

Shearlet Galerkin for transport equations: implementation and stability

E. Fonn

Research Report No. 2011-50
August 2011

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Shearlet Galerkin for transport equations: implementation and stability

E. Fonn

August 8, 2011

Abstract

We provide implementation details for the framework of a solver for the advection-reaction equation using piecewise linear shearlet frames. It is hoped that the directional features of shearlets will provide a sparse representation of the solution under an adaptive setting. We investigate the stability of the resulting stiffness matrices, and find that the effective condition numbers are too high for iterative solvers, even with a modest number of degrees of freedom. This is a strong hint of instability in relevant norms.

1 Introduction

This paper describes an implementation for the framework of an adaptive frame solver for operator equations. Previous research on such methods can be found in [22], and particularly for elliptic operators [9], [8] and [7].

The motivation for this paper comes from the stationary advection-reaction equation (1). The solutions to this equation can exhibit strong anisotropic features, and it is hoped that frames which are designed to capture directional information will be well suited for adaptive methods for solving this equation.

There are several such systems available, most notably *curvelets* ([5], [4]), *countourlets* [10], *directional wavelets* [1] and *shearlets* ([16], [15], [13]), as well as others ([3], [21]). The theory behind these systems comes mostly from the image compression community, where they are used for sparse representation of images and edge detection (such as in [24]), and until now there has been little effort made in using them to solve PDEs.

In this paper we will concern ourselves with compactly supported shearlets and their related ridgelets in two spatial dimensions, whose implementation can be made largely parallel. Compactly supported shearlets were introduced in [14]. They have a particularly nice representation as affine transformations of tensor product functions, and moreover, they can be extended to more than two dimensions. The representations they provide are also provably (almost) optimally sparse [12], although it should be mentioned that this relies on a posteriori information when solving PDEs.

Section 2 introduces the model problem, while section 3 presents the shearlets in detail. In section 4 the main pseudocode is laid out. Finally, some numerical experiments are presented in section 5.

2 Variational problem

We are concerned with the solution of the stationary advection-reaction equation with constant advection,

$$\mathbf{s} \cdot \nabla u(\mathbf{x}) + \kappa(\mathbf{x})u(\mathbf{x}) = f(\mathbf{x}), \quad (1)$$

for $\mathbf{x} \in \Omega \subset \mathbb{R}^2$. Here, \mathbf{s} is the transport direction, $\kappa(\mathbf{x})$ is the reaction coefficient, and $f(\mathbf{x})$ is the source term. This requires *inflow boundary* conditions

$$u(\mathbf{x}) = g(\mathbf{x}), \quad \text{on } \Gamma_- = \{\mathbf{x} \in \partial\Omega \mid \mathbf{s} \cdot \mathbf{n}(\mathbf{x}) < 0\},$$

where $\mathbf{n}(\mathbf{x})$ denotes the outward pointing normal vector for $\mathbf{x} \in \partial\Omega$. Furthermore, in our case, we will always work with $\Omega = [0, 1]^2$.

Solutions to this equation can exhibit strong anisotropic features. Consider the case $\kappa = f = 0$. Since the boundary condition values are transported perfectly along the flow \mathbf{s} , the regularity of g will directly determine the regularity of u , and any discontinuities or irregularities will be pointed in the direction of \mathbf{s} . It makes sense to use this as a test equation for anisotropic frames.

We will discretize the equation using the least squares method [17], resulting in the bilinear form

$$\mathbf{a}(u, v) = (\mathbf{s} \cdot \nabla u + \kappa u, \mathbf{s} \cdot \nabla v + \kappa v)_{L^2(\Omega)}$$

and the linear form

$$\ell(v) = (f, \mathbf{s} \cdot \nabla v + \kappa v)_{L^2(\Omega)}.$$

We can use the offset method to incorporate boundary conditions. Thus, let $\tilde{u} = u + g'$, where \tilde{u} solves (1), u is a function that vanishes on $\partial\Omega$ and g' is an extension of g to Ω . Then, the variational formulation turns into

$$\mathbf{a}(u, v) = \ell(v) - \mathbf{a}(g', v) = \tilde{\ell}(v),$$

where $\tilde{\ell}$ is the linear form ℓ with $f - \mathbf{s} \cdot \nabla g' - \kappa g'$ in place of f .

This reduces our search to functions which are zero on Γ_- . However, this is not a feature our function spaces will generally have. We can enforce this by multiplying a basis function ψ with some given function $z(x)$ which is zero on Γ_- , but nonzero everywhere else, in the spirit of the partition of unity method (see [18] and [2]). In the following analysis, this multiplier function is always included. More on this in section 3.3.

3 Shearlets

Shearlets are one among several function systems designed to efficiently capture anisotropic features in two dimensions. The shearlets are generated by a family of mother shearlets $\hat{\Psi}_m : \mathbb{R}^2 \rightarrow \mathbb{R}$ via affine transformations and translations,

$$\Psi_{\mathbf{A}}(\mathbf{x}) = \left(\mathcal{D}_{\mathbf{A}} \hat{\Psi} \right) (\mathbf{x}) = |\mathbf{A}|^{-1/2} \hat{\Psi}(\mathbf{A}\mathbf{x}), \quad \Psi_{\mathbf{c}}(\mathbf{x}) = \left(\mathcal{T}_{\mathbf{c}} \hat{\Psi} \right) (\mathbf{x}) = \hat{\Psi}(\mathbf{x} - \mathbf{c}),$$

where $\mathbf{A} \in \mathbb{R}^{2 \times 2}$ and $\mathbf{c} \in \mathbb{R}^2$.

3.1 Parameters

Given mother shearlets $\hat{\Psi}_m$, with $1 \leq m \leq M$, the shearlets are defined as

$$\Psi_{m,\mathbf{c},r}^{j,k} = \mathcal{T}_{\mathbf{c}} \mathcal{D}_{\mathbf{A}_r^{j,k}} \hat{\Psi}_m, \quad (2)$$

where

$$\mathbf{A}_r^{j,k} = \begin{pmatrix} 1 & s_x k / s_y \\ & 1 \end{pmatrix} \begin{pmatrix} s_x^j & \\ & s_y^j \end{pmatrix} \begin{pmatrix} & 1 \\ 1 & \end{pmatrix}^r, \quad (3)$$

and $\mathbf{c} \in \mathbb{R}^2$. The leftmost matrix is a *shearing* transformation and the middle matrix is an anisotropic scaling. The values of s_x and s_y can be tweaked to allow for various cases which we shall come back to. Some interesting choices are $(s_x, s_y) \in \{(4, 2), (2, \sqrt{2}), (2, 1)\}$. We will assume that $s_x > s_y \geq 1$ and often also that s_x/s_y is an integer.

The significance of the parameters are as follows.

- m is the mother shearlet index, satisfying $1 \leq m \leq M(j)$, where $M(j) \geq 1$ is the number of generators for level j .
- $\mathbf{c} \in \mathbb{R}^2$ is the translation parameter. If $\hat{\Psi}_m$ is a function with support centered in the origin (as we will see later), $\Psi_{m,\mathbf{c},\cdot}^{j,\cdot}$ is a function with support centered at \mathbf{c} .
- The *scale* or *level* parameter $j \geq 0$ is an integer that scales $\hat{\Psi}_m$ anisotropically. For large j , the functions $\Psi_{\cdot,\cdot,\cdot}^{j,\cdot}$ will have very slender supports, but also smaller. Specifically, if $\hat{\Psi}_m$ has square support, $\Psi_{m,\cdot,\cdot}^{j,0}$ has rectangular support that is $(s_x/s_y)^j$ times longer in the x -direction than in the y -direction.
- The *shear* parameter k is an integer satisfying $-2^j \leq k \leq 2^j$. Its purpose is to tilt the support of $\Psi_{\cdot,\cdot,\cdot}^{j,k}$ to a certain direction. The maximal tilt is achieved for $k = \pm 2^j$, which is at an angle of $\pi/4$ off the original direction (where $k = 0$). Note also that with increasing level, the resolution in directions also increases two-fold, and that if $\hat{\Psi}_m$ has support centered at the origin, then $\Psi_{m,\mathbf{c},\cdot}^{j,k}$ has support centered at \mathbf{c} for all k .
- Finally the *cone* parameter $r \in \{0, 1\}$ mirrors the axes if $r = 1$, so that we can generate shearlets pointing in all directions. The principal directions for $r = 0$ are all in the up-down “cone”, and for $r = 1$ they are in the left-right “cone”.

We offer some illustrations to make these concepts clear in figure 1.

3.2 The mother shearlet

The mother shearlets $\hat{\Psi}_m$ are usually chosen as tensor products $\hat{\Psi}_m(\mathbf{x}) = \psi_m(x_1)\varphi_m(x_2)$, where ψ_m are wavelets and φ_m are scaling functions. These can be chosen to be, say, piecewise linear, so that $\hat{\Psi}$ is piecewise bilinear on a rectangular mesh. Other choices are possible, but for our purposes we will restrict ourselves to linear ψ and φ , whose supports are (subsets of) the interval $[-1/2, 1/2]$, so that $\hat{\Psi}_m$ is supported in the square $[-1/2, 1/2]^2$, which we will consider to be our “reference” element in the sense of section 4.2.

For scaling functions, we will always use $\varphi(x) = 1 - 2|x|$ for $|x| < 1/2$, which is simply a hat function.

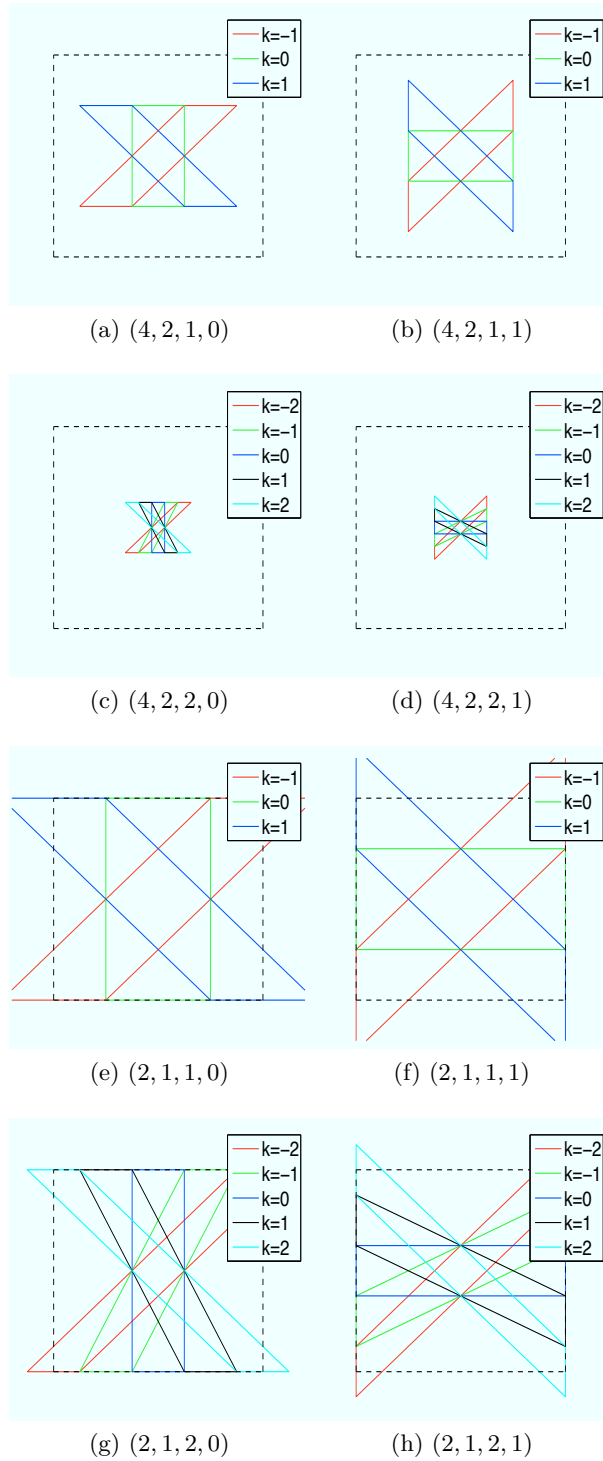


Figure 1: The support of some shearlets are shown in outlines of various colors. The blue box is always the square $[0, 1]^2$. The figures show all the possible shears k superimposed over one another. The caption to each figure gives the values of s_x , s_y , j and r , in that order.

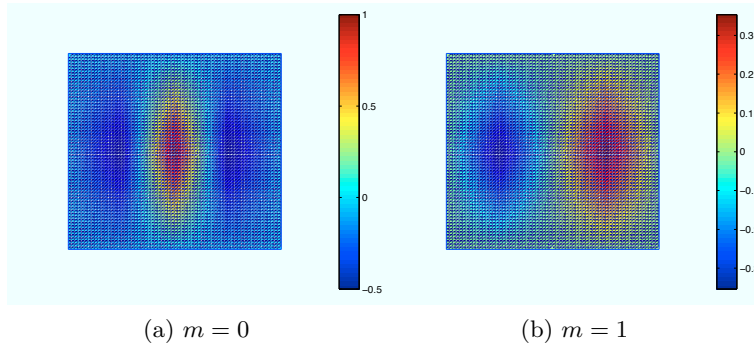


Figure 2: Two mother shearlets.

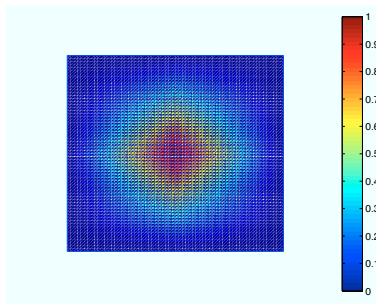


Figure 3: A level 0 mother shearlet.

We (usually) use two different wavelets. They are

$$\psi_1(x) = \varphi(2x) - \frac{1}{2}\varphi(2x-1) + \frac{1}{2}\varphi(2x+1), \quad \psi_2(x) = \frac{1}{2\sqrt{2}}\varphi(2x-1) - \frac{1}{2\sqrt{2}}\varphi(2x+1),$$

with φ as previously defined (having a support of length 1).

This yields the two mother shearlets shown in figure 2.

This is valid for $j > 0$. For the first level, we use a single bilinear hat function as mother shearlet, namely $\psi = \varphi$. Without this, we cannot capture the lower frequencies of a signal. This is shown in figure 3.

Of course, there is nothing in our construction that prohibits mother shearlets from also depending on j .

3.3 Boundary conditions

Mathematically, there are several different kinds of boundary conditions worth investigating.

As far as the inflow boundary conditions are concerned, we mentioned in section 2 that we wish to use an offset function. This again requires that our basis functions vanish on the inflow boundary, which, by design, the shearlets do not.

Thus, in the spirit of the partition of unity method ([18] and [2]), we multiply all basis functions with a global function $z(\mathbf{x})$ which is zero on the inflow boundary and

nonzero everywhere else. It is in our utmost interest to keep z as simple as possible, both to preserve shearlet qualities and not to impose unreasonable conditions on quadrature rules.

If the inflow boundary consists only of entire edges of the domain, z can be polynomial. If, for example $s(\mathbf{x}) = (1, 1)^T$, the inflow boundary consists of the edges along the first and second axes, and so $z(\mathbf{x}) = x_1x_2$ is a natural choice.

Another possibility is periodic boundary conditions. By construction, the shearlets can be periodically extended and remain well-defined. The periodic case requires some special care in the implementation, which will be dealt with where applicable in section 4.

3.4 Translations

It remains to specify the translation parameter \mathbf{c} . Assume that the fundamental wavelets and scaling functions ψ . and φ . are piecewise linear with exactly W and S equally sized pieces respectively. Now fix j and let $k = 0$. Then, these shearlets are piecewise linear in both directions on a mesh with resolution of size s_x^{-j}/W and s_y^{-j}/S respectively. As long as s_x and s_y are integers, this mesh matches perfectly with the boundaries of the domain.

This indicates that we should choose translation steps spaced by s_x^{-j}/W in the first direction, and s_y^{-j}/S in the second, whence the translation parameter attains the values

$$\mathbf{c} \in \left\{ \begin{pmatrix} L(j) + \alpha/s_x^j W \\ D(j) + \beta/s_y^j S \end{pmatrix} \mid \alpha \in \{0, \dots, R(j)\}, \beta \in \{0, \dots, U(j)\} \right\}. \quad (4)$$

The exact values of the positions for the “bottom left” shearlet in $(L(j), D(j))$ and the maximal necessary displacement in the left-right direction $R(j)$ and the up-down direction $U(j)$ will depend on the boundary conditions. Assuming inflow or no boundary conditions and s_x/s_y integral, we find that

$$L(j) = \frac{1}{s_x^j W} - \frac{1}{2} \left(\frac{1}{s_x^j} + \frac{1}{s_y^j} \right), \quad R(j) = s_x^j W (1 - 2L(j)) + 1$$

captures all translations where at least *one* shear intersects the domain. The up-down direction is easier, because the shearing does not affect it:

$$D(j) = 0, \quad U(j) = s_y^j S.$$

We offer some illustrations in figures 4 and 5 for the case $W = S = 4$ and $s_x = 4$, $s_y = 2$. For the remainder of this discussion, we will refer to a translation point c as parametrized by α and β according to (4).

Note that W and S are numbers that depend on the mother shearlets $\hat{\Psi}_m$. We have allowed for mother shearlets to depend on j , and thus, we must consider that W and S are also functions of j .

Some care will be required if the wavelets and scaling functions ψ . and φ . have different numbers of polynomial subparts (i.e. W and S depend on m). Our construction do not allow coupling between the translation \mathbf{c} and m . In this case, it would be most appropriate to let $W = \text{lcm} \{W_m\}$ and $S = \text{lcm} \{S_m\}$ (least common multiple), if these numbers are not prohibitively large.

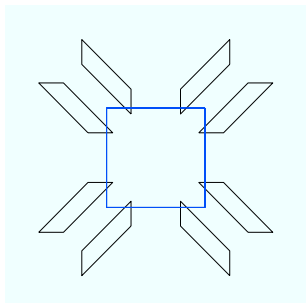


Figure 4: The extreme shearlets for $j = 1$, $s_x = 4$, $s_y = 2$.

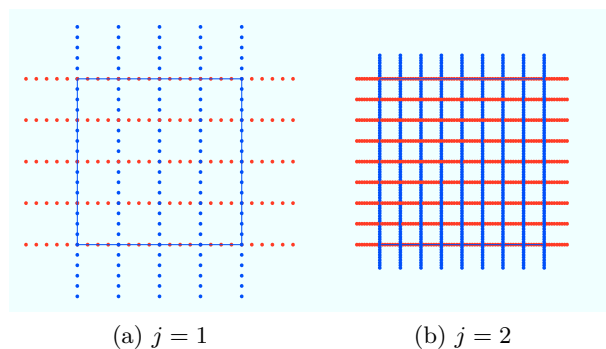


Figure 5: Translation points for the first two levels for $s_x = 4$, $s_y = 2$, $W = S = 4$. The blue points are for $r = 0$ and the red points are for $r = 1$.

j	$N(j)$	j	$N(j)$
1	$8.10 \cdot 10^2 \cdot M(1)$	1	$3.42 \cdot 10^2 \cdot M(1)$
2	$7.47 \cdot 10^3 \cdot M(2)$	2	$1.05 \cdot 10^3 \cdot M(2)$
3	$8.90 \cdot 10^4 \cdot M(3)$	3	$3.62 \cdot 10^3 \cdot M(3)$
4	$1.22 \cdot 10^6 \cdot M(4)$	4	$1.34 \cdot 10^4 \cdot M(4)$
5	$1.81 \cdot 10^7 \cdot M(5)$	5	$5.13 \cdot 10^4 \cdot M(5)$
6	$2.79 \cdot 10^8 \cdot M(6)$	6	$2.01 \cdot 10^5 \cdot M(6)$

Table 1: Number of degrees of freedom per level. $(s_x, s_y) = (4, 2)$ on the left and $(s_x, s_y) = (2, 1)$ on the right. In both cases, $S(j) = 2$ and $W(j) = 4$.

3.5 Number of degrees of freedom

The exact formula for the number of shearlets on level $j \geq 0$ is (assuming s_x/s_y is integral):

$$N(j) = 2M(j) (2^{j+1} + 1) (s_y^j S(j) + 1) ((1 + s_x^j + (s_x/s_y)^j) W(j) - 1),$$

where $M(j)$ is the number of mother shearlets at level j , and $S(j)$ and $W(j)$ have the meaning given in section 3.4.

This leads to an asymptotic growth rate

$$N(j) \sim (2s_x s_y)^j \cdot M(j) \cdot W(j) \cdot S(j),$$

and with the reasonable assumptions that M , W and S are bounded from above as $j \rightarrow \infty$, we have

$$N(j) \sim (2s_x s_y)^j.$$

Still, this can be a very rapid growth rate. For the case $s_x = 4$, $s_y = 2$ we have $N(j) \sim 16^j$, and for the more modest choice $s_x = 2$, $s_y = 1$, we have $N(j) \sim 4^j$. The exact numbers for levels $j = 1, \dots, 6$ for these two cases are listed in table 1 (with $S(j) = 2$, $W(j) = 4$).

The hope is that the shearlets lend themselves particularly well to an adaptive approach, which will allow us to avoid this problem altogether.

4 Algorithms

4.1 Numbering

The shearlets are numbered according to a hierarchy of the parameters, as such:

1. m is the least significant parameter. Thus, if $\Psi_{m,\mathbf{c},r}^{j,k}$ is shearlet number N , then $\Psi_{m+1,\mathbf{c},r}^{j,k}$ is shearlet number $N + 1$.
2. k comes next. Let M be the total number of mother shearlets. Then, if $\Psi_{m,\mathbf{c},r}^{j,k}$ is shearlet number N , then $\Psi_{m,\mathbf{c},r}^{j,k+1}$ is shearlet number $N + M$.
3. Next is the translation parameter \mathbf{c} , with x -axis before y -axis. If there are K shears on level j , then if shearlet $\Psi_{m,\mathbf{c},r}^{j,k}$ is shearlet number N , then the corresponding shearlet immediately to the right is shearlet number $N + MK$, and the corresponding shearlet immediately above it is number $N + MKR$, where R is the total number of translation points in the left-right direction.

4. Just before last is the cone parameter r .
5. As the only open-ended parameter j is naturally last.

Algorithm 1 computes the parameters given a shearlet number, and algorithm 2 performs the opposite task. These rely on knowing the values of $M(j)$ and $K(j)$ (there is no generality lost in allowing M to vary by level), $R(j)$ and so on, which we have assumed are available in the functions NumMothers, NumShears, NumRight and NumUp. The function NumAtLevel counts the total number of shearlets at a level, and is merely a convenient shorthand. Note that the shearlet numbers use 1-base indexing, while the parameters are use 0-based indexing.

$$\text{NumAtLevel}(j) = 2 \cdot \text{NumMothers}(j) \cdot \text{NumShears}(j) \cdot \text{NumRight}(j) \cdot \text{NumUp}(j).$$

Algorithm 1 Computes $j, r, \mathbf{c} = \mathbf{c}(\alpha, \beta), k$ and m from a given shearlet number.

Require: Shearlet number n

```

1:  $j \leftarrow 0$ 
2: while  $n > \text{NumAtLevel}(j)$  do
3:    $n \leftarrow n - \text{NumAtLevel}(j)$ 
4:    $j \leftarrow j + 1$ 
5: end while
6:  $r \leftarrow 0$ 
7: if  $n > \text{NumAtLevel}(j)/2$  then
8:    $r \leftarrow 1$ 
9:    $n \leftarrow n/2$ 
10: end if
11:  $m \leftarrow (n - 1) \bmod \text{NumMothers}(j)$ 
12:  $n \leftarrow \lceil n/\text{NumMothers}(j) \rceil$ 
13:  $k \leftarrow ((n - 1) \bmod \text{NumShears}(j)) - (\text{NumShears}(j) - 1)/2$ 
14:  $n \leftarrow \lceil n/\text{NumShears}(j) \rceil$ 
15:  $\alpha \leftarrow (n - 1) \bmod \text{NumRight}(j)$ 
16:  $n \leftarrow \lceil n/\text{NumRight}(j) \rceil$ 
17:  $\beta \leftarrow n - 1$ 
18: return  $j, r, \mathbf{c} = \mathbf{c}(\alpha, \beta), k, m$ 

```

Both algorithms are useful, and can be used to implement functions to manipulate shearlets such as StepRight, StepUp and Shear, details of which we leave out.

In the following, we will describe shearlets in algorithms as a single variable (usually s or t), which can be assumed to work as a structure, and that subfields can be accessed by dot-notation such as $s.j$. We will also speak of s as the shearlet itself (i.e. the mathematical function). The meaning should in either case be clear from the context.

4.2 Transformation

For our purposes we need, at least at some point, be able to evaluate shearlets at arbitrary points in \mathbb{R}^2 . This is done using the transformation technique in (2)-(3).

Algorithm 2 Computes the shearlet number given the other parameters.

Require: $j, r, \mathbf{c} = \mathbf{c}(\alpha, \beta), k$ and m

```

1:  $n \leftarrow 0$ 
2: for  $i \leftarrow 0, \dots, j - 1$  do
3:    $n \leftarrow n + \text{NumAtLevel}(i)$ 
4: end for
5: if  $r = 1$  then
6:    $n \leftarrow n + \text{NumMothers}(j) \cdot \text{NumShears}(j) \cdot \text{NumRight}(j) \cdot \text{NumUp}(j)$ 
7: end if
8:  $n \leftarrow n + \beta \cdot \text{NumMothers}(j) \cdot \text{NumShears}(j) \cdot \text{NumRight}(j)$ 
9:  $n \leftarrow n + \alpha \cdot \text{NumMothers}(j) \cdot \text{NumShears}(j)$ 
10:  $n \leftarrow n + (k + (\text{NumShears}(j) - 1)/2) \cdot \text{NumMothers}(j)$ 
11:  $n \leftarrow n + m + 1$ 
12: return  $n$ 

```

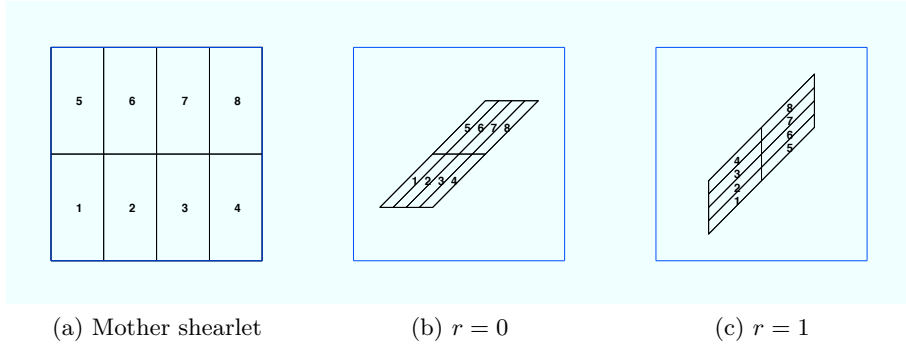


Figure 6: An example of subdomain numbering, where $W = 4$ and $S = 2$. The resulting shearlets are piecewise polynomial on $WS = 8$ sections.

A function called BaseTransform takes a shearlet and produces the transformation matrix \mathbf{A} and the translation vector \mathbf{c} .

$$(\mathbf{A}, \mathbf{c}) \leftarrow \text{BaseTransform}(s).$$

The mapping $\mathbf{x} \rightarrow \mathbf{A}(\mathbf{x} - \mathbf{c})$ is an affine transformation mapping the square $[-1/2, 1/2]^2$ to $\text{supp } s$.

The reason for the name BaseTransform is that it will be useful to have a more general function called Transform that can produce mappings into subdomains of $\text{supp } s$. Recall that s is defined as (an affine transformation of) a tensor product of two piecewise linear functions, so that s is piecewise polynomial with maximal degree 2. By *subdomain* we mean the “pieces” on which s is polynomial. A numbering of subdomains is shown in figure 6.

Thus we define a function Transform(s, i) which takes a subdomain number as a second optional argument. If i is not specified or $i = 0$, it is assumed that BaseTransform(s) is meant. See algorithm 3.

Algorithm 3 Transform computes \mathbf{A} and \mathbf{c} for a given shearlet and subdomain number.

Require: Shearlet s and optional subdomain number $0 \leq i \leq 8$

- 1: $(\mathbf{A}, \mathbf{c}) \leftarrow \text{BaseTransform}(s)$
 - 2: **if** i is given and $i > 0$ **then**
 - 3: $\mathbf{d} \leftarrow \left(\frac{1}{W} \left((i - 1 \bmod W) + \frac{1}{2} \right) - \frac{1}{2}, \frac{1}{S} \left(\lfloor \frac{i-1}{W} \rfloor + \frac{1}{2} \right) - \frac{1}{2} \right)^T$
 - 4: $\mathbf{c} \leftarrow \mathbf{A}^{-1} \mathbf{d} + \mathbf{c}$
 - 5: $\mathbf{A} \leftarrow \begin{pmatrix} W & \\ & S \end{pmatrix} \cdot \mathbf{A}$
 - 6: **end if**
 - 7: **return** (\mathbf{A}, \mathbf{c})
-

In the following, we will often be concerned with the polygon defined by the support of a given shearlet, or one of its subdomains. We will consider polygons as a $2 \times n$ matrix of vertices $(\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n)$ numbered counterclockwise, where $\mathbf{p}_n \neq \mathbf{p}_1$. A useful convenience is the function `Corners`, which computes these vertices by transforming the corners of the reference square $[-1/2, 1/2]^2$. See algorithm 4. If no shearlet is given, it returns the corners of the computational domain.

Algorithm 4 Corners computes the vertices of a shearlet support or subdomain.

Require: Optional shearlet s and optional subdomain number $0 \leq i \leq 8$

- 1: **if** s is not given **then**
 - 2: **return** $\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$
 - 3: **end if**
 - 4: $(A, \mathbf{c}) \leftarrow \text{Transform}(s, i)$
 - 5: $\mathbf{p} \leftarrow \frac{1}{2} \begin{pmatrix} -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix}$
 - 6: $\mathbf{p} \leftarrow \mathbf{A}^{-1} \mathbf{p} + \mathbf{c}$
 - 7: **if** $s.r = 1$ **then**
 - 8: Reverse the order of \mathbf{p}
 - 9: **end if**
 - 10: **return** \mathbf{p}
-

4.3 Assembly strategy

4.3.1 Stiffness matrix

Now, given a bilinear form $\mathbf{a}(\cdot, \cdot)$ we want to compute the corresponding stiffness matrix for a given index set I of shearlets. In an ordinary hierarchical setting (i.e. with hat functions), one would rely on a *refinement relation* that expresses a function as a linear combination of functions on the next level. This would allow us to compute the stiffness matrix for the highest level only, and then use the refinement

relation to either fill out the coarser matrices or to allow fast ($O(N)$) matrix-vector multiplication. In the shearlet case, there are several problems with this.

1. The number of shearlets on each level increases very fast, asymptotically (in our worst case) 16^j , compared to the more conventional 4^j . Thus, the saved effort from not having to directly calculate the coarse levels becomes almost negligible.
2. In a more traditional setting, such as with triangular elements and hat functions, it is possible to assemble the matrix in time that scales linearly with the number of elements, as this circumvents the necessity of having to check if the supports of two functions overlap. No such shortcut is readily available to us in the shearlet case.¹
3. The primary motivation for implementing a shearlet basis/frame is to employ adaptive methods, which do not easily lend themselves to such a strategy.

Thus, for the time being, we fall back to the primitive loop in algorithm 5. Here, we have assumed a symmetric \mathbf{a} . This is also where parallelization should enter the picture.

Algorithm 5 Assembly loop.

Require: Shearlet index set I .

- 1: **for** $i, j \in I$ **do**
 - 2: $\mathbf{A}(i, j) \leftarrow \mathbf{a}(s_j, s_i)$
 - 3: $\mathbf{A}(j, i) \leftarrow \mathbf{a}(s_i, s_j)$
 - 4: **end for**
-

How, then, do we go about computing $\mathbf{a}(s_i, s_j)$? The most obvious approach is straightforward.

1. Identify the corners of the supports of s_i and s_j using Corners.
2. Check if the supports intersect. If they do not, return 0.
3. If the supports intersect, loop over pairs (m, n) of subdomains of supports of s_i and s_j . Compute the intersection of this pair if it is nonempty.
4. This results in a set of nonempty disjoint polygons P_k whose union is the intersection of supports of s_i and s_j , which has the property that both s_i and s_j are polynomial on any P_k .
5. Further divide each P_k into a set of triangles. On each triangle, form a quadrature rule. Take the union of these quadrature rules as a quadrature rule for the intersection of the supports of s_i and s_j .

Once the quadrature rule is available, the integrand of the bilinear form \mathbf{a} can be evaluated on the quadrature points and summed up accordingly.

¹It is true that there are simple methods that can identify a number of negatives (no intersection), such as checking the center and diameter of shearlet supports, or cheap intersection checking routines. However, these cases still have to be *tested*, and moreover, the shearlet matrix should nevertheless not be sparse.

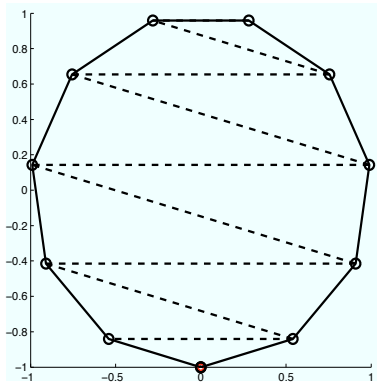


Figure 7: Splitting a polygon (here a regular 11-gon) into triangles.

There should be a basic choice of quadrature rule defined on a reference triangle, which can then be transformed. This part is essentially identical to triangular Lagrangian FEM. Of course, the basic quadrature rule has to be of order high enough to resolve the variation in s_i and s_j if there are no variable coefficients. If there *are* variable coefficients, the case becomes a little more complicated, which we discuss in section 4.3.4.

This strategy relies heavily on reliable intersection checking and computing routines. Initial experiments suggest that most of the processing time is spent doing operations like these, and care should be taken to implement these properly (see section 4.3.3).

Thus, we now assume the existence of these functions:

1. A function `CheckIntersection(s, t)` for checking if the supports of shearlets s and t intersect at all.
2. A function `ComputeIntersection(p, q)` for computing intersections between two convex polygons p and q .
3. A function `SplitPolygon(p)` for reducing a polygon p to a collection of triangles.
4. A function `TransformQuadrule(q, p)` for transforming a quadrature rule q defined on a reference triangle onto the triangle p .

The algorithms for checking and computing intersections are discussed further in section 4.3.3. The `SplitPolygon` function can be implemented in a myriad of different ways, one of which is shown in algorithm 6. This algorithm splits a polygon as shown in figure 7, where the red vertex is vertex number 1. The algorithm alternates between adding a triangle on the “right” or “left” side, and keeps track of the next unused vertex on both sides. No new vertices are introduced.

Algorithm 7 contains pseudocode for the `BuildDoubleQuadrule` function, which builds a quadrature rule for the intersection of two shearlet supports. and algorithm 8 wraps this code to a function `EvalBLF` for evaluating bilinear forms.²

²Note that the application of the quadrature rule in algorithm 8 should be vectorized for faster implementation. The pseudocode is merely conceptual.

Algorithm 6 SplitPolygon divides a polygon p into a collection of triangles.

Require: $2 \times N$ matrix \mathbf{p} denoting a polygon

```
1: start  $\leftarrow$  3, end  $\leftarrow$  1, right  $\leftarrow$  false,  $T \leftarrow \left\{ \left[ \mathbf{p}_{\cdot,1} \ \mathbf{p}_{\cdot,2} \ \mathbf{p}_{\cdot,N} \right] \right\}$ 
2: while start + end  $\leq N$  do
3:   if right then
4:      $T \leftarrow T \cup \left\{ \left[ \mathbf{p}_{\cdot,\text{start}-1} \ \mathbf{p}_{\cdot,\text{start}} \ \mathbf{p}_{\cdot,N-\text{end}+1} \right] \right\}$ 
5:     right  $\leftarrow$  true
6:     end  $\leftarrow$  end + 1
7:   else
8:      $T \leftarrow T \cup \left\{ \left[ \mathbf{p}_{\cdot,\text{start}-1} \ \mathbf{p}_{\cdot,N-\text{end}} \ \mathbf{p}_{\cdot,N-\text{end}+1} \right] \right\}$ 
9:     right  $\leftarrow$  true
10:    end  $\leftarrow$  end + 1
11:   end if
12: end while
13: return  $T$ 
```

Algorithm 7 BuildDoubleQuadruple constructs a quadrature rule for the intersection of the supports of two shearlets s and t .

Require: Shearlets s , t and basic quadrature rule q

```
1:  $r \leftarrow \emptyset$ 
2: for  $i, j \in \{1, \dots, 8\}$  do
3:    $\mathbf{p} \leftarrow \text{Corners}(s, i)$ 
4:    $\mathbf{q} \leftarrow \text{Corners}(t, j)$ 
5:    $B \leftarrow \text{SplitPolygon}(\text{ComputeIntersection}(p, q))$ 
6:   for  $\mathbf{c} \in B$  do
7:      $r \leftarrow r \cup \{ \text{TransformQuadruple}(q, \mathbf{c}) \}$ 
8:   end for
9: end for
```

Algorithm 8 EvalBLF evaluates a bilinear form.

Require: Shearlets s , t , basic quadrature rule q and a function $f(s, t, x)$ for evaluating the integrand of a bilinear form at a point x .

```
1: if CheckIntersection( $s, t$ ) then
2:    $q \leftarrow \text{BuildDoubleQuadruple}(s, t, q)$ 
3:    $r \leftarrow 0$ 
4:   for quadrature point  $\mathbf{x}$  with weight  $w \in q$  do
5:      $r \leftarrow r + w \cdot f(s, t, x)$ 
6:   end for
7:   return  $r$ 
8: else
9:   return 0
10: end if
```

4.3.2 Load vector

For evaluating the right hand side load vector $\ell(\cdot)$ the strategy is mostly the same, except we can ignore most of the intersection checking. The functions `BuildSingleQuadrule` and `EvalLF` mirror `BuildDoubleQuadrule` and `EvalBLF` and their pseudocode is presented in algorithms 9 and 10.³

Algorithm 9 `BuildSingleQuadrule` constructs a quadrature rule for the support of a shearlet s .

Require: Shearlet s and basic quadrature rule q

```
1:  $r \leftarrow \emptyset$ 
2: for  $i \in \{1, \dots, 8\}$  do
3:    $\mathbf{p} \leftarrow \text{Corners}(s, i)$ 
4:    $B \leftarrow \text{SplitPolygon}(p)$ 
5:   for  $\mathbf{c} \in B$  do
6:      $r \leftarrow r \cup \text{TransformQuadrule}(q, \mathbf{c})$ 
7:   end for
8: end for
```

Algorithm 10 `EvalLF` evaluates a linear form.

Require: Shearlet s , basic quadrature rule q and a function $f(s, \mathbf{x})$ for evaluating the integrand of a bilinear form at a point x .

```
1:  $q \leftarrow \text{BuildSingleQuadrule}(s, q)$ 
2:  $r \leftarrow 0$ 
3: for quadrature point  $\mathbf{x}$  with weight  $w \in q$  do
4:    $r \leftarrow r + w \cdot f(s, \mathbf{x})$ 
5: end for
6: return  $r$ 
```

4.3.3 Intersections

Our method relies on computing intersections between many different polygons, or deciding if polygons are disjoint. There are several algorithms for computing and detecting intersections, see for example [23], [20] and [6]. These all assume convexity, which is valid in our case.

For checking disjointness, we rely on the *separating axis theorem*, which is commonly used in game programming for collision detection.

Theorem 1 (Separating Axis Theorem (SAT)). *Let A and B be convex sets in \mathbb{R}^2 . Then $A \cap B = \emptyset$ if and only if there exists a line that separates A from B . Moreover, if A and B are polygons with empty intersection, such a separating line can always be found among the prolongations of the edges of A and B .*

³Again, the application of the quadrature rule should be vectorized.

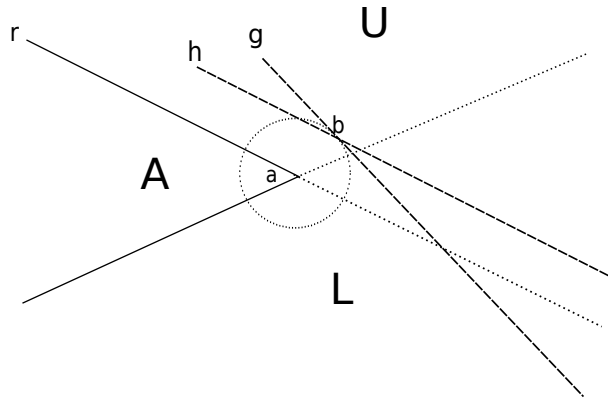


Figure 8: The separating axis theorem.

Proof. The first statement is obvious. Now, assume that A and B are disjoint convex polygons, and let \mathbf{a} , \mathbf{b} be points of minimal distance between them. If either \mathbf{a} or \mathbf{b} is not a vertex, they are points on edges, and the prolongation of either of these edges yield a separating line.

If both \mathbf{a} and \mathbf{b} are vertices, consider vertex \mathbf{a} and the prolongation of its edges dividing the space outside of A into three regions as shown in figure 8, two of which are named U and L . Assume vertex \mathbf{b} is not in L . If none of B intersects L , the edge r will be a separating line. Otherwise, consider the edge from \mathbf{b} in direction of L . It must lie in the wedge between lines g and h . If it lies under g , \mathbf{b} will no longer be closest to A , and if it lies over h , B will not reach the region L . No matter which, this edge will be a separating line.

If \mathbf{b} is in L , it is not in U , and the argument works correspondingly. \square

The consequence of the SAT is that we can decide if two polygons p and q are disjoint by looping through the edges of one polygon (say p). If the vertices are ordered counterclockwise, and we assume that an edge e therefore has a counterclockwise “direction”, we already know that p lies on the left side of e , so we need only check if q lies entirely on the right, which can be done by checking the vertices of q (again, because of convexity). At this point, if no separating line has been found, one has to loop through the edges of q in the similar fashion.

This is expressed in algorithm 11. Note that it does not actually check if they intersect *in the computational domain*. However, some few false positives are not worrying.

For actually *computing* the intersection between two polygons we have employed the General Polygon Clipper (GPC) library from the University of Manchester [19].

In the case of periodic boundary conditions, it is clear that this does not suffice. Two shearlets that do not intersect in the nonperiodic case can nevertheless intersect if they are periodically extended. For checking the intersection, it is necessary to translate the domain of *one* of the shearlets by (m, n) , $m, n \in \{-1, 0, 1\}$. This range of m, n can be restricted somewhat if we take into account where the shearlets are located with respect to one another, and also their diameters. For the smallest shearlets, it will be possible to reduce the number of translations to be checked to one. Also note that in this case, there are no false positives, as *every* possible point

Algorithm 11 CheckIntersection checks if the supports of two shearlets s and t are disjoint or not. Returns true if the supports intersect.

Require: Shearlets s and t

```
1:  $\mathbf{p} \leftarrow \text{Corners}(s)$ ,  $\mathbf{q} \leftarrow \text{Corners}(t)$ 
2: for  $e \in \text{Edges}(\mathbf{p})$  do
3:    $s \leftarrow \text{true}$ 
4:   for  $\mathbf{v} \in \text{Vertices}(\mathbf{q})$  do
5:     if  $\mathbf{v}$  is on the left side of  $e$  then
6:        $s \leftarrow \text{false}$ 
7:       break
8:     end if
9:   end for
10:  if  $s$  then
11:    return false
12:  end if
13: end for
14: Repeat loop from line 2 with  $p$  switched with  $q$ 
15: return true
```

of intersection is “in” the domain.

For computing the intersection in the periodic case, we must construct unions of polygons. GPC can handle this. Given the support of a shearlet, we intersect it, and all its relevant translations by (m, n) , with the computational domain. This gives us a union of polygons in $[0, 1]^2$ representing the support of that shearlet. Two such unions can be intersected with GPC and the result is also a union of polygons.

4.3.4 Error in quadrature

If the bilinear form \mathbf{a} does not contain variable coefficients, it will normally be possible to evaluate the stiffness matrix exactly (up to machine precision) by choosing a basic quadrature rule which has sufficiently high order. This will also be possible for the right hand side vector if the function f is simple enough. In most interesting computations, however, these functions will be general, and so we should consider the effect of this.

In a traditional setting with hat functions, and the strategy involving the refinement relation we discussed in section 4.3, the largest “basic unit” on which we form quadrature rules are triangles on the finest level. The size of these approach zero as the levels increase, and so, for smooth integrands, the quadrature error will automatically approach zero as we refine the finite element space.

In our case, we apply “raw” quadrature to every element, the coarse shearlets included, and so the quadrature error will not automatically vanish as we refine our space. In this case it will be necessary to modify BuildDoubleQuadrule and BuildSingleQuadrule to further subdivide the triangles before transforming the basic quadrature rule onto them.

To make quadrature errors on two different levels correspond, we must subdivide the polygons on the coarser level until they correspond to the dimensions of the

polygons on the finer level. Let $s = \min\{s_x, s_y\}$. Thus, on level j , shearlets extend a maximal linear distance on the order of s^{-j} , and after l_j subdivisions, this will be $s^{-j} \cdot 2^{-l_j}$. For two levels i and j to correspond, we need

$$s^j \cdot 2^{l_j} = s^i \cdot 2^{l_i},$$

or

$$l_j = l_i + (i - j) \log_2(s).$$

When integrating the intersection between two shearlets s and t it will be prudent to use $j = \max(s.j, t.j)$.

With piecewise linear shearlets, a quadrature rule that can integrate quadratics exactly is sufficient to capture the constant coefficient case, and for triangles, this quadrature rule is the equal-weight midpoint quadrature.

4.3.5 Evaluating shearlets and their gradients

Of course, at the end of the day, we need a function for evaluating a shearlet (or its gradient) at a given point. We transform a point \mathbf{x} to the square $[-1/2, 1/2]^2$ using `BaseTransform`, whereupon we can use simple linear interpolation to compute the value of a shearlet at that point (recall that *mother* shearlets are defined as tensor products of piecewise linear functions).

For the gradient, this becomes slightly more complicated. First, we note that

$$\hat{\Psi}_m(x_1, x_2) = \psi(x_1)\varphi(x_2) \implies \nabla \hat{\Psi}_m(x_1, x_2) = [\psi'(x_1)\varphi(x_2) \quad \psi(x_1)\varphi'(x_2)]^T,$$

so our problem is reduced to computing the derivative of a piecewise linear function in one dimension. After that, the gradient of $\hat{\Psi}_m$ can be assembled, and transformed back to “real” space using the matrix \mathbf{A} from `BaseTransform`.

If $\psi(x)$ (say) is given by its values ψ_i at points x_i ordered in increasing order, we can compute the derivative at some point x by first identifying j such that $x_j \leq x < x_{j+1}$. Then

$$\psi'(x) = \frac{\psi_{j+1} - \psi_j}{x_{j+1} - x_j}.$$

If no such j exists, we can set the derivative to zero (assuming that ψ is defined by constant extrapolation).

This approach can be vectorized in (say) MATLAB with some intricacies. An example can be found in section A.3.

Again, some modifications are necessary in the periodic case. Before transformation, a point \mathbf{x} can be translated to any other point on a grid $\{\mathbf{x} + m(1, 0) + n(0, 1)\}$, and if any of these translates land in the domain of a shearlet, the value of the shearlet at that point must be taken.

To do this, we transform also the vectors $(1, 0)$ and $(0, 1)$ to the reference square, yielding $\boldsymbol{\eta}$ and $\boldsymbol{\xi}$. Now, the problem is reduced to finding a translate $\mathbf{x} + m\boldsymbol{\xi} + n\boldsymbol{\eta}$ in $[-1/2, 1/2]^2$. The key observation is that due to the nature of the transformation matrix \mathbf{A} (see section 3.1), either η_2 or ξ_2 will be zero (depending on the cone parameter r). Assume without loss of generality that $\xi_2 = 0$. Thus, only m can affect the second coordinate of the translate, and so we can immediately solve for m (or determine that such an m does not exist). With m fixed, solving for n is equally simple by considering the equation in the first coordinate. See algorithm 12

This algorithm can also be vectorized.

Algorithm 12 PreparePoints transforms a point \mathbf{x} to a point \mathbf{y} in the reference square (in case of periodicity, if such a translate can be found).

Require: Shearlet s , point \mathbf{x}

```

1:  $(\mathbf{A}, \mathbf{c}) \leftarrow \text{BaseTransform}(s)$ 
2:  $\mathbf{y} \leftarrow \mathbf{A}(\mathbf{x} - \mathbf{c})$ 
3: if periodic then
4:    $\boldsymbol{\eta} \leftarrow \mathbf{A}(\cdot, 1), \boldsymbol{\xi} \leftarrow \mathbf{A}(\cdot, 2)$ 
5:   if  $\xi_2 \neq 0$  then
6:     swap  $\boldsymbol{\eta}$  and  $\boldsymbol{\xi}$ 
7:   end if
8:    $n_{\min} \leftarrow \lceil -(\mathbf{x}_2 + 1/2)/\eta_2 \rceil, n_{\max} \leftarrow \lfloor -(\mathbf{x}_2 - 1/2)/\eta_2 \rfloor, n \leftarrow n_{\min}$ 
9:    $m_{\min} \leftarrow \lceil -(\mathbf{x}_2 + n\eta_1 + 1/2)/\xi_1 \rceil, m_{\max} \leftarrow \lfloor -(\mathbf{x}_2 + n\eta_1 - 1/2)/\xi_1 \rfloor, m \leftarrow m_{\min}$ 
10:  return  $\mathbf{y} + m\boldsymbol{\xi} + n\boldsymbol{\eta}$ 
11:  (will be outside  $[-1/2, 1/2]^2$  if  $n_{\min} > n_{\max}$  or  $m_{\min} > m_{\max}$ )
12: end if

```

5 Stability

The shearlets are not linearly dependent, and so the stiffness matrices are expected to have a number of zero eigenvalues. This is not a (major) obstacle to solving the system, however. Of greater importance is the *effective* condition number, defined as the ratio between the greatest and smallest eigenvalues which are not zero, within machine precision bounds. There will typically be an obvious gap between machine-precision zero eigenvalues and nonzero ones.

The frame property holds for shearlet systems of various types (such as the $s_x = 2, s_y = 1$ case, see [11]), but all proofs so far apply only to shearlets without compact support or shearlets with prohibitively large polynomial degree, and for domains without boundary. Such properties make practical application for PDEs nigh impossible. It would be most welcome if a piecewise linear compactly supported shearlet system could be shown to be stable.

In figures 9-14 we present spectra for various cases of our construction. The machine-precision zero eigenvalues are shown in blue, and the genuinely non-zero eigenvalues are shown in red. In each plot, the computed effective condition number is shown in the top left.

In each case, the eigenvalue computation was carried out by MATLAB's `eig`.

As can be seen, the prospects are not particularly encouraging. The system appears asymptotically unstable in all cases, and prohibitively badly conditioned after only a few levels. This strongly indicates that the frame property does not hold for the infinite set of piecewise linear, compactly supported shearlets.

It remains an open question whether a viable and stable piecewise polynomial shearlet frame can be found, and which properties it may have.

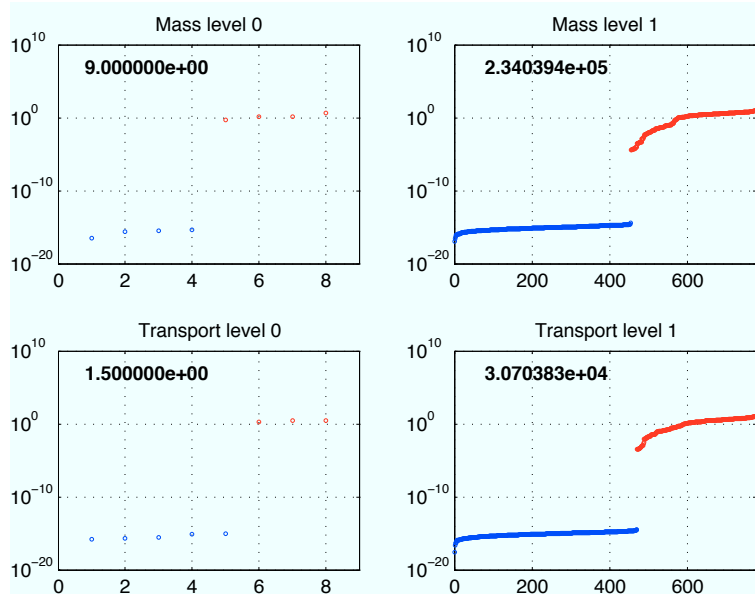


Figure 9: Eigenvalue plots for $s_x = 4$, $s_y = 2$, periodic with two mother shearlets.

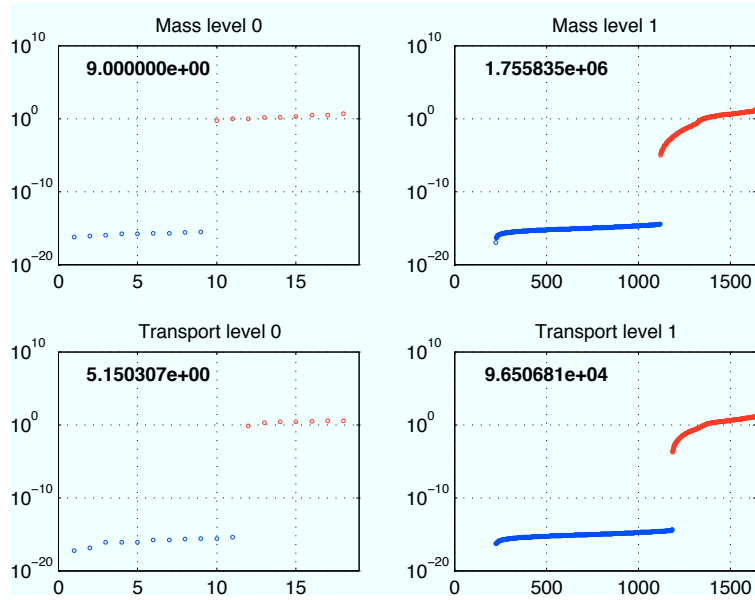


Figure 10: Eigenvalue plots for $s_x = 4$, $s_y = 2$, non-periodic with two mother shearlets.

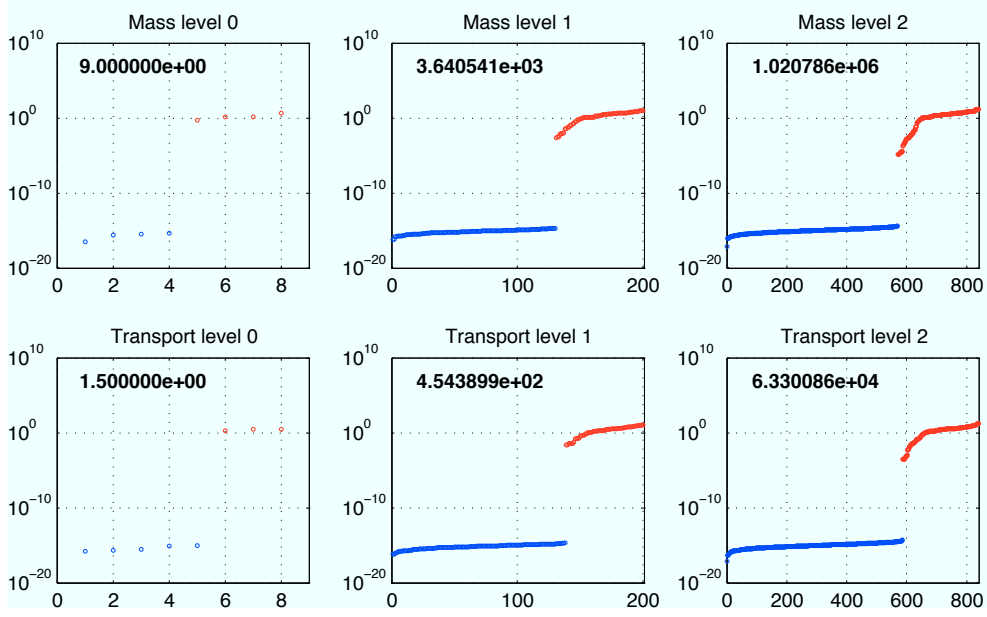


Figure 11: Eigenvalue plots for $s_x = 2$, $s_y = 1$, periodic with two mother shearlets.

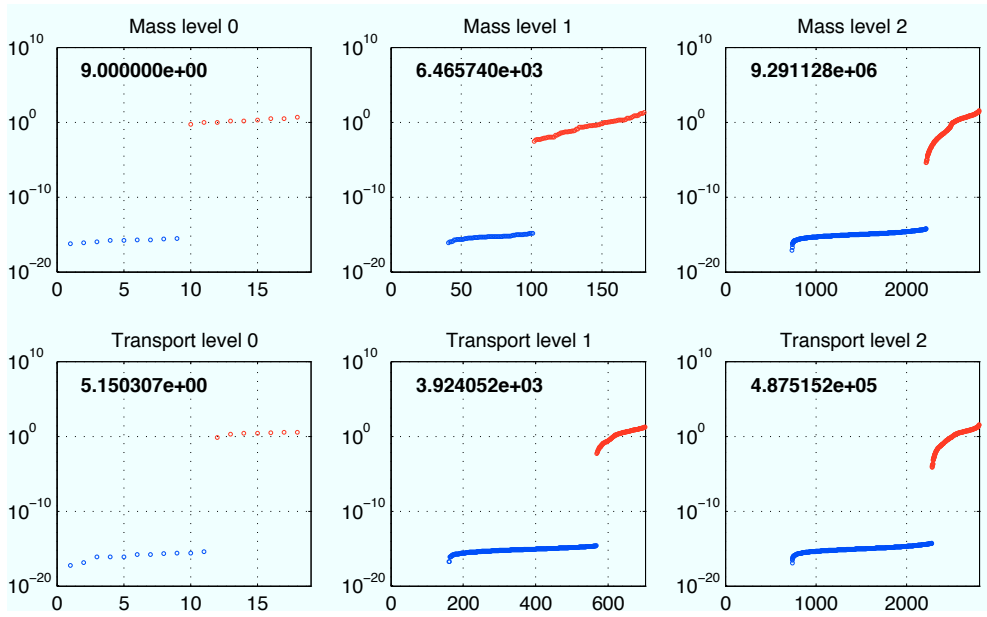


Figure 12: Eigenvalue plots for $s_x = 2$, $s_y = 1$, non-periodic with mother shearlets.

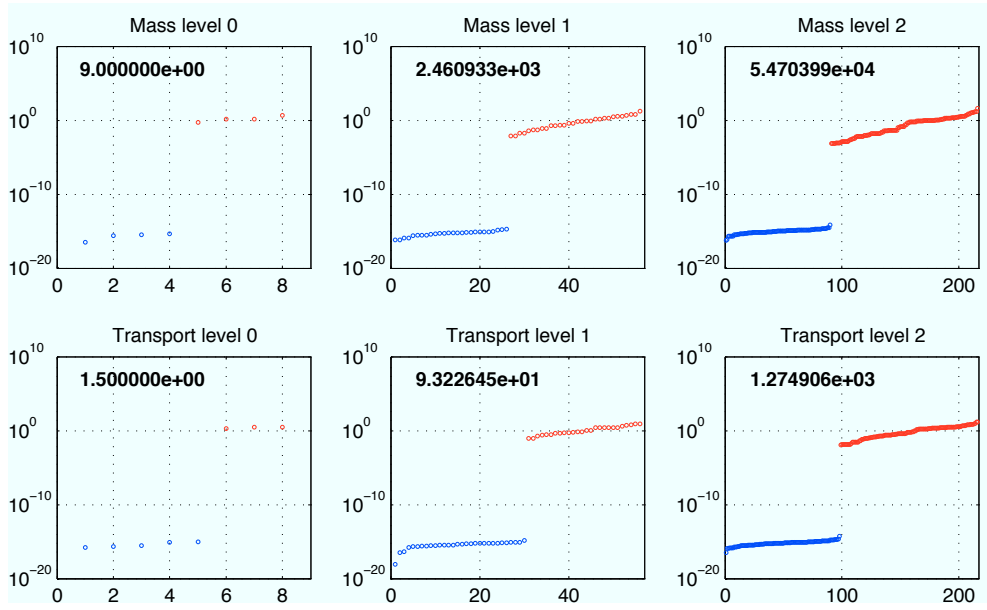


Figure 13: Eigenvalue plots for $s_x = 2$, $s_y = 1$, periodic with one mother wavelet (as in figure 3, but on all levels.)

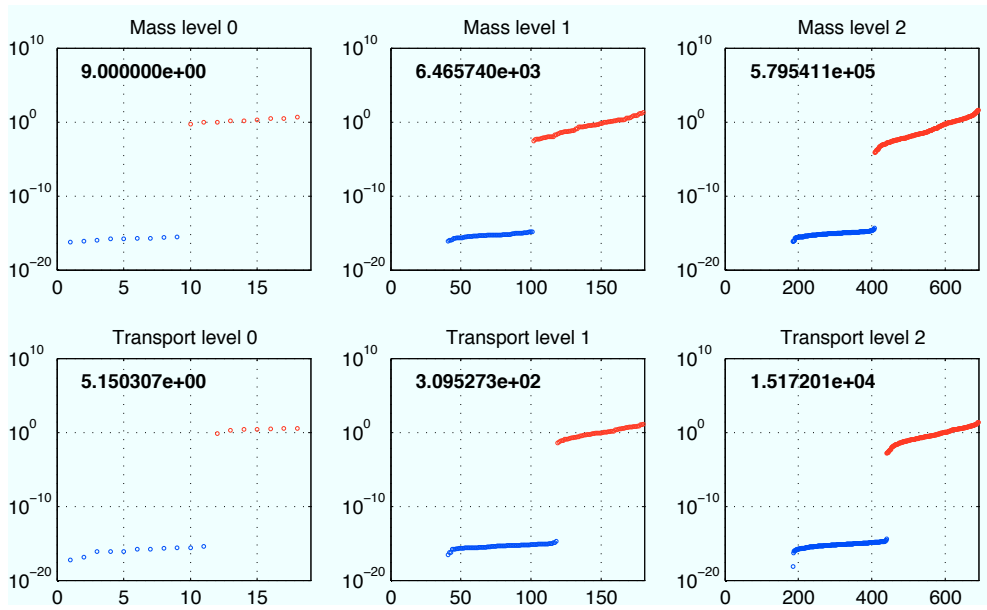


Figure 14: Eigenvalue plots for $s_x = 2$, $s_y = 1$, non-periodic with one mother wavelet (as in figure 3, but on all levels.)

A Code listings

The following code listings are mostly real MATLAB code equivalents for the algorithms defined earlier. There is usually a one-to-one correspondence between algorithm and file, but not always. The names also usually carry over.

A.1 Settings

The `Settings` method holds a persistent struct with various parameters. Due to issues with parallelization, every function that needs these settings (which is essentially each of them) take the settings struct as an optional last parameter. If it is not given, the function will call `Settings` to get them. It is good practice to always pass this struct explicitly, lest inconsistencies should occur in parallel loops.

The default parameters correspond closely to what has been described in this paper.

```
1 function out = Settings(varargin)
2
3     persistent params
4     persistent locked
5
6     if ~exist('params') || isempty(params) || ((nargin > 0 &&
7         strcmp(varargin{1}, 'clear')) && ~locked)
8
9         params.let = 'shearlet';
10        params.interior = 0;
11        params.periodic = 0;
12        params.qr = P706();
13        params.k = @(x) ones(1, size(x,2));
14        params.f = @(x) ones(1, size(x,2));
15        params.s = @(x) [ones(1, size(x,2));
16            ones(1, size(x,2))];
17        params.maxlevel = 2;
18        params.numfunctions = 2 * ones(1, params.maxlevel+1);
19        params.numfunctions(1) = 1;
20
21        for l=2:params.maxlevel+1
22            params.wvfunctions(l,1).xpts = [-.5, -.25, 0,
23                .25, .5];
24            params.wvfunctions(l,1).fvals = [0, -.5, 1, -.5,
25                0];
26            params.wvfunctions(l,2).xpts = [-.5, -.25, 0,
27                .25, .5];
28            params.wvfunctions(l,2).fvals = [0, -sqrt(2)/4,
29                0, sqrt(2)/4, 0];
30            params.scfunctions(l,1).xpts = [-.5, 0, .5];
31            params.scfunctions(l,1).fvals = [0, 1, 0];
32            params.scfunctions(l,2).xpts = [-.5, 0, .5];
33            params.scfunctions(l,2).fvals = [0, 1, 0];
34        end
35        params.wvfunctions(1,1).xpts = [-.5, 0, .5];
36        params.wvfunctions(1,1).fvals = [0, 1, 0];
37        params.scfunctions(1,1).xpts = [-.5, 0, .5];
```

```

32     params.scfunctions(1,1).fvals = [0, 1, 0];
33
34     params.xscale = 4;
35     params.yscale = 2;
36
37     params = fixlevels(params);
38
39     params.desc = 'Default';
40
41     locked = 0;
42
43 end
44
45 if nargin > 0 && strcmp(varargin{1}, 'lock')
46     locked = 1;
47     disp('Settings locked');
48 elseif nargin > 0 && strcmp(varargin{1}, 'unlock')
49     locked = 0;
50     disp('Settings unlocked');
51 elseif locked && nargin > 0
52     disp('Settings change interrupted');
53 else
54
55     for j = 1:2:nargin
56         switch lower(varargin{j})
57             case 'let'
58                 if strcmp(varargin{j+1}, 'ridgelet')
59                     params.let = 'ridgelet';
60                 else
61                     params.let = 'shearlet';
62                 end
63             case 'interior'
64                 if varargin{j+1} == 1
65                     params.interior = 1;
66                 else
67                     params.interior = 0;
68                 end
69             case 'periodic'
70                 if varargin{j+1} == 1
71                     params.periodic = 1;
72                 else
73                     params.periodic = 0;
74                 end
75             params = fixlevels(params);
76             case 'qr'
77                 params.qr = varargin{j+1};
78             case 'k'
79                 params.k = varargin{j+1};
80             case 'f'
81                 params.f = varargin{j+1};
82             case 's'
83                 params.s = varargin{j+1};
84             case 'glob'

```

```

85         params.globd = varargin{j+1};
86     case 'globd'
87         params.globd = varargin{j+1};
88     case 'maxlevel'
89         params.maxlevel = varargin{j+1};
90         for j =
91             length(params.numfunctions):params.maxlevel;
92                 params.wvfunctions(end+1,:) =
93                     params.wvfunctions(end,:);
94                 params.scfunctions(end+1,:) =
95                     params.scfunctions(end,:);
96                 params.numfunctions(end+1) =
97                     params.numfunctions(end);
98         end
99         params = fixlevels(params);
100     case 'numfunctions'
101         params.numfunctions = varargin{j+1};
102         params = fixlevels(params);
103     case 'wvfunctions'
104         params.wvfunctions = varargin{j+1};
105     case 'scfunctions'
106         params.scfunctions = varargin{j+1};
107     case 'xscale'
108         params.xscale = varargin{j+1};
109         params = fixlevels(params);
110     case 'yscale'
111         params.yscale = varargin{j+1};
112         params = fixlevels(params);
113     case 'desc'
114         params.desc = varargin{j+1};
115     end
116 end
117
118 end
119
120 out = params;
121
122 function params = fixlevels(params)
123
124     params.rightskip = zeros(1,params.maxlevel+1);
125     params.upskip = zeros(1,params.maxlevel+1);
126     params.leftstart = zeros(1,params.maxlevel+1);
127     params.downstart = zeros(1,params.maxlevel+1);
128     params.numright = zeros(1,params.maxlevel+1);
129     params.numup = zeros(1,params.maxlevel+1);
130     params.numshears = zeros(1,params.maxlevel+1);
131     params.numatlevel = zeros(1,params.maxlevel+1);
132     params.numbelowlevel = zeros(1,params.maxlevel+1);
133
134     for i = 0:params.maxlevel
135         if i == 0
136             params.rightskip(i+1) = 0.5;
137             params.upskip(i+1) = 0.5;

```

```

134         params.numshears(i+1) = 1;
135     else
136         params.rightskip(i+1) = params.xscale^(-i)/4;
137         params.upskip(i+1) = params.yscale^(-i)/2;
138         params.numshears(i+1) = 2^i + 1;
139     end
140
141     params.downstart(i+1) = 0;
142     if params.periodic
143         params.leftstart(i+1) = 0;
144         params.numright(i+1) =
145             1/params.rightskip(i+1);
146         params.numup(i+1) = 1/params.upskip(i+1);
147     else
148         if i == 0
149             params.leftstart(i+1) = 0;
150         else
151             params.leftstart(i+1) =
152                 params.rightskip(i+1) -
153                 0.5*(params.xscale^(-i) +
154                     params.yscale^(-i));
155         end
156         params.numright(i+1) =
157             round((1-2*params.leftstart(i+1))/params.rightskip(i+1))
158             + 1;
159         params.numup(i+1) =
160             round((1-2*params.downstart(i+1))/params.upskip(i+1))
161             + 1;
162     end
163
164     params.numatlevel(i+1) = 2 *
165         params.numfunctions(i+1) * params.numup(i+1)
166         * params.numright(i+1) *
167         params.numshears(i+1);
168     params.numbelowlevel(i+1) =
169         sum(params.numatlevel);
170 end
171 end
172 end

```

A.2 Making and manipulating shearlets

The `GetShearlet` method is called with a single integer as argument and returns a struct which defines the shearlet. The `Right`, `Up` and `Flip` methods move shearlets to the right or upwards or flips the cone. These all rely on the `RefreshShearletData` function to fill in the struct with more useful information. Here, we also present our code for `GetTransform` and `GetShearletCorners`.

```

1 function s = GetShearlet(i, params)
2
3     if nargin < 2

```

```

4         params = Settings;
5     end
6
7     orig = i;
8
9     s.Level = 0;
10    while i > params.numatlevel(s.Level+1)
11        i = i - params.numatlevel(s.Level+1);
12        s.Level = s.Level + 1;
13    end
14
15    s.Cone = 0;
16    if i > params.numatlevel(s.Level+1) / 2
17        i = i - params.numatlevel(s.Level+1) / 2;
18        s.Cone = 1;
19    end
20
21    s.F = mod(i-1, params.numfunctions(s.Level+1));
22    i = ceil(i/params.numfunctions(s.Level+1));
23
24    s.K = mod(i-1, params.numshears(s.Level+1)) -
25        (params.numshears(s.Level+1)-1)/2;
26    i = ceil(i/params.numshears(s.Level+1));
27
28    s.R = mod(i-1, params.numright(s.Level+1));
29    i = ceil(i/params.numright(s.Level+1));
30
31    s.U = i-1;
32
33    s = RefreshShearletData(s, params);
34
35    if s.N ~= orig
36        disp(['Error in getting shearlet ' num2str(orig) '
37            [GetShearlet.m]']);
38    end
end

```

```

1 function s = Right(s, n, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     if nargin < 2 || isempty(n)
8         n = 1;
9     end
10
11    s.R = s.R + n;
12    if s.R >= params.numright(s.Level+1)
13        s.R = params.numright(s.Level+1) - 1;
14    elseif s.R < 0
15        s.R = 0;

```



```

16     end;
17     s = RefreshShearletData(s, params);
18
19 end

```

```

1 function s = Up(s, n, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     if nargin < 2 || isempty(n)
8         n = 1;
9     end
10
11    s.U = s.U + n;
12    if s.U >= params.numup(s.Level+1)
13        s.U = params.numup(s.Level+1) - 1;
14    elseif s.U < 0
15        s.U = 0;
16    end;
17    s = RefreshShearletData(s, params);
18
19 end

```

```

1 function s = Flip(s, params)
2
3     if nargin < 2
4         params = Settings;
5     end
6
7     s.Cone = 1 - s.Cone;
8     s = RefreshShearletData(s, params);
9
10 end

```

```

1 function s = RefreshShearletData(s, params)
2
3     if nargin < 2
4         params = Settings;
5     end
6
7     if s.Cone == 0,
8         s.X = params.leftstart(s.Level+1) + s.R *
9             params.rightskip(s.Level+1);
10        s.Y = params.downstart(s.Level+1) + s.U *
11            params.upskip(s.Level+1);
12    else
13        s.Y = params.leftstart(s.Level+1) + s.R *
14            params.rightskip(s.Level+1);
15        s.X = params.downstart(s.Level+1) + s.U *
16            params.upskip(s.Level+1);

```

```

13     end;
14
15     n = 0;
16     if s.Level > 0
17         n = params.numbelowlevel(s.Level);
18     end
19     if s.Cone == 1
20         n = n + params.numfunctions(s.Level+1) *
                params.numup(s.Level+1) *
                params.numright(s.Level+1) *
                params.numshears(s.Level+1);
21     end
22     n = n + s.U * params.numfunctions(s.Level+1) *
                params.numright(s.Level+1) *
                params.numshears(s.Level+1);
23     n = n + s.R * params.numfunctions(s.Level+1) *
                params.numshears(s.Level+1);
24     n = n + params.numfunctions(s.Level+1) * (s.K +
                (params.numshears(s.Level+1)-1)/2);
25     n = n + s.F;
26     n = n + 1;
27
28     s.N = n;
29
30     r = GetShearletCorners(s, 0, params);
31     s.Radius = max(sqrt(sum((r-repmat([s.X;s.Y],1,4)).^2,
                1)));
32
33 end

```

```

1 function [P, c] = GetTransform(s, i, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     c = [s.X; s.Y];
8     if isfield(s, 'K')
9         a = params.xscale / params.yscale;
10        P = [1, a*s.K; 0, 1] * [params.xscale^s.Level, 0; 0,
                params.yscale^s.Level];
11    else
12        P = [params.xscale^s.Level, 0; 0,
                params.yscale^s.Level];
13    end;
14    if s.Cone == 1, P = P * [0 1; 1 0]; end;
15
16    if nargin > 1 && i > 0
17        XS =
18            length(params.wvfunctions(s.Level+1,s.F+1).xpts)-1;
19        YS =
20            length(params.scfunctions(s.Level+1,s.F+1).xpts)-1;

```

```

20     Qi = [XS, 0; 0, YS];
21     ci = [(mod(i-1,XS)+.5)/XS-.5;
           (floor((i-1)/XS)+.5)/YS-.5];
22     c = P\ci+c;
23     P = Qi*P;
24     end
25
26 end

```

```

1  function p = GetShearletCorners(s, i, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     if nargin < 1
8         p = [0, 1, 1, 0; 0, 0, 1, 1];
9     else
10        if nargin < 2
11            i = 0;
12        end
13        p = [-.5, .5, .5, -.5; -.5, -.5, .5, .5];
14        [P, c] = GetTransform(s, i, params);
15        p = (P\p) + repmat(c, 1, size(p,2));
16        if (s.Cone == 1), p = fliplr(p); end;
17    end
18
19 end

```

A.3 Evaluating shearlets

The `PreparePoints` method takes care of all the transforming being done, as well as some work related to periodicity. It returns the transformed points as well as a vector of boolean values denoting whether or not this point falls inside the support (not always obvious in the periodic case). `EvaluateShearlet` and `EvaluateShearletGradient` use this to do the raw work.

```

1  function [x, computefor] = PreparePoints(s, x, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     [P, c] = GetTransform(s, 0, params);
8     x = P*(x-repmat(c, 1, size(x,2)));
9
10    computefor = 1:size(x,2);
11
12    if params.periodic
13        dirs = P * [1 0; 0 1];
14        if s.Cone == 1
15            dirs = fliplr(dirs);

```

```

16     end
17     n_min = ceil((-x(2,:)-0.5)/dirs(2,2));
18     n_max = floor((-x(2,:)+0.5)/dirs(2,2));
19
20     computefor = setdiff(computefor, find(n_min>n_max));
21     n = n_min;
22
23     m_min = ceil((-x(1,:)-n*dirs(1,2)-0.5)/dirs(1,1));
24     m_max = floor((-x(1,:)-n*dirs(1,2)+0.5)/dirs(1,1));
25
26     computefor = setdiff(computefor, find(m_min>m_max));
27     m = m_min;
28
29     x = x + repmat(m,2,1).*repmat(dirs(:,1),1,size(x,2))
        + repmat(n,2,1).*repmat(dirs(:,2),1,size(x,2));
30 end
31
32 end

```

```

1 function out = EvaluateShearlet(s, x, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     xorig = x;
8     [x, computefor] = PreparePoints(s, x, params);
9
10    wv_xpts = params.wvfunctions(s.Level+1,s.F+1).xpts;
11    wv_fvals = params.wvfunctions(s.Level+1,s.F+1).fvals;
12    gm = @(x) inter(wv_xpts, wv_fvals, x);
13
14    sc_xpts = params.scfunctions(s.Level+1,s.F+1).xpts;
15    sc_fvals = params.scfunctions(s.Level+1,s.F+1).fvals;
16    th = @(x) inter(sc_xpts, sc_fvals, x);
17
18    out = zeros(1,size(x,2));
19    out(computefor) =
20        th(x(2,computefor)).*gm(x(1,computefor));
21    out = 2^(-3*s.Level/2)*out;
22
23    if isfield(params, 'glob')
24        out = out .* params.glob(xorig(:,computefor));
25    end
26
27    function out = inter(xs, ys, x)
28        out = interp1([min([xs,x])-1, xs, max([xs,x])+1], [0,
29            ys, 0], x);
30    end
31
32 end

```

```

1 function out = EvaluateShearletGradient(s, x, params)

```

```

2
3     if nargin < 3
4         params = Settings;
5     end
6
7     xorig = x;
8     [x, computefor] = PreparePoints(s, x, params);
9
10    wv_xpts = params.wvfunctions(s.Level+1,s.F+1).xpts;
11    wv_fvals = params.wvfunctions(s.Level+1,s.F+1).fvals;
12    gm = @(x) inter(wv_xpts, wv_fvals, x);
13    gmd = @(x) interd(wv_xpts, wv_fvals, x);
14
15    sc_xpts = params.scfunctions(s.Level+1,s.F+1).xpts;
16    sc_fvals = params.scfunctions(s.Level+1,s.F+1).fvals;
17    th = @(x) inter(sc_xpts, sc_fvals, x);
18    thd = @(x) interd(sc_xpts, sc_fvals, x);
19
20    [P, c] = GetTransform(s, 0, params);
21    shld = zeros(2, size(x,2));
22    shld(:, computefor) =
23        [th(x(2, computefor)).*gmd(x(1, computefor));
24         thd(x(2, computefor)).*gm(x(1, computefor))];
25    shld = 2^(-3*s.Level/2)*P'*shld;
26
27    if ~isfield(params, 'glob') || ~isfield(params, 'globd')
28        out = shld;
29    else
30        shl = zeros(1, size(x,2));
31        shl(computefor) =
32            th(x(2, computefor)).*gm(x(1, computefor));
33        shl = 2^(-3*s.Level/2)*shl;
34        out = repmat(shl, 2, 1).*params.globd(xorig) +
35            repmat(params.glob(xorig), 2, 1).*shld;
36    end;
37
38    function out = inter(xs, ys, x)
39        out = interp1([min([xs,x])-1, xs, max([xs,x])+1], [0,
40            ys, 0], x);
41    end
42
43    function out = interd(xs, ys, x)
44        gpts = length(xs);
45        vpts = length(x);
46
47        loc = sum(repmat(x, gpts, 1) > repmat(xs', 1, vpts), 1);
48        ys = [ys(1), ys, ys(end)];
49        xs = [xs(1)-1, xs, xs(end)+1];
50
51        out = (ys(loc+2)-ys(loc+1))./(xs(loc+2)-xs(loc+1));
52    end
53 end

```

A.4 Linear and bilinear forms

These methods evaluate the integrand of some linear and bilinear forms. They take one or two shearlets and the points of evaluation as parameters. We have BLFKK, BLFSS, BLFSK and BLF for the bilinear forms for the reaction term, the transport term, the cross term, and everything together respectively. For linear forms we have similarly LFK and LFS.

```
1 function out = BLFKK(u, v, x, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     out = params.k(x).^2 .* EvaluateShearlet(u, x, params) .*
8         EvaluateShearlet(v, x, params);
9 end
```

```
1 function out = BLFSS(u, v, x, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     out = sum(params.s(x) .* EvaluateShearletGradient(u, x,
8         params), 1) .* sum(params.s(x) .*
9         EvaluateShearletGradient(v, x, params), 1);
10 end
```

```
1 function out = BLFSK(u, v, x, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     out = sum(params.s(x) .* EvaluateShearletGradient(u, x,
8         params), 1) .* params.k(x) .* EvaluateShearlet(v, x,
9         params);
10 end
```

```
1 function out = BLF(u, v, x, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     ux = EvaluateShearlet(u, x, params);
8     vx = EvaluateShearlet(v, x, params);
9     ud = EvaluateShearletGradient(u, x, params);
10    vd = EvaluateShearletGradient(v, x, params);
```

```

11
12     p = params;
13
14     out = p.k(x).^2.*ux.*vx + sum(p.s(x).*ud, 1).*p.k(x).*vx
15         + sum(p.s(x).*vd, 1).*p.k(x).*ux + sum(p.s(x).*ud,
16         1).*sum(p.s(x).*vd, 1);

```

```

1 function out = LFK(v, x, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     out = EvaluateShearlet(v, x, params) .* params.f(x) .*
8         params.k(x);
9 end

```

```

1 function out = LFS(v, x, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     out = sum(params.s(x) .* EvaluateShearletGradient(v, x,
8         params), 1) .* params.f(x);
9 end

```

A.5 Intersections

Here, we have code for `CheckPolygonIntersection`, which checks if two convex polygons are disjoint or not, `CheckShearletIntersection`, which checks if the supports of two shearlets are disjoint or not (and relies on the former), and `GetIntersection`, which actually computes the intersection between two shearlet (sub)supports, relying on the GPC library [19] which is compiled. This method needs the services of `CrossesBoundary`, which computes flags denoting if a shearlet crosses parts of the boundary of the domain (if periodicity is enabled).

```

1 function out = CheckPolygonIntersection(p, q)
2
3     if isempty(p) || isempty(q)
4         out = 0;
5         return;
6     end
7
8     cp = [p(:,end), p];
9     cq = [q(:,end), q];
10

```

```

11  % Loop through P
12  for i = 2:size(cp,2)
13
14      base_tmp = cp(:,i) - cp(:,i-1);
15      base = [base_tmp(2); -base_tmp(1)];
16
17      separator = 1;
18      for j = 2:size(cq,2)
19          if dot(cq(:,j) - cp(:,i), base) < 0
20              separator = 0;
21              break;
22          end
23      end
24
25      if separator
26          out = 0;
27          return;
28      end
29
30  end
31
32  for i = 2:size(cq,2)
33
34      base_tmp = cq(:,i) - cq(:,i-1);
35      base = [base_tmp(2); -base_tmp(1)];
36
37      separator = 1;
38      for j = 2:size(cp,2)
39          if dot(cp(:,j) - cq(:,i), base) < 0
40              separator = 0;
41              break;
42          end
43      end
44
45      if separator
46          out = 0;
47          return;
48      end
49
50  end
51
52  out = 1;
53
54  end

```

```

1  function out = CheckShearletIntersection(u, v, params)
2
3      if nargin < 3
4          params = Settings;
5      end
6
7      p = GetShearletCorners(u, 0, params);
8      q = GetShearletCorners(v, 0, params);

```



```

9
10 Nvals = 0;
11 Mvals = 0;
12
13 if params.periodic
14     deltax = v.X - u.X;
15     deltay = v.Y - u.Y;
16     rad = v.Radius + u.Radius;
17
18     Nvals = ceil(-rad-deltax):floor(rad-deltax);
19     Mvals = ceil(-rad-deltay):floor(rad-deltay);
20 end
21
22 for N = Nvals
23     for M = Mvals
24         if CheckPolygonIntersection(p, q +
25             repmat([N;M],1,size(q,2)))
26             out = 1;
27             return;
28         end
29     end
30
31     out = 0;
32
33 end

```

```

1 function [np, p] = GetIntersection(u, i, v, j, params)
2
3     if nargin < 5
4         params = Settings;
5     end
6
7     if nargin < 4
8         j = 0;
9     end
10
11    if nargin < 3
12        v = [];
13    end
14
15    if nargin < 2
16        i = 0;
17    end
18
19    sp = GetShearletCorners();
20    S.x = sp(1,:); S.y = sp(2,:); S.hole = 0;
21    U = BuildPolygon(u, i, params);
22
23    R = PolygonClip(U, S, 1);
24
25    if length(R) == 0
26        np = 0;

```

```

27     p = [];
28     return;
29 end
30
31 if nargin > 2 && ~isempty(v)
32     V = BuildPolygon(v, j, params);
33     R = PolygonClip(R, V, 1);
34
35     if length(R) == 0
36         np = 0;
37         p = [];
38         return;
39     else
40         np = length(R);
41         for k = 1:length(R)
42             p(k).p = fliplr([R(k).x'; R(k).y']);
43         end
44         return;
45     end
46 else
47     np = length(R);
48     for k = 1:length(R)
49         p(k).p = fliplr([R(k).x'; R(k).y']);
50     end
51     return;
52 end
53
54 function U = BuildPolygon(u, i, params)
55
56     up = GetShearletCorners(u, i, params);
57     U(1).x = up(1,:); U(1).y = up(2,:); U(1).hole = 0;
58
59     if params.periodic
60         [r, l, u, d] = CrossesBoundary(params, u, i);
61
62         U = AddToPol(U, 1, -r, 0);
63         U = AddToPol(U, 1, l, 0);
64         U = AddToPol(U, 1, 0, -u);
65         U = AddToPol(U, 1, 0, d);
66         U = AddToPol(U, 1, -(r&u), -(r&u));
67         U = AddToPol(U, 1, l&u, -(l&u));
68         U = AddToPol(U, 1, -(r&d), r&d);
69         U = AddToPol(U, 1, l&d, l&d);
70     end
71
72 end
73
74 function pol = AddToPol(pol, ref, xadd, yadd)
75     if (xadd ~= 0) | (yadd ~= 0)
76         k = length(pol) + 1;
77         pol(k).x = pol(ref).x + xadd;
78         pol(k).y = pol(ref).y + yadd;
79         pol(k).hole = pol(ref).hole;

```

```

80     end
81   end
82
83 end

```

A.6 Building quadrature rules

Here, we have `SplitTriangle`, which splits a collection of triangles (defined as a $6 \times N$ matrix with vertices stacked in the first dimension) into a collection of smaller triangles, in the light of the argumentation presented in section 4.3.4. We also have `BuildQuadRuleOnPolygon`, which takes a polygon and constructs a quadrature rule on it, also implementing the subdivision routine of algorithm 6, using `TransformQuadRule` for transforming a base quadrature rule onto a triangle. This method is again used by `BuildDoubleQuadRule` and `BuildSingleQuadRule`, which constructs quadrature rules on the intersection of the supports of two shearlets or the support of one shearlet respectively.

```

1  function out = SplitTriangle(trs, L)
2
3      if L > 0
4
5          out = zeros(6, 4*size(trs,2));
6          i = 1;
7
8          for tr = trs
9              q = reshape(tr, 2, 3);
10             out(:, i:i+3) = [reshape([q(:,1),
11                                     avg(q(:,1),q(:,2)), avg(q(:,1),q(:,3))], 6,
12                                     1),...
13                               reshape([q(:,2),
14                                       avg(q(:,2),q(:,3)),
15                                       avg(q(:,2),q(:,1))], 6, 1),...
16                               reshape([q(:,3),
17                                       avg(q(:,3),q(:,1)),
18                                       avg(q(:,3),q(:,2))], 6, 1),...
19                               reshape([avg(q(:,1),q(:,2)),
20                                       avg(q(:,3),q(:,2)),
21                                       avg(q(:,3),q(:,1))], 6, 1)];
22
23             i = i + 4;
24         end
25
26         out = SplitTriangle(out, L-1);
27
28     else
29
30         out = trs;
31
32     end
33
34     function out = avg(a, b)
35         out = 0.5 * (a+b);
36     end

```

```
28
29 end
```

```
1 function r = BuildQuadRuleOnPolygon(p, L, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     r.x = [];
8     r.w = [];
9
10    if isempty(p)
11        return;
12    end
13
14    idxs = [1, 2, size(p,2)];
15    sw = 2; endidx = 1; startidx = 3;
16
17    while true
18        trs = SplitTriangle(reshape(p(:, idxs), 6, 1), L);
19
20        for tr = trs
21            temp = TransformQuadRule(params.qr, reshape(tr,
22                2, 3));
23            if sum(temp.w) > 0
24                r.x = [r.x, temp.x];
25                r.w = [r.w, temp.w];
26            end;
27        end
28
29        if (sw == 2)
30            idxs = [idxs(2), size(p,2)-endidx, idxs(3)];
31            endidx = endidx + 1;
32            sw = 1;
33        elseif (sw == 1)
34            idxs = [idxs(1), startidx, idxs(2)];
35            startidx = startidx + 1;
36            sw = 2;
37        end;
38
39        if length(unique(idxs)) < 3,
40            break;
41        end;
42    end;
43 end
```

```
1 function r = BuildDoubleQuadRule(u, v, params)
2
3     if nargin < 3
4         params = Settings;
5     end
```

```

6
7     if u.N == v.N
8         r = BuildSingleQuadRule(u, 0, params);
9     else
10        r.x = [];
11        r.w = [];
12
13        numSectionsU =
14            (length(params.wvfunctions(u.Level+1,u.F+1).xpts)-1)
15            *
16            (length(params.scfunctions(u.Level+1,u.F+1).xpts)-1);
17        numSectionsV =
18            (length(params.wvfunctions(v.Level+1,v.F+1).xpts)-1)
19            *
20            (length(params.scfunctions(v.Level+1,v.F+1).xpts)-1);
21
22        for i = 1:numSectionsU
23            for j = 1:numSectionsV
24
25                [np,p] = GetIntersection(u, i, v, j, params);
26
27                for k = 1:np
28                    rule = BuildQuadRuleOnPolygon(p(k).p, 0,
29                        params);
30                    r.x = [r.x, rule.x];
31                    r.w = [r.w, rule.w];
32                end
33            end; end;
34        end
35    end
36 end

```

```

1 function r = BuildSingleQuadRule(v, L, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     r.x = [];
8     r.w = [];
9
10    if nargin < 2 || isempty(L)
11        L = params.maxlevel - v.Level;
12    end
13
14    numSections =
15        (length(params.wvfunctions(v.Level+1,v.F+1).xpts)-1)
16        *
17        (length(params.scfunctions(v.Level+1,v.F+1).xpts)-1);
18
19    for i = 1:numSections

```

```

17         [np,p] = GetIntersection(v, i, [], 0, params);
18
19         for k = 1:np
20             rule = BuildQuadRuleOnPolygon(p(k).p, L, params);
21             r.x = [r.x, rule.x];
22             r.w = [r.w, rule.w];
23         end
24     end
25
26 end
27
28 end

```

A.7 Tying it all together

The methods `IntegrateBLF` and `EvaluateBLF` (along with their LF namesakes for linear forms) take care of the actual integration (using a quadrature rule) in the former case, and orchestrating the whole affair of checking intersections and building quadrature rules (and *then* integrating) in the second case.

```

1 function out = IntegrateBLF(bl, u, v, qr, params)
2
3     if nargin < 5
4         params = Settings;
5     end
6
7     out = sum(qr.w .* bl(u, v, qr.x, params));
8
9 end

```

```

1 function out = IntegrateLF(lf, v, qr, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     out = sum(qr.w .* lf(v, qr.x, params));
8
9 end

```

```

1 function out = EvaluateBLF(bl, u, v, params)
2
3     if nargin < 4
4         params = Settings;
5     end
6
7     if CheckShearletIntersection(u, v, params);
8
9         [np,p] = GetIntersection(u, 0, v, 0, params);
10
11         if np == 0
12             out = 0;

```

```

13         return;
14     else
15         qr = BuildDoubleQuadRule(u, v, params);
16         if size(qr.x,2) > 0
17             out = IntegrateBLF(blf, u, v, qr, params);
18         else
19             out = 0;
20         end
21     end
22
23     else
24         out = 0;
25     end
26
27 end

```

```

1 function out = EvaluateLF(lf, v, params)
2
3     if nargin < 3
4         params = Settings;
5     end
6
7     if CheckPolygonIntersection(GetShearletCorners(),
8         GetShearletCorners(v, 0, params))
9
10        [np,p] = GetIntersection(v, 0, [], 0, params);
11
12        if np == 0
13            out = 0;
14            return;
15        else
16            qr = BuildSingleQuadRule(v, [], params);
17            if size(qr.x,2) > 0
18                out = IntegrateLF(lf, v, qr, params);
19            else
20                out = 0;
21            end
22        end
23
24    else
25        out = 0;
26    end
27 end

```

A.8 Matrices and vectors

On the top of the food chain, we find the functions `Stiffness` and `Load`, which store the stiffness matrices and load vectors in persistent variables. This is also where the parallelization takes place.

`Stiffness` takes as arguments a matrix identifier, two index sets denoting a submatrix of the global stiffness matrix, a bilinear form and a boolean denoting

whether or not the matrix should be symmetric. It then looks up the matrix in memory, determines which (if any) new elements need to be computed, computes them, and then returns the desired submatrix. Load works in a similar manner.

```

1 function ret = Stiffness(k, I, J, blf, sym, params)
2
3     N = 10000;
4     NNZ = 1000*N;
5
6     persistent A;
7     persistent nodes;
8     persistent defined;
9
10    if isa(k, 'char')
11        if strcmp(k, 'clear')
12            disp('Clearing...');
13            clear A;
14            clear nodes;
15            clear defined;
16            ret = 0;
17            return;
18        end
19    end
20
21    if ~exist('defined')
22        defined = [];
23    end
24
25    if (length(defined) < k) || (defined(k) == 0)
26        disp('Allocating...');
27        A{k} = spalloc(N, N, NNZ);
28        nodes{k} = zeros(1, N);
29        defined(k) = 1;
30    end
31
32    if nargin < 6
33        params = Settings();
34    end
35
36    maxidx = max(max(I),max(J));
37    if maxidx > length(nodes{k})
38        nodes{k} = [nodes{k}, zeros(1,
39                    maxidx-length(nodes{k}))];
40    end
41
42    newI = find(nodes{k}(I) == 0);
43    newJ = find(nodes{k}(J) == 0);
44    newnodes = unique([I(newI), J(newJ)]);
45    oldnodes = find(nodes{k} == 1);
46
47    if size(newnodes, 1) == 0
48        newnodes = [];
49    end

```



```

50     nums = 0;
51     N = length(newnodes);
52     tots = length(oldnodes)*N + N*(N+1)/2;
53     disp(' ');
54     if tots > 0
55         progmeter(0, ['Computing ' num2str(tots) ' new
56             elements of matrix ' num2str(k)]);
57     end
58     for i = newnodes,
59         tempH = zeros(1,length(oldnodes));
60         tempV = zeros(length(oldnodes),1);
61         parfor j = 1:length(oldnodes)
62             tempH(j) = EvaluateBLF(blF,
63                 GetShearlet(oldnodes(j), params),
64                 GetShearlet(i, params), params);
65             if sym,
66                 tempV(j) = tempH(j);
67             else
68                 tempV(j) = EvaluateBLF(blF, GetShearlet(i,
69                     params), GetShearlet(oldnodes(j),
70                         params), params);
71             end
72         end
73     end
74     A{k}(i,oldnodes) = tempH;
75     A{k}(oldnodes,i) = tempV;
76     A{k}(i,i) = EvaluateBLF(blF, GetShearlet(i, params),
77         GetShearlet(i, params), params);
78     nums = nums + length(oldnodes) + 1;
79     progmeter(nums/tots);
80     nodes{k}(i) = 1;
81     oldnodes = [oldnodes, i];
82 end
83 ret = A{k}(I,J);
84 if tots > 0
85     progmeter done
86 end
end

```

```

1 function ret = Load(k, I, lf, params)
2
3     N = 10000;
4
5     persistent L;
6     persistent nodes;
7     persistent defined;
8

```

```

9     if isa(k, 'char')
10         if strcmp(k, 'clear')
11             disp('Clearing...');
12             clear L;
13             clear nodes;
14             clear defined;
15             ret = 0;
16             return;
17         end
18     end
19
20     if ~exist('defined')
21         defined = [];
22     end
23
24     if (length(defined) < k) || (defined(k) == 0)
25         disp('Allocating...');
26         L{k} = zeros(N, 1);
27         nodes{k} = zeros(1, N);
28         defined(k) = 1;
29     end
30
31     if nargin < 4
32         params = Settings();
33     end
34
35     newI = find(nodes{k}(I) == 0);
36     newnodes = I(newI);
37
38     if size(newnodes, 1) == 0
39         newnodes = [];
40     end
41
42     nums = 0;
43     templk = L{k};
44     tempnodesk = nodes{k};
45     parfor i = newnodes,
46         templk(i) = EvaluateLF(lf, GetShearlet(i), params);
47         %nums = nums + 1;
48         %progmeter(nums/length(newnodes));
49
50         tempnodesk(i) = 1;
51     end
52     nodes{k} = tempnodesk;
53     L{k} = templk;
54
55     ret = L{k}(I);
56
57 end

```

References

- [1] J. P. Antoine, R. Murenzi, and P. Vandergheynst. Directional wavelets revisited: Cauchy wavelets and symmetry detection in patterns. *Appl. Computat. Harmon. Anal.*, 6:314–345, 1999.
- [2] I. Babuška and J. M. Melenk. The partition of unity method. *International Journal for Numerical Methods in Engineering*, 40:727–758, 1997.
- [3] E. Candes and L. Demanet. The curvelet representation of wave propagators is optimally sparse. *Communications in Pure and Applied Mathematics*, 58:1472–1528, 2004.
- [4] E. Candes and D. Donoho. New tight frames of curvelets and optimal representations of objects with piecewise c^2 singularities. *Communications in Pure and Applied Mathematics*, 57:219–266, 2002.
- [5] E. Candès and D. Donoho. Curvelets: a surprising effective nonadaptive representation for objects with edges. In *in Curve and Surface Fitting: Saint-Malo*. University Press, 2000.
- [6] B. Chazelle and D. P. Dobkin. Detection is easier than computation (extended abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 146–153, New York, NY, USA, 1980. ACM.
- [7] S. Dahlke, M. Fornasier, M. Primbs, T. Raasch, and M. Werner. Nonlinear and adaptive frame approximation schemes for elliptic pdes: Theory and numerical experiments. *Numerical Methods for Partial Differential Equations*, 25:1366–1401, 2009.
- [8] S. Dahlke, M. Fornasier, and T. Raasch. Adaptive frame methods for elliptic operator equations. *Advances in Computational Mathematics*, 27:27–63, 2007.
- [9] S. Dahlke, T. Raasch, M. Werner, M. Fornasier, and R. Stevenson. Adaptive frame methods for elliptic operator equations: the steepest descent approach. *IMA Journal of Numerical Analysis*, 27:717–740, 2007.
- [10] M. N. Do and M. Vetterli. The contourlet transform: an efficient directional multiresolutoin image representation. *IEEE Transactions on Image Processing*, 14:2091–2106, 2005.
- [11] P. Grohs. Ridgelet-type frame decompositions for sololev spaces related to linear transport. Technical report, SAM, ETH Zurich, 2010.
- [12] K. Guo and D. Labate. Optimally sparse multidimensional representation using shearlets. *SIAM J. Math. Anal.*, 39:298–318, 2007.
- [13] K. Guo and D. Labate. Representation of fourier integral operators using shearlets. *Journal of Fourier Analysis and Applications*, 14(3):327–371, 2008.
- [14] P. Kittipoom, G. Kutyniok, and W.-Q Lim. Construction of compactly supported shearlet frames. Technical report, 2010.
- [15] G. Kutyniok and T. Saurer. Adaptive directional subdivision schemes and shearlet multiresolution analysis. *SIAM J. Math. Anal.*, 41:1436–1471, 2009.
- [16] D. Labate, W.-Q Lim, G. Kutyniok, and G. Weiss. Sparse multidimensional representation using shearlets. *SPIE Proc.*, 5914:254–262, 2005.

- [17] T. A. Manteuffel, K. J. Ressel, and G. Starke. A boundary functional for the least-squares finite-element solution of neutron transport problems. *SIAM J. Numer. Anal.*, 37(2):556–586, 2000.
- [18] J. M. Melenk and I. Babuška. The partition of unity finite element method: Basic theory and applications. *Comput. Methods Appl. Mech. Engrg.*, 139:289–314, 1996.
- [19] A. Murta and T. Howard. See <http://www.cs.man.ac.uk/~toby/alan/software/>.
- [20] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19:384–391, 1982.
- [21] H. Smith. A parametrix construction for wave equations with $c^1, 1$ -coefficients. *Annales de l'Institut Fourier*, 48:797–835, 1998.
- [22] R. Stevenson. Adaptive solution of operator equations using wavelet frames. *SIAM J. Numer. Anal.*, 41:1074–1100, 2003.
- [23] G. T. Toussaint. A simple linear algorithm for intersecting convex polygons. *The Visual Computer*, 1:118–123, 1985.
- [24] S. Yi, D. Labate, G. R. Easley, and H. Krim. A shearlet approach to edge analysis and detection. *IEEE Trans. Image Process.*, 18:929–941, 2009.

Research Reports

No.	Authors/Title
11-50	<i>E. Fonn</i> Shearlet Galerkin for transport equations: implementation and stability
11-49	<i>M.J. Castro, U.S. Fjordholm, S. Mishra and C. Parés</i> Entropy conservative and entropy stable schemes for non-conservative hyperbolic systems
11-48	<i>D. Kressner and Ch. Tobler</i> Preconditioned low-rank methods for high-dimensional elliptic PDE eigenvalue problems
11-47	<i>H. Heumann and R. Hiptmair</i> Convergence of lowest order semi-Lagrangian schemes
11-46	<i>K. Grella and Ch. Schwab</i> Sparse discrete ordinates method in radiative transfer
11-45	<i>X. Claeys and R. Hiptmair</i> Boundary integral formulation of the first kind for acoustic scattering by composite structures
11-44	<i>A. Chkifa, A. Cohen, R. DeVore and Ch. Schwab</i> Sparse adaptive Taylor approximation algorithms for parametric and stochastic elliptic PDEs
11-43	<i>S. Chen and S. Mao</i> Anisotropic error bounds of Lagrange interpolation with any order in two and three dimensions
11-42	<i>R. Hiptmair and J. Li</i> Shape derivatives in differential forms I: An intrinsic perspective
11-41	<i>Ph. Grohs and Ch. Schwab</i> Sparse twisted tensor frame discretization of parametric transport operators
11-40	<i>J. Li, H. Liu, H. Sun and J. Zou</i> Imaging acoustic obstacles by hypersingular point sources
11-39	<i>U.S. Fjordholm, S. Mishra and E. Tadmor</i> Arbitrarily high order accurate entropy stable essentially non-oscillatory schemes for systems of conservation laws
11-38	<i>U.S. Fjordholm, S. Mishra and E. Tadmor</i> ENO reconstruction and ENO interpolation are stable