Sparse Zonal Harmonic Factorization for Efficient SH Rotation: Supplemental Material and Implementation Sketch

Derek Nowrouzezahrai[1,2,3], Patricio Simari[4], and Eugene Fiume[3]
[1]Université de Montréal, [2]Disney Research Zurich, [3]University of Toronto, [4]Autodesk Research

This document supplements the submitted manuscript with elaborations on our investigation motivating the selection criteria for sparsity-promoting candidate lobe directions, details on forming double- and triple-product integrals entirely (or partially) in the RZHB, and includes source code for both the *standard* and GPU-accelerated *signal-tailored* rotation algorithms. Moreover, we include MATLAB code for performing the one-time optimized selection of lobe directions/weights (including *lobe sharing*), and a table of lobe directions up to $N = 8$. This information will not only facilitate immediate integration of our technique into existing spherical harmonic rendering frameworks, but also bootstraps further investigation of lobe direction optimization.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Color, shading, shadowing, & texture*

Additional Key Words and Phrases: Spherical harmonic rotation, Real-time rendering, GPU

ACM Reference Format:

Nowrouzezahrai, D., Simari, P., and Fiume, E. 2010. Sparse Zonal Harmonic Factorization for Efficient SH Rotation and Shading: Supplemental Material and Implementation Sketch. ACM Trans. Graph. XX, Y, Article ZZZ (Month 2010), 9 pages.
DOI = xx.xxxx/xxxxxxx.xxxxxxx
http://doi.acm.org/xx.xxxx/xxxxxxx.xxxxxxx

1. OUTLINE

We begin with a discussion outlining some of our initial investigation towards determining metrics for promoting sparsity in $\hat{\mathsf{A}}_l$ and Y_l (Section 2). Ultimately, the conclusion of these experiments was that selecting candidate lobe directions from the zeros of band-l SH basis functions would promote sparsity in both $\hat{\mathsf{A}}_l$ and Y_l.

Next, we outline the derivation of coupling coefficients in the RZHB, for double-product integration where either one or both terms of the double-product integrand are expressed with their ZHF (Section 3).

GPU shader code for the signal-tailored rotation, CPU code for the standard rotation algorithms, and MATLAB code to perform the (one-time) evolutionary optimization procedure for determining the final lobe directions and weights are included in Section 4. We also include optimized sparse matrix-vector multiplication, up to $N = 6$, for our two algorithms.

Lastly, a table of sparsity-optimized lobe directions, up to $N = 8$, are included in Section 6. Plugging subsets of these directions (according to the definition of Ω in the paper) into each Y_l, inverting the matrix, and factoring out the D_l components will yield the lobe weights.

2. PROMOTING SPARSITY IN THE ZHF AND ZHE

Sparsity arises in (the rows of) $\hat{\mathsf{A}}_l$ when lobe directions are chosen such that a particular band-l SH basis function can be perfectly represented using a weighted sum of strictly fewer than $2l + 1$ rotated ZH lobes. For example, Section 4 in the paper showed that each band-0 SH basis function could be represented with $l = 1$, as opposed to $2l + 1 = 3$, ZH lobes by aligning the lobe directions along the x, y, and z axes.

Next we will briefly discuss the path that led towards the selection criteria for candidate lobe directions, which are in turn used in the one-time sparse optimization (Listing 3).

Investigating sparsity:. a row in $\hat{\mathsf{A}}_l$ maps a single SH basis function into the RZHB; if a row is sparse, its corresponding SH basis function can be represented with *fewer* than $2l + 1$ rotated ZH lobes.

To our knowledge, there is no predictable structure to the potential sparsity in $\hat{\mathsf{A}}_l$, and so we initially experimented with several approaches to search for sparse solutions. Naïvely minimizing the l_1-norm of $\hat{\mathsf{A}}_l$ resulted in a net reduction of the magnitude of row elements but did not yield sparse solutions. Of our initial trials, the most successful approaches involved either simultaneously minimizing the l_1-norm of Y_l and $[\mathsf{Y}_l]^{-1}$, or concurrently applying logarithmic biasing to the Y_l and $[\mathsf{Y}_l]^{-1}$ matrices,

$$\underset{\forall d,\, [\omega_{l,d}]}{\operatorname{argmin}} \sum_{ij} \log\left(\left|[\hat{\mathsf{A}}_l]_{ij}\right| + 1\right) + \log\left(\left|[\mathsf{Y}_l]_{ij}\right| + 1\right). \quad (1)$$

Both approaches result in similarly sparse solutions but were sensitive to the initial (random) setting of the lobe directions.

The first term in Equation 1 reduces the magnitude of $\hat{\mathsf{A}}_l$'s elements while the second term effectively motivates the lobe directions to align with zeros of the SH basis functions. These ap-

---

Derek Nowrouzezahrai acknowledges funding from the National Sciences and Engineering Research Council of Canada (NSERC), the Canadian Research Network for Mathematics of Information Technology and Complex Systems (MITACS), the Ontario Ministry of Research and Innovation (MRI), and the Ontario Ministry of Education and Training. {derek@iro.umontreal.ca, patricio.simari@autodesk.com, elf@dgp.toronto.edu}
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0730-0301/YYYY/12-ARTXXX $10.00
DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

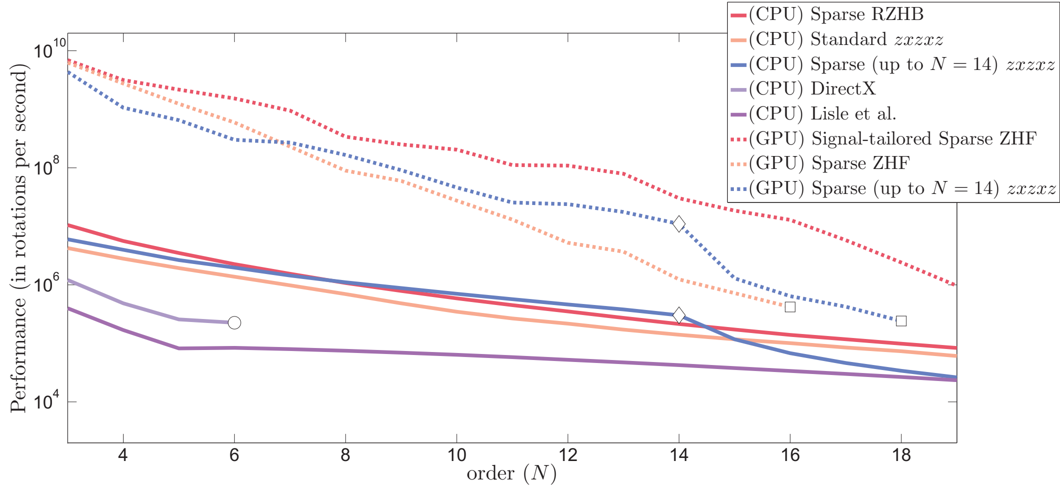

Fig. 1. Comprehensive performance comparison plot.

proaches hinted to a potentially optimal set of lobe directions (in terms of a sparsity metric) that relate to the zeros of SH functions.

To further investigate this potential, we computed the analytic form of $[\mathsf{Y}_l]^{-1}$ for several l and noticed that, in all cases, many of the elements of the inverse shared common sub-expressions with the elements in Y_l; namely, polynomials in $\cos\theta$ and sinusoids in ϕ with frequencies between $\pm l/(2\pi)$. Hence, empirical evidence suggests that, apart from clearly promoting sparsity in Y_l, choosing lobe directions that align with the zeros of SH basis functions will also promote sparsity in $\hat{\mathsf{A}}_l = [\mathsf{Y}_l]^{-1}$.

3. COMBINED DOUBLING AND TRIPLING COEFFICIENTS WITH SH AND RZHB

Equation 13 in the paper reduces to a dot product since the SH coupling matrix is the identity: $\int_{S^2} y_i(\omega)\, y_j(\omega)\, d\omega = \delta_{ij}$, where the Kroenecker delta, δ_{ab}, is 1 when $a = b$ and 0 otherwise.

If we represent one of the two terms in the integrand of Equation 13 with its ZHF, then the elements of the coupling matrix become

$$
\begin{aligned}
[\mathsf{C}^{\mathrm{D}}]_{ij} &= \int_{S^2} y^0_{l(i)}(\omega \to \omega_{l(i),i-l(i)})\, y_j(\omega)\, d\omega \\
&= \sum_{k=-l(i)}^{l(i)} y^k_{l(i)}(\omega_{l(i),i-l(i)}) \int_{S^2} y_i(\omega)\, y_j(\omega)\, d\omega \\
&= y_j(\omega_{l(i),i-l(i)})\,, \qquad (2)
\end{aligned}
$$

where $l(i)$ maps single basis function indices to band indices, and we abuse a combination of single and double indexing. Note that $\mathsf{C}^{\mathrm{D}} = \mathsf{Y}$, and thus, can be computed and stored once.

If both of the terms in the double product integrand are represented in the RZHB, with coefficients $\hat{\boldsymbol{z}}_{\boldsymbol{l}}$ and $\hat{\boldsymbol{z}}_{\boldsymbol{t}}$, Equation 13 simplifies to

$$
L_o(x, \omega_o) = [\mathsf{Y}^{\mathrm{T}} \cdot \hat{\boldsymbol{z}}_{\boldsymbol{l}}]^{\mathrm{T}} \cdot [\mathsf{Y}^{\mathrm{T}} \cdot \hat{\boldsymbol{z}}_{\boldsymbol{t}}] = \hat{\boldsymbol{z}}_{\boldsymbol{l}} \cdot \mathsf{Y} \cdot \mathsf{Y}^{\mathrm{T}} \cdot \hat{\boldsymbol{z}}_{\boldsymbol{t}}\,, \qquad (3)
$$

where the coupling matrix is $\mathsf{Y} \cdot \mathsf{Y}^{\mathrm{T}}$ and either of the matrices in this product can be replaced by Y_{R} if a term is rotated.

Similar expressions for triple-product integrals can be derived in the cases where one, two, or all three terms of the integrand are expressed with their ZHF expansion.

4. CODE SAMPLES

We outline HLSL source code for *signal-tailored* SH rotation in Listing 1, standard SH rotation in Listing 2, and the MATLAB lobe direction optimization (including *lobe sharing*) in Listing 3.

Both rotation algorithms assume hard-coded sparse matrix-vector multiplication functions exist, and we provide example code for these functions up to $N = 6$ in Listings 4 and 5. The MATLAB code leverages the *Genetic Algorithm and Direct Search toolbox*.

5. COMPREHENSIVE PERFORMANCE PLOT

We include a comprehensive performance plot above in Figure 1, where we additionally benchmark against Lisle et al.'s SH rotation (which uses slower, more general SH basis function evaluation code, as opposed to our optimized SH basis function evaluation routines), the DirectX SDK rotation code, and our independent implementation of $zxzxz$ rotation which does not hard-code sparse matrix-vector multiplications (unlike the code used in our GPU implementation or for CPU benchmarking), instead relying on loops to construct these products. The DirectX rotation API only supports up to $N = 6$ (low-order limit marked with $\circ$ in Figure 1) and $\square$ marks the point at which standard ZHF and $zxzxz$ shader implementations fail to compile. As in the main document, $\diamondsuit$ marks the point after which loops are used to compose higher-order x-rotations in the $zxzxz$ implementation.

6. TABLE OF SPARSITY-OPTIMIZED LOBE DIRECTIONS

Figure 2 below lists lobe directions (assuming *lobe sharing*) up to $N = 8$. Four decimal values are used for spacing and layout purposes. Below the list in the table is a visualization of the distribution of the lobe directions, up to $N = 7$. The code in Listing 3 was used to determine these values and plugging the lobe directions into Y and inverting the matrix yields the necessary lobe weights for ZHF.

Received December 2010; accepted Month Year

Listing 1. HLSL code for signal-tailored SH rotation.

```
#define N       // SH Order

// The per-band SH reconstructions of the original function.
TextureCube funcExpand[N];

// The signal-tailored algorithm uses the inverted rotation of the
// lobe directions to sample the per-band SH reconstructions.
float3x3 invertedRotation;

// The RZHB lobe directions are passed from the CPU *once* at
// initialization, or hardcoded for a fixed N in the shader. We use
// lobe sharing, otherwise there would be N^2 lobe directions.
float3 lobeDir[2*N-1];

// Signal-tailored rotation shader
void SHRotation(out float4 rotCoeffs[N*N]) {
 // Here, we assume an RGBA function is being rotated.
 float4 sample[N*N];

 // Rotated lobe directions
 float3 rotDir[2*N-1];

 for(int d = 0; d < 2*N-1; d++)
   rotDir[d] = mul(lobeDir[d],
                   invertedRotation);

 // Sample with bilinear texture interpolation and clamping. Here
 // we assume lobe sharing with nested lobe directions (see Ω in
 // the paper). DC/Linear bands are handled separately (see text).
 for(int l = 2; l < N; l++)
  for(int m = -l; m <= l; m++) {
   int i = l*(l+1)+m; // SH single index
   int d = m+l; // shared lobe index
   sample[i]=funcExpand[l].Sample(linClamp,
                                  rotDir[d]);
  }

 // Sparse Â multiplication
 AhatMultiply(sample, rotCoeffs);
}
```

Listing 2. C code for standard SH rotation algorithm.

```
// Number of ZH lobes, assuming lobe sharing
#define NUM_LOBES (2*(N-1)+1)

void SHRotation(matrix *pmRot,
                float* inSH, float *outSH){
 vector3 rotLobeDirections[NUM_LOBES];
 vector3 temp, optimalLinearDir;
 matrix mRInv;

 // To store the elements of Y^R
 float SHEval_RotLobes[NUM_LOBES][N*N];

 // RZHB coefficients: ẑ
 float z[N*N];

 // Trivial DC rotation and optimal linear lobe rotation
 outSH[0] = inputSH[0];
 temp.x=-inSH[3];temp.y=-inSH[1];temp.z=inSH[2];
 MatrixInverse( mRInv, NULL, pmRot );
 Vec3Transform( optimalLinearDir, temp, mRInv );
 outSH[3] = -optimalLinearDir.x;
 outSH[1] = -optimalLinearDir.y;
 outSH[2] = optimalLinearDir.z;

 // Rotate all ZHF lobe axes by mRot (for L > 1).
 // The optimized lobes are stored in ZHFLobes (see Section 6).
 Vec3TransformArray(rotLobeDirections,ZHFLobes,
                    mInv,NUM_LOBES);

 // Evaluate order-N SH basis functions at rotated lobe directions
 for(unsigned lobe=0; lobe<NUM_LOBES; lobe++)
  SHEvalBasis(SHEval_RotLobes[lobe],N,
              rotLobeDirections[lobe]);

 // Sparse (transposed) matrix multiplication: ẑ = Â^T · f
 // Hard-coded sparse (transposed) matrix multiplication for N=3.
 AhatTransposeMultiply(inSH,z);

 //Map back to SH with dense matrix multiplication
 for(unsigned l=2; l <= N-1; l++ ) {
  unsigned x=l*l;
  unsigned num_band_bf=2*l+1;
  // Transposed matrix multiplication: f = Y^T · ẑ
  // Assumes lobe sharing (see Ω in paper).
  for(unsigned j=0;j<num_band_bf;j++)
   for(unsigned i=0;i<num_band_bf;i++)
    outSH[x+j]+=SHEval_RotLobes[i][x+j]*z[x+i];
 }
}
```

Listing 3. MATLAB code for (one-time) sparse $\omega_{l,d}$ optimization.

```
function [x_max, max_frac, max_weights] = ...
         FindWeightsAndLobes(L, x_max)

 % Load candidate lobe directions: zeros of SH basis functions.
 candidates=loadCandidatesDirections(l);
 total_entries = 0;
 for k=2:l
  total_entries=total_entries+(2*k+1)^2;
 end

function nrm = L_norm(m,l)
 nrm=sqrt((2*l+1)*factorial(l-abs(m)));
 nrm=nrm/(4*pi*factorial(l+abs(m))));
 if m~=0
  nrm=nrm*sqrt(2);
 end
end

% Outputs row-vector of band-l SH basis functions
% @ ω = (θ,φ) = (t,p) for all −l ≤ m ≤ l
function ylm = bandL_Ylm(t,p,l)
 m=0:1:l;
 n=arrayfun(@L_norm,m,l(ones(1,length(m))));
 Plm=legendre(l,cos(theta))';
 n_Plm=n.*Plm;
 % extend to m < 0 terms
 Plm=n_Plm(abs(l:-1:-l)+repmat(1,1,2*l+1));
 ylm=Plm.*[sin(p*abs(-l:1:-1)),
           1,cos(p*(1:1:l))];
end

% Objective function to minimize: maximize sparsity of Y^-1
% format of input→[thetas,phis]
function [value,Y,Yi]=sparsity(x)
 m=numel(x)/2;
 l=(m-1)/2; % band index
 T=x(1:m); % thetas
 P=x(m+1:end); % phis
 Y=zeros(m,m);
 for i=1:m
  Y(i,:)=bandL_Ylm(T(i),P(i),l);
 end

 % Check for ill-conditioning
 if abs(det(Y))>1e-10
  Yi=inv(Y);
  value=sum(abs(Yi(:))<1e-9);
 else
  value=-Inf; Y = NaN; Yi = NaN;
 end
end

% Compute order-N sparsity, summing up per-band sparsities.
function [frac,weights]=global_sparsity(x)
 m=numel(x)/2;
 total_sparsity=0;

 if nargout>1
  weights= cell(L,1);
 end

 for i=2:L
  [s,Y,Yi]=sparsity([x(1:(2*i+1)),
                     x(m+(1:(2*i+1)))]);
  if s<0
   frac=-Inf;
   return;
  else
   total_sparsity=total_sparsity+s;
   if nargout>1
    weights{i}=Yi;
   end
  end
 end
 frac=total_sparsity/total_entries;
end

% The genetic algorithm's population selection
function population = create(len,fitness,opt)
 npop=sum(opt.PopulationSize);
 population = zeros(npop,len);
 for i=1:npop
  for c=1:length(candidates)
   population(i,c)=pickRand(candidates{c});
  end
 end
end

% The genetic algorithm's mutation function
function children = mutation(parents,opt,
                             nvars,fitness,
                             state,score,pop)
 children=pop(parents,:);
 rate=1/nvars;
 for ch=1:size(children,1)
  for c=1:nvars
   if rand<rate
    children(ch,c)=pickRand(candidates{c});
   end
  end
 end
end

if nargin<2
  % initialize options for genetic algorithm
  opts=gaoptimset('PopulationType','Custom',
       'PopulationSize',500,'Generations',Inf,
       'CrossoverFraction',.8,'EliteCount',2,
       'StallTimeLimit',Inf,'StallGenLimit',50,
       'TolFun',0,'CreationFcn',@create,
       'MutationFcn',@mutation,'CrossoverFcn',
       @crossoverscattered,'SelectionFcn',
       @selectiontournament,'Display','iter',
       'UseParallel','always');
 [x_max,max_frac]=ga(@(x)(-global_sparsity(x)),
  length(candidates),[],[],[],[],[],[],[],opts);
 frac_max=-frac_max;
end

[max_frac,max_weights]=global_sparsity(x_max);
end
```

Listing 4. HLSL code for sparse $\hat{A}$ multiplication for orders 3 to 6.

```
// Sparse matrix multiplication for N=3.
void AhatMultiply(float4 input[N*N], out float4 output[N*N]) {
 output[4] = 1.37294f * input[5] + 1.37294f * input[7];
 output[5] = 1.37294f * input[5] + 1.37294f * input[8];
 output[6] = 1.58533f * input[6];
 output[7] = -1.37294f * input[7] + -1.37294f * input[8];
 output[8] = -1.83058f * input[4] + -0.915291f * input[6];
}

// Sparse matrix multiplication for N=4.
void AhatMultiply(float4 input[N*N], out float4 output[N*N]) {
 output[4] = 1.58533f * input[4] + 2.11378f * input[5] + 1.05689f * input[6];
 output[5] = 0.457646f * input[4] + 1.83058f * input[6] + -2.28823f * input[8];
 output[6] = 1.58533f * input[4];
 output[7] = -1.37294f * input[4] + -1.83058f * input[6] + -2.28823f * input[7];
 output[8] = -0.915291f * input[4] + -1.83058f * input[6];
 output[9] = -1.18427f * input[13] + 1.18427f * input[15];
 output[10] = 1.67481f * input[13] + 2.23308f * input[14] + -0.55827f * input[15] + 1.498f * input[9];
 output[11] = 2.18797f * input[11] + 1.52889f * input[13] + -1.52889f * input[15];
 output[12] = -1.33985f * input[9];
 output[13] = -2.52644f * input[10] + -1.26322f * input[11] + -2.64811f * input[13] + 2.64811f * input[15];
 output[14] = 0.966953f * input[13] + 0.966953f * input[15] + 0.864869f * input[9];
 output[15] = 2.36854f * input[12] + 1.18427f * input[13] + 1.18427f * input[15] + 2.11849f * input[9];
}

// Sparse matrix multiplication for N=5.
void AhatMultiply(float4 input[N*N], out float4 output[N*N]) {
 output[4] = 1.83058f * input[4] + 0.915291f * input[8];
 output[5] = 3.17066f * input[4] + 2.61566f * input[5] + -2.242f * input[6] + -3.69911f * input[7] + 0.154778f * input[8];
 output[6] = 1.58533f * input[8];
 output[7] = -2.61566f * input[5] + 2.242f * input[6] + 0.373666f * input[8];
 output[8] = 1.83058f * input[6] + 0.915291f * input[8];
 output[9] = -1.44659f * input[11] + 0.248196f * input[14] + -2.21435f * input[15] + 0.847395f * input[9];
 output[10] = -0.399466f * input[11] + 2.13568f * input[12] + -0.922527f * input[13] + 0.399466f * input[14]
             + -1.12986f * input[9];
 output[11] = 1.86755f * input[11] + 1.86755f * input[14] + 2.85871f * input[15] + -1.09398f * input[9];
 output[12] = -1.33985f * input[13];
 output[13] = -2.96153f * input[11] + -0.773562f * input[14] + -2.85871f * input[15] + 2.64111f * input[9];
 output[14] = 2.13568f * input[10] + 0.165464f * input[11] + -0.922527f * input[13] + -0.399466f * input[14]
             + -1.47623f * input[15] + 1.36386f * input[9];
 output[15] = -0.599199f * input[11] + -0.599199f * input[14] + -2.21435f * input[15] + 2.04579f * input[9];
 output[16] = -1.12986f * input[16] + -1.12986f * input[18] + 0.248196f * input[20] + 1.59786f * input[22];
 output[17] = -1.46773f * input[18] + 1.1008f * input[20] + 0.815406f * input[21] + 1.99775f * input[23];
 output[18] = -2.11378f * input[16] + -1.05689f * input[18] + 1.58533f * input[20] + -1.05689f * input[21];
 output[19] = -2.13568f * input[17] + -1.61938f * input[18] + 3.02031f * input[19] + -2.60553f * input[20]
             + 6.14712f * input[21] + 2.26523f * input[23] + -4.53046f * input[24];
 output[20] = 1.18164f * input[20];
 output[21] = 2.13568f * input[17] + 0.140048f * input[18] + 1.98914f * input[20] + -2.44879f * input[21]
             + 2.26523f * input[24];
 output[22] = -1.05689f * input[18] + 1.05689f * input[21];
 output[23] = 0.815406f * input[18] + 1.1008f * input[20] + -1.46773f * input[21] + 1.99775f * input[24];
 output[24] = 0.798932f * input[18] + -0.599199f * input[20] + 0.798932f * input[21];
}
```

```
// Sparse matrix multiplication for N=6.
void AhatMultiply(float4 input[N*N], out float4 output[N*N]) {
 output[4] = 0.915291f * input[4] + 1.83058f * input[8];
 output[5] = -0.647209f * input[4] + -1.29442f * input[5] + 1.94163f * input[6];
 output[6] = 1.58533f * input[4];
 output[7] = 0.268083f * input[4] + -0.454539f * input[5] + -0.804248f * input[6] + -2.1016f * input[7] + 0.990704f * input[8];
 output[8] = -0.915291f * input[4] + -1.83058f * input[5];
 output[9] = 0.847395f * input[10] + 0.599199f * input[13] + 0.599199f * input[15];
 output[10] = 1.05911f * input[10] + -1.7976f * input[11] + -2.54219f * input[12] + 0.691895f * input[13]
              + -0.573185f * input[14] + 1.751f * input[15] + 1.67038f * input[9];
 output[11] = 1.09398f * input[10] + -0.773562f * input[13] + -0.773562f * input[15];
 output[12] = 1.33985f * input[9];
 output[13] = 0.453142f * input[10] + 0.32042f * input[13] + -1.6746f * input[14] + 1.22671f * input[15];
 output[14] = 1.7976f * input[11] + -0.691895f * input[13] + -0.691895f * input[15] + -0.691895f * input[9];
 output[15] = -0.351003f * input[10] + -1.44659f * input[13] + 1.29714f * input[14] + 0.248196f * input[15];
 output[16] = -1.44659f * input[16] + 1.12986f * input[17] + 1.59786f * input[21] + 1.12986f * input[22];
 output[17] = 0.043814f * input[16] + -0.185451f * input[17] + -1.30453f * input[18] + -0.317406f * input[20]
              + -0.317406f * input[22] + 0.758453f * input[23];
 output[18] = -1.05689f * input[20] + 1.05689f * input[22];
 output[19] = 1.99785f * input[16] + -3.73689f * input[17] + -1.30453f * input[18] + -2.33088f * input[20]
              + -2.33088f * input[22] + -2.00668f * input[23];
 output[20] = 1.18164f * input[16];
 output[21] = -66.4551f * input[16] + 84.4028f * input[17] + 60.1671f * input[18] + -53.6759f * input[19] + 53.8045f * input[20]
              + 23.856f * input[21] + 47.0193f * input[22] + -15.3268f * input[23] + 44.6846f * input[24];
 output[22] = -1.58533f * input[16] + 2.11378f * input[17] + 1.05689f * input[20] + 1.05689f * input[22];
 output[23] = -65.9212f * input[16] + 80.4805f * input[17] + 57.5581f * input[18] + -53.6759f * input[19]
              + 48.9812f * input[20] + 23.856f * input[21] + 47.2184f * input[22] + -16.575f * input[23] + 40.7741f * input[24];
 output[24] = 0.599199f * input[16] + -0.798932f * input[20] + -0.798932f * input[22];
 output[25] = -1.14263f * input[26] + -0.524785f * input[27] + -0.524785f * input[34];
 output[26] = 0.180665f * input[25] + -0.368734f * input[26] + -1.87753f * input[28] + -0.890202f * input[29]
              + -0.681331f * input[30] + -0.368734f * input[31] + 0.890202f * input[35];
 output[27] = 0.510998f * input[26] + 0.704073f * input[27] + -0.722661f * input[29] + -0.722661f * input[31]
              + 0.704073f * input[34];
 output[28] = 0.815982f * input[25] + -2.69405f * input[26] + -2.25505f * input[27] + 2.30433f * input[29]
              + -2.0098f * input[30] + 0.543848f * input[31] + 1.73971f * input[33] + -1.36306f * input[34]
              + -0.611963f * input[35];
 output[29] = -1.10388f * input[26] + 0.780563f * input[29] + 0.780563f * input[31];
 output[30] = 1.06883f * input[25];
 output[31] = 0.780563f * input[29] + -0.780563f * input[31] + -1.10388f * input[35];
 output[32] = -0.644318f * input[25] + 0.640177f * input[26] + 0.30334f * input[27] + -0.701003f * input[29]
              + -0.701003f * input[31] + -1.73971f * input[32] + 1.19533f * input[34];
 output[33] = -2.97832f * input[26] + -1.69978f * input[27] + 2.46732f * input[29] + -2.67061f * input[30]
              + -1.69978f * input[34] + -0.722661f * input[35];
 output[34] = 0.180665f * input[25] + -0.938764f * input[27] + 0.938764f * input[34];
 output[35] = -2.21991f * input[26] + -1.26694f * input[27] + 1.30039f * input[29] + -1.99055f * input[30]
              + 0.53864f * input[31] + -1.26694f * input[34] + -1.30039f * input[35];
}
```

Listing 5. C code for sparse $\hat{A}^T$ multiplication for orders 3 to 6.

```
// Sparse matrix multiplication for N=3.
void AhatTransposeMultiply(const float * in, float * out) {
 out[4] = -1.83058f*in[8];
 out[5] = 1.37294f*in[4] + 1.37294f*in[5];
 out[6] = 1.58533f*in[6] + -0.915291f*in[8];
 out[7] = 1.37294f*in[4] + -1.37294f*in[7];
 out[8] = 1.37294f*in[5] + -1.37294f*in[7];
}

// Sparse matrix multiplication for N=4.
void AhatTransposeMultiply(const float * in, float * out) {
 out[4] = 1.58533f*in[4] + 0.457646f*in[5] + 1.58533f*in[6] + -1.37294f*in[7] + -0.915291f*in[8];
 out[5] = 2.11378f*in[4];
 out[6] = 1.05689f*in[4] + 1.83058f*in[5] + -1.83058f*in[7] + -1.83058f*in[8];
 out[7] = -2.28823f*in[7];
 out[8] = -2.28823f*in[5];
 out[9] = 1.498f*in[10] + -1.33985f*in[12] + 0.864869f*in[14] + 2.11849f*in[15];
 out[10] = -2.52644f*in[13];
 out[11] = 2.18796f*in[11] + -1.26322f*in[13];
 out[12] = 2.36854f*in[15];
 out[13] = -1.18427f*in[9] + 1.67481f*in[10] + 1.52889f*in[11] + -2.64811f*in[13] + 0.966953f*in[14] + 1.18427f*in[15];
 out[14] = 2.23308f*in[10];
 out[15] = 1.18427f*in[9] + -0.55827f*in[10] + -1.52889f*in[11] + 2.64811f*in[13] + 0.966953f*in[14] + 1.18427f*in[15];
}

// Sparse matrix multiplication for N=5.
void AhatTransposeMultiply(const float * in, float * out) {
 out[4] = 1.83058f*in[4] + 3.17066f*in[5];
 out[5] = 2.61566f*in[5] + -2.61566f*in[7];
 out[6] = -2.242f*in[5] + 2.242f*in[7] + 1.83058f*in[8];
 out[7] = -3.69911f*in[5];
 out[8] = 0.915291f*in[4] + 0.154778f*in[5] + 1.58533f*in[6] + 0.373666f*in[7] + 0.915291f*in[8];
 out[9] = 0.847395f*in[9] + -1.12986f*in[10] + -1.09398f*in[11] + 2.64111f*in[13] + 1.36386f*in[14] + 2.04579f*in[15];
 out[10] = 2.13568f*in[14];
 out[11] = -1.44659f*in[9] + -0.399466f*in[10] + 1.86755f*in[11] + -2.96153f*in[13] + 0.165464f*in[14] + -0.599199f*in[15];
 out[12] = 2.13568f*in[10];
 out[13] = -0.922527f*in[10] + -1.33985f*in[12] + -0.922527f*in[14];
 out[14] = 0.248196f*in[9] + 0.399466f*in[10] + 1.86755f*in[11] + -0.773562f*in[13] + -0.399466f*in[14] + -0.599199f*in[15];
 out[15] = -2.21435f*in[9] + 2.85871f*in[11] + -2.85871f*in[13] + -1.47623f*in[14] + -2.21435f*in[15];
 out[16] = -1.12986f*in[16] + -2.11378f*in[18];
 out[17] = -2.13568f*in[19] + 2.13568f*in[21];
 out[18] = -1.12986f*in[16] + -1.46773f*in[17] + -1.05689f*in[18] + -1.61938f*in[19] + 0.140048f*in[21] + -1.05689f*in[22]
           + 0.815406f*in[23] + 0.798932f*in[24];
 out[19] = 3.02031f*in[19];
 out[20] = 0.248196f*in[16] + 1.1008f*in[17] + 1.58533f*in[18] + -2.60553f*in[19] + 1.18164f*in[20] + 1.98914f*in[21]
           + 1.1008f*in[23] + -0.599199f*in[24];
 out[21] = 0.815406f*in[17] + -1.05689f*in[18] + 6.14712f*in[19] + -2.44879f*in[21] + 1.05689f*in[22] + -1.46773f*in[23]
           + 0.798932f*in[24];
 out[22] = 1.59786f*in[16];
 out[23] = 1.99775f*in[17] + 2.26523f*in[19];
 out[24] = -4.53046f*in[19] + 2.26523f*in[21] + 1.99775f*in[23];
}
```

```
// Sparse matrix multiplication for N=6.
void AhatTransposeMultiply(const float * in, float * out) {
 out[4] = 0.915291f*in[4] + -0.647209f*in[5] + 1.58533f*in[6] + 0.268083f*in[7] + -0.915291f*in[8];
 out[5] = -1.29442f*in[5] + -0.454539f*in[7] + -1.83058f*in[8];
 out[6] = 1.94163f*in[5] + -0.804248f*in[7];
 out[7] = -2.1016f*in[7];
 out[8] = 1.83058f*in[4] + 0.990704f*in[7];
 out[9] = 1.67038f*in[10] + 1.33985f*in[12] + -0.691895f*in[14];
 out[10] = 0.847395f*in[9] + 1.05911f*in[10] + 1.09398f*in[11] + 0.453142f*in[13] + -0.351003f*in[15];
 out[11] = -1.7976f*in[10] + 1.7976f*in[14];
 out[12] = -2.54219f*in[10];
 out[13] = 0.599199f*in[9] + 0.691895f*in[10] + -0.773562f*in[11] + 0.32042f*in[13] + -0.691895f*in[14] + -1.44659f*in[15];
 out[14] = -0.573185f*in[10] + -1.6746f*in[13] + 1.29714f*in[15];
 out[15] = 0.599199f*in[9] + 1.751f*in[10] + -0.773562f*in[11] + 1.2267f*in[13] + -0.691895f*in[14] + 0.248196f*in[15];
 out[16] = -1.44659f*in[16] + 0.043814f*in[17] + 1.99785f*in[19] + 1.18164f*in[20] + -66.4551f*in[21] + -1.58533f*in[22]
           + -65.9212f*in[23] + 0.599199f*in[24];
 out[17] = 1.12986f*in[16] + -0.185451f*in[17] + -3.73689f*in[19] + 84.4028f*in[21] + 2.11378f*in[22] + 80.4805f*in[23];
 out[18] = -1.30453f*in[17] + -1.30453f*in[19] + 60.1671f*in[21] + 57.5581f*in[23];
 out[19] = -53.6759f*in[21] + -53.6759f*in[23];
 out[20] = -0.317406f*in[17] + -1.05689f*in[18] + -2.33088f*in[19] + 53.8045f*in[21] + 1.05689f*in[22] + 48.9812f*in[23]
           + -0.798932f*in[24];
 out[21] = 1.59786f*in[16] + 23.856f*in[21] + 23.856f*in[23];
 out[22] = 1.12986f*in[16] + -0.317406f*in[17] + 1.05689f*in[18] + -2.33088f*in[19] + 47.0193f*in[21] + 1.05689f*in[22]
           + 47.2184f*in[23] + -0.798932f*in[24];
 out[23] = 0.758453f*in[17] + -2.00668f*in[19] + -15.3268f*in[21] + -16.575f*in[23];
 out[24] = 44.6846f*in[21] + 40.7741f*in[23];
 out[25] = 0.180665f*in[26] + 0.815982f*in[28] + 1.06883f*in[30] + -0.644318f*in[32] + 0.180665f*in[34];
 out[26] = -1.14263f*in[25] + -0.368734f*in[26] + 0.510998f*in[27] + -2.69405f*in[28] + -1.10388f*in[29] + 0.640177f*in[32]
           + -2.97832f*in[33] + -2.21991f*in[35];
 out[27] = -0.524785f*in[25] + 0.704073f*in[27] + -2.25505f*in[28] + 0.30334f*in[32] + -1.69978f*in[33] + -0.938764f*in[34]
           + -1.26694f*in[35];
 out[28] = -1.87753f*in[26];
 out[29] = -0.890202f*in[26] + -0.722661f*in[27] + 2.30433f*in[28] + 0.780563f*in[29] + 0.780563f*in[31] + -0.701003f*in[32]
           + 2.46732f*in[33] + 1.30039f*in[35];
 out[30] = -0.681331f*in[26] + -2.0098f*in[28] + -2.67061f*in[33] + -1.99055f*in[35];
 out[31] = -0.368734f*in[26] + -0.722661f*in[27] + 0.543848f*in[28] + 0.780563f*in[29] + -0.780563f*in[31] + -0.701003f*in[32]
           + 0.53864f*in[35];
 out[32] = -1.73971f*in[32];
 out[33] = 1.73971f*in[28];
 out[34] = -0.524785f*in[25] + 0.704073f*in[27] + -1.36306f*in[28] + 1.19533f*in[32] + -1.69978f*in[33] + 0.938764f*in[34]
           + -1.26694f*in[35];
 out[35] = 0.890202f*in[26] + -0.611963f*in[28] + -1.10388f*in[31] + -0.722661f*in[33] + -1.30039f*in[35];
}
```

N	Ω
3	$\{(1.5708, 1.5708), (0.9553, -2.3562), (3.1416, 2.3562), (0.9553, 0.7854), (2.1863, 2.3562)\}$
4	$\{(3.1416, 2.6180), (1.5708, -2.6180), (1.5708, 1.5708), (2.0344, -3.1416), (2.0344, -1.5708), (2.0344, -0.5236), (2.0344, 1.5708)\}$
5	$\{(1.5708, 0.7854), (1.1832, 0.0000), (1.5708, -3.1416), (1.1832, 0.7854), (3.1416, 0.0000), (1.5708, 1.5708), (1.5708, 0.3927), (2.2845, -1.5708), (0.8571, -3.1416)\}$
6	$\{(0.0000, 0.0000), (1.5708, 1.5708), (2.1863, 1.5708), (2.1863, -2.7489), (1.5708, -2.3562), (1.5708, -2.7489), (1.5708, -0.7854), (0.6997, 1.5708), (0.6997, -2.3562), (0.9553, 1.5708), (1.5708, 0.0000)\}$
7	$\{(1.5708, 0.7854), (1.0213, -2.6180), (2.1203, -1.5708), (1.5708, -1.5708), (3.1416, 1.5708), (1.5708, 0.5236), (2.1203, 1.5708), (1.8241, 1.5708), (0.5913, -0.3142), (1.8241, -1.5708), (2.1203, -3.1416), (1.5708, 0.3927), (2.3389, -1.5708)\}$
8	$\{(1.5708, -0.5236), (2.0719, 2.6180), (0.6928, 1.5708), (1.5708, -1.5708), (3.1416, -0.3927), (0.6928, -1.5708), (1.7989, -3.1416), (2.0053, 1.5708), (1.8518, -3.1416), (2.0053, -1.5708), (0.6928, -2.3562), (2.2040, -1.5708), (0.8755, 0.0000), (2.2040, 1.5708), (0.6928, 2.6180)\}$

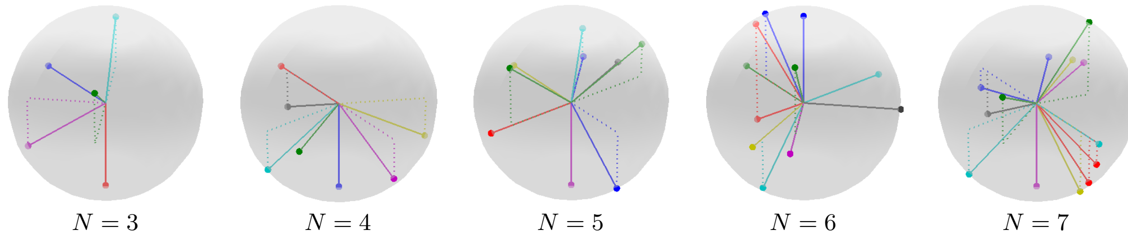

Fig. 2. Sparsity inducing lobe directions up to $N = 8$, and visualizations up to $N = 7$.