical approximation of the gyroaveraging operator

by

Oren Bassik

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Mathematics

New York University

May 2021

Professor Antoine Cerfon

Abstract

We implement and benchmark a variety of schemes for calculating gyroaverages in the 2D, compactly supported non-periodic setting. Schemes implemented include bilinear spline collocation, bicubic spline collocation, padded (bivariate) FFT interpolation, and bivariate tensor Chebyshev interpolation. In particular we quantify the impact of shared-memory parallelism and the use of GPU accelerators to speed up calculations, as well as the trade-off of accuracy vs computational cost, for both smooth functions and those with singularities.

Contents

Abstract iii
List of Figures v
List of Tables vi

1 Introduction **1**
1.1 Background 1
1.2 Outline 2

2 Four discretizations **3**
2.1 Defining the problem 3
2.2 Bilinear interpolation 4
2.3 Bicubic Interpolation 6
2.4 Fourier transform based gyroaveraging 7
2.5 Chebyshev Spectral gyroaveraging 8

3 Numerical Results **13**
3.1 Gallery of test functions 13
3.2 Testing environment and setup 14
3.3 Convergence Study 15

A One more comment **17**

Bibliography **18**

List of Figures

3.1 Gallery of test functions 16

List of Tables

Chapter 1

Introduction

1.1 Background

Given a function f taking real or complex values on some domain in $\mathbb{R}^2$, and an additional parameter specifying a radius ρ, define the gyroaveraging operator as

$$\mathcal{G}(f)(x, y; \rho) = \frac{1}{2\pi} \int_0^{2\pi} f(x + \rho \sin(\gamma), y + \rho \cos(\gamma)) \, \mathrm{d}\gamma$$

This operator is a special case of what is called in various texts a spherical mean operator, peripheral mean operator, or spherical Radon transform. The spherical mean transform is defined for any dimension $n \geq 1$ as

$$\mathcal{J}(f)(\mathbf{x}; \rho) = \frac{1}{\omega_n} \int_{|\mathbf{y}|=1} f(x + \rho \mathbf{y}) \, \mathrm{d}S(\mathbf{y}) = \fint_{\partial B(\mathbf{x}, \rho)} f(\mathbf{y}) \, \mathrm{d}S(\mathbf{y})$$

This operator is introduced in many PDE textbooks, among other applications, to solve the wave equation in n dimensions. Our interest in the numerical calculation of the 2D case, as well as the name "gyroaverage", comes from an application in plasma physics: solving the "Vlasov-Poisson" equations to simulate a beam of charged particles in a cyclotron. While a detailed derivation is out of the scope of this thesis, and is described in detail in REFERENCE, a very brief description of the scheme is given (in REFERENCE) as:

> Briefly, our method of solution of the VPE consists of the following steps: (1) calculate $\mathcal{G}$f via various numerical algorithms, (2) compute χ with a free-boundary Poisson solver, and (3) time-advance f in the Vlasov equation.

The Poisson solver and time-stepping parts of the solver are well-studied and have many well-known schemews(as well as many freely available implementations);

meanwhile the gyroaveraging operator is being applied inside the time-stepping loop and suffers from (as is apparent from the definition) nonlocal "everywhere-to-everywhere" interactions. Thus it is reasonable to expect gains from speeding up the gyroaveraging as the lowest hanging fruit in improving this particular solver.

Another application (described in REFERENCE) is in the field of thermoacoustic tomography. Very briefly, an electromagnetic radiation source is applied to a body of biologic tissue, which generates heat pulses in the tissue; this in turn causes small, temporary expansions. These expansions radiate outwards as sound waves, and are detected at the boundary of the tissue by an array of transducers. The reconstruction of an image from the transducer signals is, mathematically, a problem of inverting the spherical mean transform. One angle of attack for the inverse problem is via iterative methods, which rely on applying the forward transform to repeatedly refine potential inverses; for these methods efficient and scalable algorithms for the forward transform translate into gains for the inverse.

1.2 Outline

In the remainder of this paper, we will focus on the forward gyroaveraging operator alone, in a 2D setting. We assume that we are in the context of a numerical solver, and that we have access to function values alone (and not an analytic representation), and that our domain of interest is bounded (i.e. we are gyroaveraging functions of compact support.)

Chapter TODO will recall some basic facts needed for the rest of the thesis, including: Fourier analysis, Chebyshev approximation theory and spline approximation, as well as basic properties of the spherical mean transform.

Chapter TODO will describe four different discretizations which we implemented, and the mathematical operation of applying the gyroaverage operator onto each. In the case of smooth functions, we present some error estimates.

Chapter TODO adds implementation details, including details on memory usage, asymptotic runtime complexity, CPU multicore parallelism, GPU parallelism, and the extent and nature of precomputation necessary.

Chapter TODO presents numerical results; we start by defining six functions with different salient features and in each case analyze converge and calculation speed, for each implemented algorithm.

Chapter TODO is a brief summary of the API implemented; we have tried to make a user-friendly self-contained C++ header-only library with dependencies on publicly available free software. In addition via RAII and templates we allow the user fine control over memory usage with no overhead in interface complexity, and the ability to pass any sort of callable object.

Chapter TODO summarizes the numerical results and outlines areas for further improvement and investigation.

Chapter 2

Four discretizations

In this chapter, …

2.1 Defining the problem

We proceed by defining precisely the problem we are solving, and the four discretizations we have implemented.

The input data are

1. A function $f(x, y, \rho) : [-1, 1]^2 \times [0, 2] \to \mathbb{R}$
2. A set of $\rho_k \in [0, \infty)$ which represent gyroradii;
3. We choose a set of N^2 nodes (x_i, y_j) with $x_i, y_j \in [-1, 1], 1 \leq i, j \leq N$.
4. We are then given the values of f at each of the nodes x_i, y_j, ρ_k.

We assume the function is identically 0 outside of the box $[-1, 1]$ but that it may have nonzero derivatives on the boundary. The two choices of input nodes x_i and y_i we investigate are

1. equispaced nodes
2. Chebyshev nodes

We then want to populate, for each ρ_k, a matrix $\mathcal{G}(f)$ with approximations to

$$(\mathcal{G}f)_{\rho_k}(x'_i, y'_j) = \frac{1}{2\pi} \int_0^{2\pi} f(x'_i + \rho_k \sin(\gamma), y'_j + \rho_k \cos(\gamma)) \, \mathrm{d}\gamma$$

Where the primed x'_i, y'_j are equispaced nodes in every case (even when we sample at Chebyshev nodes to begin with.) The choice of identical output space allows for

uniform comparison of convergence between schemes, and it turns out it does not penalize the Chebyshev scheme. For the remainder of this chapter we will suppress the variable ρ_k and treat f as a bivariate function.

Given that the problem is basically a quadrature problem, we proceed as many quadrature schemes do: we interpolate between the given samples and try to integrate the interpolant exactly. We have implemented four different interpolation schemes, which we detail one by one below.

2.2 Bilinear interpolation

Our simplest interpolation is bilinear. Given equispaced nodes, on each rectangle $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$ f is represented by the unique polynomial of the form $A + Bx + Cy + Dxy$ which matches the sampled value at each corner. This is a continuous but not necessarily differentiable approximation of f. One can express the map from sampled values to parameters (A, B, C, D) as a matrix, or compute interpolating values (for points between samples) on the fly.

Given bilinear interpolation, we have two algorithms which produce (roughly) the same mathematical quadrature but differ greatly in speed. The first algorithm is to set up the bilinearly interpolated values of f as a function $\tilde{f}$ and feed this function into a black-box adaptive integration algorithm (we use an adaptive Gauss-Kronrod quadrature from the C++ Boost library; further details in chapter TODO).

The second algorithm is one of the main innovations of this paper (and was suggested, more or less in full, by Prof. Cerfon.) It has a modestly expensive precomputation step after which the gyroaveraging itself is represented by a sparse matrix which can be applied directly to the sample values matrix. We describe first the precomputation with reference to figure TODO:

1. Fix a particular x_c, y_c and ρ of interest. We want to approximate

$$\mathcal{G}f(x_c, y_c) = \frac{1}{2\pi} \int_0^{2\pi} f(x_c + \rho \sin(\gamma), y_c + \rho \cos(\gamma)) \, \mathrm{d}\gamma$$

2. Enumerate the points of intersection between the circle $(x_c + \rho \sin \gamma, y_c - \rho \cos \gamma)$ and the grid lines $x = x_i$ and $y = y_j$. This will produce a vector of angles $(\gamma_0 = 0, \gamma_1, \ldots, \gamma_p = 2\pi)$ such that

   (a) Each arc corresponding to $[\gamma_i, \gamma_{i+1})$ lies entirely between grid lines, and so is represented by a single bilinear polynomial in that region

(b) The disjoint union of the arcs is the entire circle

Thus

$$\mathcal{G}f(x_c, y_c) = \frac{1}{2\pi} \sum_p \int_{\gamma_p}^{\gamma_{p+1}} f(x_c + \rho\sin(\gamma), y_c + \rho\cos(\gamma))\,\mathrm{d}\gamma \approx$$

$$\sum_p \int_{\gamma_p}^{\gamma_{p+1}} \tilde{f}_p(x_c + \rho\sin(\gamma), y_c + \rho\cos(\gamma))\,\mathrm{d}\gamma$$

where

$$\tilde{f}_p(x, y) = A_p + B_p x + C_p y + D_p xy$$

We can analytically integrate, indeed

$$\int \tilde{f}_p(x_c + \rho\sin(\gamma), y_c + \rho\cos(\gamma))\,\mathrm{d}\gamma =$$

$$A_p\gamma + B_p(\gamma x_c - \rho\cos\gamma) + C_p(\gamma y_c - \rho\sin\gamma) + \frac{D_p}{2}(\rho^2\cos^2\gamma + 2x_c y_c\gamma - 2\rho y_c\cos\gamma - 2\rho x_c\sin\gamma)$$

Thus the contribution of each arc to the total gyroaverage is seen to be linear in the coefficients A_p, B_p, C_p, D_p.

3. These coefficients are themselves a linear combination of the sample values at the corners of the grid rectangle containing each arc. For each arc, compose two matrices to find the contribution of that arc to $\mathcal{G}f(x_c, y_c)$ as a linear function of sample values.

4. Form a sparse $N^2 \times N^2$ matrix which operates on sample values and outputs the matrix $\mathcal{G}f(x_c, y_c)$ flattened as a vector. More precisely, each row represents one gyroaverage circle, and there will be 4 nonzero entries representing each arc in each row.

The number of nonzero entries in the resulting sparse matrix is $O(N^3)$: we have N^2 circles, and the number of arcs per circle is bounded by $4N$.

This sparse matrix can be initialized and cached, and depends only on the domain and ρ. After precomputation the actual gyroaveraging algorithm is simple: we multiply the sample value matrix by the precomputed sparse matrix. This has runtime complexity of $O(N^3)$ and we expect it to scale very well, and to benefit from extensive prior work on sparse linear algebra. We also expect to be constrained by the accuracy of a bilinear interpolation.

2.3 Bicubic Interpolation

Here, using equispaced nodes, we approximate f by the form

$$\tilde{f}(x,y) = \sum_{i,j=0}^{3} a_{ij} x^i y^j$$

We need choose 16 coefficients for each grid space $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$. The most natural choice is to match values and first derivatives at each corner, as well the mixed derivative f_{xy}.
We don't have analytic derivatives available, so our code uses 4th order finite difference approximations (a 5-point stencil for f_x and f_y, and 16-point stencil for f_{xy}.) To reduce memory usage, we represent the stencil in code rather than represent the finite difference operator as a sparse matrix. After calculating finite difference approximations, we use the below formula for bicubic interpolation. There are a few standard ways to do this; the below was slightly faster than a 16x16 sparse matrix multiplication on one test machine (but this is almost certainly machine/language/compiler dependent.)

2.3.1 Interpolation formula

Suppose we are interpolating on the square between (x_0, y_0) and (x_1, y_1). Let superscripts $f^{00}, f^{10}, f^{01}, f^{11}$ denote the sampled values of f at $(x_0, y_0), (x_1, y_0), (x_0, y_1), (x_1, y_1)$ respectively. Similarly let f_x, f_y,and f_{xy} denote the finite difference approximations. Then if define the matrix $\{a_{ij}\}$ as

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ 0 & 1 & 2x_0 & 3x_0^2 \\ 0 & 1 & 2x_1 & 3x_1^2 \end{pmatrix}^{-1} \begin{pmatrix} f^{00} & f^{01} & f_y^{00} & f_y^{01} \\ f^{10} & f^{11} & f_y^{10} & f_y^{11} \\ f_x^{00} & f_x^{01} & f_{xy}^{00} & f_{xy}^{01} \\ f_x^{10} & f_x^{11} & f_{xy}^{10} & f_{xy}^{11} \end{pmatrix} \begin{pmatrix} 1 & y_0 & y_0^2 & y_0^3 \\ 1 & y_1 & y_1^2 & y_1^3 \\ 0 & 1 & 2y_0 & 3y_0^2 \\ 0 & 1 & 2y_1 & 3y_1^2 \end{pmatrix}^{-T}$$

Then the polynomial $\sum a_{ij} x^i y^j$ matches all the appropriate values and derivatives at the corners.

2.3.2 Bicubic interpolation: Precomputed matrix

For fast bicubic interpolation, we take an array with $16N^2$ values, specifying a bicubic form for each square in the grid, and construct a sparse matrix similarly to the bilinear algorithm. For a particular (x_c, y_c) and given ρ, the gyroaverage has the same form

$$\sum_p \int_{\gamma_p}^{\gamma_{p+1}} \tilde{f}_p(x_c + \rho\sin(\gamma), y_c + \rho\cos(\gamma))\,\mathrm{d}\gamma$$

where in the bicubic case

$$\tilde{f}_p(x,y) = \sum_{i,j=0}^{3} a_{ij;p} x^i y^j$$

These terms are each analytically integrable. The formulas are too large to include here. The source code that handles this integration was generated by SAGE and some post-processing, about is about 100 lines of code (for all 16 integrals).

In the bicubic case, we form a sparse matrix which operates on the matrix of bicubic polynomial coefficients. The breakup of circles into arcs is exactly the same as in the bilinear case. The contribution of each arc to a particular gyroaverage takes exactly four times as many parameters (and the relationship is still linear.) Thus applying this sparse matrix has the asymptotic complexity ($O(n^3)$) as the previous case, and precisely four times as many nonzeros entries.

2.3.3 Bicubic Interpolation: Gyroaveraging

After the precomputation step, the gyroaveraging calculation has three steps:

1. Calculate finite difference approximations for f_x, f_y, and f_{xy} (this can be represented as a stencil or a sparse matrix).
2. Solve for bicubic parameters on each grid space (this is $16N^2$ elements)
3. Multiply bicubic parameter matrix by sparse precomputed gyroaverage operator matrix (this is $O(N^3)$ as discussed above)

As in the bilinear case, the precomputed sparse matrix depends only on the domain and ρ. The asymptotic complexity is the same, and the flop count almost precisely four times larger than the bilinear case. We expect similar scaling but greater accuracy from higher order interpolation.

2.4 Fourier transform based gyroaveraging

The next scheme is suggested by (TODO) and is intended for the case where f is numerically compactly supported, i.e. the function and all of its derivatives are 0 on the boundary. In this case we can view the function as periodic, in which case the conclusions of TODO apply. In this case, the gyroaveraging operator is diagonal, and we we can compute a gyroaverage with the following algorithm:

1. First, pad the sample data with zeros on all sides. The size of this padding needs to be as large as the largest ρ under consideration. If we failed to pad, this algorithm would interpret f as periodic.

2. Second, transform the (padded) sample data via a 2-dimensional DCT (discrete cosine transform). If we call the padded data f'_{ij}, then we have a matrix $a_p q$ and the decomposition of the sample data as

$$f'_{ij} = \sum_{p=0}^{N-1}{}' \sum_{q=0}^{N-1}{}' \hat{f}'_{pq} \cos\left(\frac{\pi p(2i+1)}{2N}\right) \cos\left(\frac{\pi q(2j+1)}{2N}\right)$$

   The primed sums indicate that for $p = 0$ or $q = 0$ we add a $\frac{1}{2}$ coefficient before $\hat{f}'_{pq}$. There are various conventions; this is the convention in the FFTW package with the parameter REDFT10.

3. Perform an entry-wise ("Hadamard") multiplication of this matrix by the matrix B_{pq} where

$$B_{pq} = \frac{J_0\left(\left(\frac{\pi p \rho (N-1)}{Nw}\right)^2 + \left(\frac{\pi q \rho (N-1)}{Nw}\right)^2\right)}{4N^2}$$

   where J_0 is the Bessel function of the first kind and w is the width of the expanded domain after padding.

4. Apply the inverse discrete cosine transform (in the FFTW package, this is named "REDFT01") to the matrix $B_{pq}\hat{f'}_{pq}$

5. Throw out the matrix elements corresponding to the padding.

For this algorithm, we expect spectral accuracy for smooth functions where the boundary conditions are satisfied, and poor convergence (including Gibbs-style artifacts) when they are not. The algorithmic complexity is $O(N^2 \log n)$. The matrix of Bessel values can be precomputed and cached, and with sufficient care (and spare memory) the padding can be reused from one gyroaveraging to the next. Thus the main bottleneck in this algorithm is the extremely well optimized FFT-based DCT calculation, and scales with parallelism as the FFT does.

2.5 Chebyshev Spectral gyroaveraging

Our final interpolation method is bivariate Chebyshev interpolation. Here f is represented by the bivariate Chebyshev series

$$\tilde{f}(x,y) = \sum_{p=0}^{N-1}{}'' \sum_{q=0}^{N-1}{}'' \hat{f}_{pq} T_p(-x) T_q(-y)$$

where here the primed sums indicate the first and last coefficients are halved.

The Chebyshev series is computed stably and efficiently via a DCT, given that we have samples at the Chebyshev nodes. Indeed, let $x_m = -\cos\left(\frac{m\pi}{N-1}\right)$, $y_n = -\cos\left(\frac{n\pi}{N-1}\right)$, for $0 \le m, n \le N-1$. Define the function $D_{pq}(x,y) = a_p b_q T_p(-x) T_q(-y)$. Here a_p and b_q are indicators which "undo" the double primed summation below; $a_p = \frac{1}{2}$ for $p = 0$ or $N-1$ and $a_p = 1$ otherwise, and the same for b_q. These are the basis functions against which we are decomposing f. We prove, below, a special case of "orthogonality relations", i.e. that the particular DCT below, applied to one of these functions, results in a matrix with a single nonzero entry.

At any of the nodes (x_m, y_n) we have

$$D_{pq}(x_m, y_n) = a_p b_q T_p(-x_m) T_q(-y_n) = a_p b_q \cos\left(p \cos^{-1}(-x_m)\right) \cos\left(q \cos^{-1}(-y_n)\right) =$$

$$a_p b_q \cos\frac{mp\pi}{N-1} \cos\frac{nq\pi}{N-1}$$

Furthermore, define the 2D type-I DCT (named "REDFT00" in the FFTW package), given an input matrix f_{mn}, as

$$\hat{f}_{k,\ell} = \sum_{m=0}^{N-1}{}'' \sum_{n=0}^{N-1}{}'' f_{mn} \cos\frac{m\pi k}{N-1} \cos\frac{n\pi \ell}{N-1}$$

Here the double primed summation means the first and last terms are halved. Then, if we apply this transform to a basis function D_{pq} evaluated at the Chebyshev nodes, we find that

$$\hat{D}_{k,l} = \sum_{m=0}^{N-1}{}'' \sum_{n=0}^{N-1}{}'' D_{pq}(x_m, y_n) \cos\frac{m\pi k}{N-1} \cos\frac{n\pi \ell}{N-1} =$$

$$\sum_{m=0}^{N-1}{}'' \sum_{n=0}^{N-1}{}'' a_p b_q \cos\frac{mp\pi}{N-1} \cos\frac{nq\pi}{N-1} \cos\frac{m\pi k}{N-1} \cos\frac{n\pi \ell}{N-1} =$$

$$a_p b_q \sum_{m=0}^{N-1}{}'' \left(\cos\frac{mp\pi}{N-1} \cos\frac{m\pi k}{N-1} \left(\sum_{n=0}^{N-1}{}'' \cos\frac{nq\pi}{N-1} \cos\frac{n\pi \ell}{N-1} \right) \right) =$$

Focusing on the term inside the parenthesis, by product-to-sum for cosines, we want

$$\frac{1}{2} \sum_{n=0}^{N-1}{}'' \left(\cos\frac{n(q+\ell)\pi}{N-1} + \cos\frac{n(q-\ell)\pi}{N-1} \right)$$

Adopt the notation $e(\theta) = e^{\frac{i\pi\theta}{N-1}}$, let $u = q + \ell, v = q - \ell$ then the above is

$$\frac{1}{4}\sum_{n=0}^{N-1}{}'' \left((e(q+\ell))^n + (e(-q-\ell))^n + (e(q-\ell))^n + (e(-q+\ell))^n\right)$$

Note that for $x \neq 1$, we have (this is just a "primed" geometric series):

$$\sum_{n=0}^{N-1}{}'' x^n = \frac{(x+1)(x^{N-1}-1)}{2(x-1)}$$

Applied here, this makes our sum

$$\frac{1}{8}\left(\frac{(e(u)+1)(-1^u-1)}{e(u)-1} + \frac{(e(-u)+1)(-1^u-1)}{e(-u)-1}\right) +$$
$$\frac{1}{8}\left(\frac{(e(v)+1)(-1^v-1)}{e(v)-1} + \frac{(e(-v)+1)(-1^v-1)}{e(-v)-1}\right)$$

Since $e(-\theta) = e(\theta)^{-1}$, after clearing denominators the above collapses to 0. This holds whenever $q \neq l$. By the same argument, the summation will be 0 unless $k = p$ as well.

The above shows that if f is given globally by

$$f(x,y) = \sum_{p=0}^{N-1}{}'' \sum_{q=0}^{N-1}{}'' a_{pq} D_{pq}(x,y)$$

Then the DCT will recover the coefficients a_{pq}. Furthermore, by decomposing f into this basis, we can compute the gyroaverage of f as the linear combination of gyroaverages of the basis functions D_{pq}. If the gyroaverages of the basis functions are known analytically, or with high precision, then error will mainly come from the Chebyshev approximation, which is spectral for smooth functions and enjoys many other extremal properties as outlined in TODO.

2.5.1 Dense Chebyshev gyroaveraging matrix: Precomputation

For this algorithm, we form as a dense matrix the operator which takes a vector (flattened 2D matrix) of Chebyshev coefficients (a_{pq} above, or $\hat{f}_{pq}$ below) and returns a vector (also a flattened 2D matrix) of gyroaverages. This matrix is a flattened version of a 4-parameter array. The entries that need to be computed are

$$I_\rho(p,q,i,j) = (\mathcal{G}D_{pq})(x_i,y_j) = \frac{1}{2\pi}\int_0^{2\pi} T_p(x_i + \rho\sin(\gamma))T_q(y_j + \rho\cos(\gamma)))\,\mathrm{d}\gamma$$

The above needs a few points of precision however:

1. The x_i, y_i can actually be any set of points. Our code actually targets equispaced points but targeting Chebyshev nodes has the same cost.

2. The above definition is assuming that $T_n(x)$ is defined to be identically 0 outside $[-1, 1]$. Alternatively we could have included an indicator function in our definition. This point is critical; we handle it in our code by specifying bounds of integration so that arcs outside the unit square are skipped.

This matrix is a dense matrix with N^4 entries. It scales poorly with N and is indeed quite expensive to precompute (multiple core-days for $N \approx 80$, on 2020 hardware). After trying some shortcuts, the current iteration of our code computes these integrals very literally, using the same adaptive Gauss-Kronrod quadrature from the Boost package as mentioned earlier (TODO). A few thoughts on this precomputation:

1. These integrals are all analytically integrable and look like (large, complicated) polynomials in $\sin(\gamma)$ and $\cos(\gamma)$. However, already for $p = q = 20$ the SAGE package seems to give up.

2. There are some more-or-less obvious symmetries which we do not currently exploit.

3. These integrals are somewhat oscillatory, especially for large p, q and near the boundary, and a Filon or Levin type quadrature might be much faster. Indeed, we saw some evidence of "thrashing" when running these precomputations.

4. We would ideally want these integrals accurate to machine precision, which is not really something one can guarantee unless the quadrature itself is done in higher precision. Defaulting to 128-bit doubles is probably better, but we have not implemented that. For the lower degree Chebyshev polynomials, Gauss-Kronod integration should be close to machine accurate anyway.

5. The values of $I_\rho(p, q, i, j)$ seem like they might obey a Clenshaw-style recurrence relation for fixed i, j and versus lower degrees of p, q. This is motivated by various recurrence formulas for T_n and its integrals and derivatives, as well as reduction-of-power formulas for integration of powers of trigonometric functions. This would speed up the precomputation (and indeed probably bring it to "real-time".) However we have not discovered this yet.

2.5.2 Chebyshev gyroaveraging: Algorithm

After the above dense matrix is precomputed, the gyroaveraging algorithm is simple:

1. Starting with f_{mn}, the values of f sampled at Chebyshev tensor product nodes, perform the below DCT (same as defined above)

$$\hat{f}_{k,\ell} = \sum_{m=0}^{N-1}{}'' \sum_{n=0}^{N-1}{}'' f_{mn} \cos\frac{m\pi k}{N-1} \cos\frac{n\pi \ell}{N-1}$$

   which computes the Chebyshev coefficients (with $O(N^2 \log(N)$ complexity.)

2. Flatten the above coefficient matrix into a vector with N^2 elements, and apply the precomputed dense $N^2 \times N^2$ matrix.

The dense multiplication is the bottleneck, and as we see later, can benefit greatly from parallelism (in particular GPU acceleration.) For this algorithm, we expect spectral accuracy for functions with are smooth on the rectangular domain.

Chapter 3

Numerical Results

3.1 Gallery of test functions

In this section we present numerical results, running the each gyroaveraging algorithm described above against the below test functions. In each case the domain is the unit square and we tested three values for ρ: $\{0.625, 0.46875, 0.875\}$.

1. The first function is from REF and has an analytic gyroaverage. The parameter A is set to 22 so this function has compact support up to machine double precision on the boundary of the unit square.

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

2. The second function is a bivariate version of Runge's function, which is known to cause difficulties for high-degree equispaced polynomial interpolation.

$$\text{SmoothRunge}(x, y) = \frac{(1-x^2)(1-y^2)}{1+25((x-0.2)^2+(y+0.5)^2)}$$

3. Next, we introduce a point singularity. The below horn is continuous but not differentiable at 0.

$$\text{NonsmoothSqrt}(x, y) = \sqrt{(x-0.2)^2+(y+0.5)^2}$$

4. Our next function has 3 non-differentiable kinks, at $x = y$ and $|x - y| = 0.75$.

$$\text{NonsmoothRidge}(x, y) = \max(0, 0.75 - |x-y|)^4(4|x-y|+1)$$

3.2 Testing environment and setup

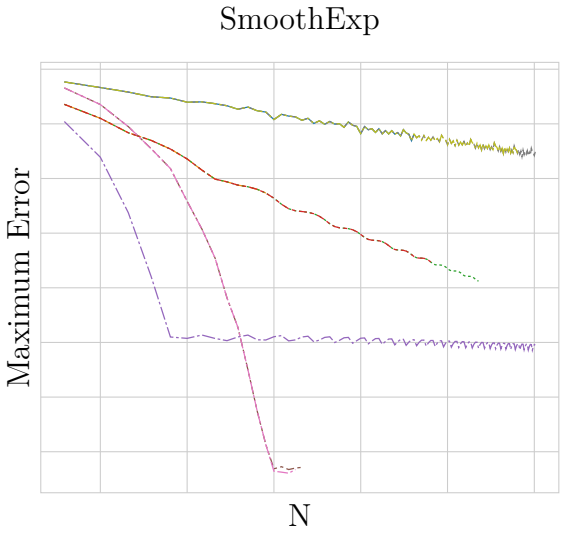
The benchmarking process is outlined below:

1. First, each algorithm was run on each function to completion for increasing N until a memory exception triggered. This has two effects
   (a) Precomputed matrices for the sparse and dense matrix-vector product algorithms are generated and cached
   (b) Precomputed benchmark gyroaverage for each function and N is generated and cached. This is calculated straight from the definition using black-box Gauss-Kronod quadrature from the Boost C++ library. For the smooth functions this was verified to be correct within one digit of machine precision vs a double-precision analytic calculation.
2. After a complete precomputation step, for each triple of (algorithm, function, N),
   (a) Run a single gyroaverage calculation, to warm up cache, and force initialization of GPUs and compilation of GPU machine code
   (b) Rerun the gyroaverage calculation, bechmarking the wall-clock time and storing the a vector of errors for each value of ρ
   (c) The error is defined, given matrices f'_{ij} the approximation and $\hat{f}_{ij}$ representing the benchmark, as

$$\frac{\max_{i,j} \left| \hat{f}_{ij} - f'_{ij} \right|}{\max_{i,j} \left| \hat{f}_{ij} \right|}$$

The benchmarks were run on the NYU Greene supercomputing cluster around December 2020, using two different classes of nodes. The CPU nodes are 48 cores (2 2.9Ghz Intel Xeon 24C CPUs) with 192Gb of memory. The GPU nodes are additionally equipped with 4 RTX8000 GPUs. We used, for our testing, a single GPU at a time and a maximum of 20 cores. In all cases the input data and output data are held in CPU memory and the full penalty of GPU memory transfer is included in the benchmarks.

3.3 Convergence Study



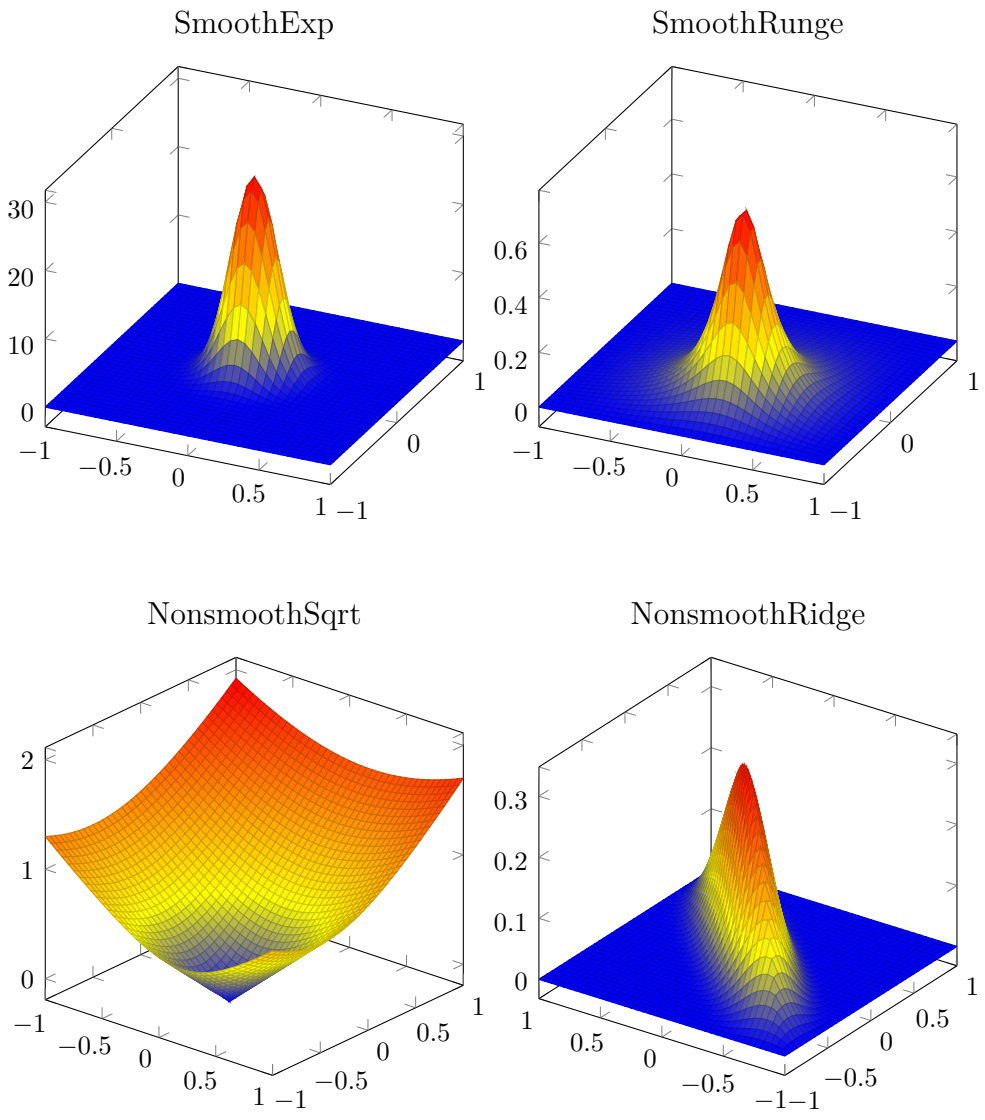


Figure 3.1: Gallery of test functions

Appendix A

One more comment

This is an appendix.

Bibliography

[1] J. B. Conway, *Functions of One Complex Variable I*. Second edition. Springer-Verlag, Graduate Texts in Mathematics **11**, 1991.