1 Introduction

2 Equation and discretization

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

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

for $x \in \Omega \subset \mathbb{R}^2$. Here, $s(x)$ is the transport direction, $\kappa(x)$ is the reaction coefficient, and $f(x)$ is the source term. This equation does not need boundary conditions unless $\kappa(x) \equiv 0$, in which case we use *inflow boundary* conditions

$$u(x) = g(x),$$

on the inflow boundary $\Gamma_- = \{x \in \partial\Omega \mid s(x) \cdot n(x) < 0\}$, where $n(x)$ is the outward pointing normal vector for $x \in \partial\Omega$. Furthermore, in our case, we will always work with $\Omega = [0,1]^2$.

This is an equation that can exhibit strong anisotropic features. Consider the case $\kappa = f = 0$. Since the boundary condition values are transported perfectly along the flow s, the regularity of g will directly determine the regularity of u, and any discontinuities or irregularities will be pointed in the direction of s. This is a very good test equation for anisotropic frames.

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

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

and the linear form

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

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

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

where $\tilde{\ell}$ is the linear form ℓ with $f - s \cdot \nabla g - \kappa g$ 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 [4] and [1]). In the following analysis, this multiplier function is always implied. 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

$\hat{\Psi}_m : \mathbb{R}^2 \to \mathbb{R}$ via affine transformations and translations,

$$\Psi_A(x) = \left(\mathcal{D}_A \hat{\Psi}\right)(x) = |A|^{-1/2} \hat{\Psi}(Ax), \qquad \Psi_c(x) = \left(\mathcal{T}_c \hat{\Psi}\right)(x) = \hat{\Psi}(x - c),$$

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

3.1 Parameters

Given mother shearlets $\hat{\Psi}_m$, the shearlets are defined as

$$\Psi^{j,k}_{m,c,r} = \mathcal{T}_c \mathcal{D}_{A^{j,k}_r} \hat{\Psi}_m, \tag{2}$$

where

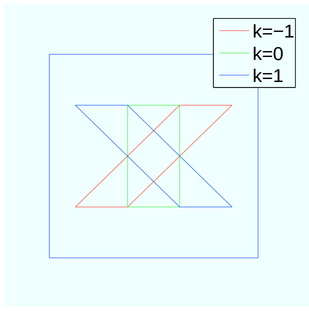
$$A^{j,k}_r = \begin{pmatrix} 1 & s_x k/s_y \\ & 1 \end{pmatrix} \begin{pmatrix} s_x^j & \\ & s_y^j \end{pmatrix} \begin{pmatrix} & 1 \\ 1 & \end{pmatrix}^r, \tag{3}$$

and $c \in \mathbb{R}^2$. 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 $(s_x, s_y) \in \left\{(4,2), (2,\sqrt{2}), (2,1)\right\}$. We will assume that $s_x > s_y \geq 1$ and often also that s_x/s_y is an integer.

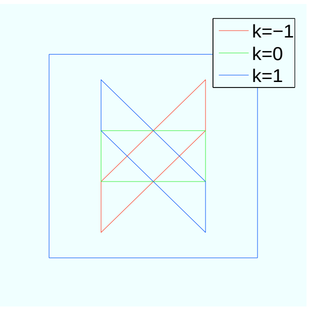
The significance of the parameters are as follows.

- m is the mother shearlet indicator.
- $c \in \mathbb{R}^2$ is the translation parameter. If $\hat{\Psi}_m$ is a function with support centered in the origin (as we will see later), $\Psi^{\cdot,\cdot}_{m,c,\cdot}$ is a function with support centered at c.
- The *scale* or *level* parameter $j \geq 0$ is an integer that scales $\hat{\Psi}_m$ anisotropically. For large j, the functions $\Psi^{j,\cdot}_{\cdot,\cdot,\cdot}$ will appear very long and thin, but nevertheless smaller. Specifically, if $\hat{\Psi}_m$ has square support, $\Psi^{j,0}_{m,\cdot,\cdot}$ 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 $\Psi^{\cdot,k}_{\cdot,\cdot,\cdot}$ 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 $\Psi^{\cdot,k}_{m,c,\cdot}$ has support centered at 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".

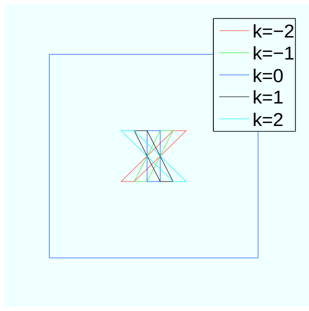
We offer some illustrations to make these concepts clear in figure 1.



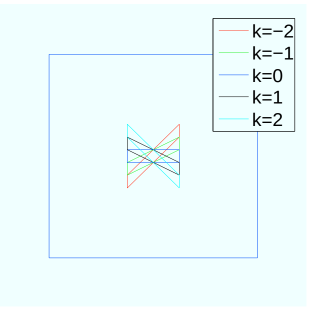
(a) $(4, 2, 1, 0)$



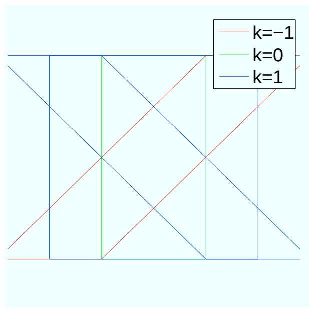
(b) $(4, 2, 1, 1)$



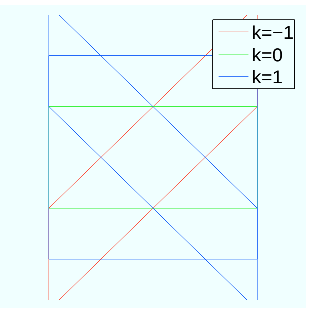
(c) $(4, 2, 2, 0)$



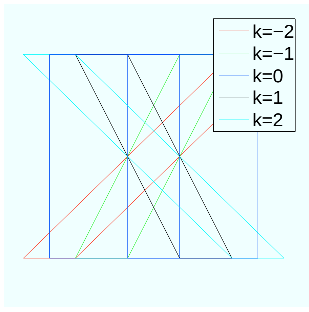
(d) $(4, 2, 2, 1)$



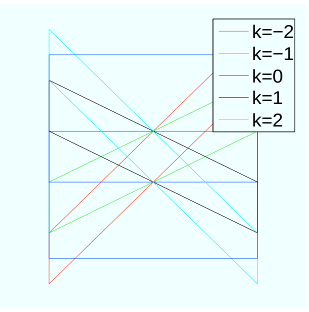
(e) $(2, 1, 1, 0)$



(f) $(2, 1, 1, 1)$

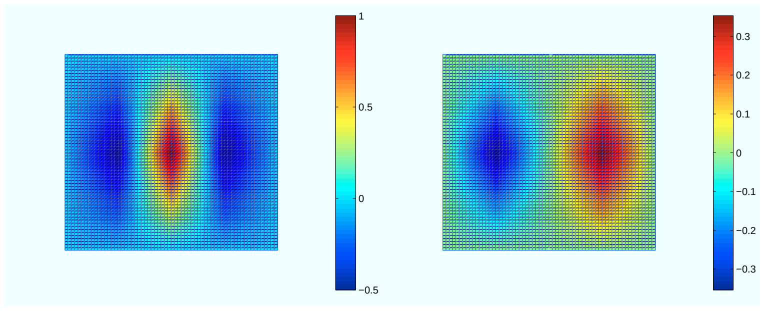


(g) $(2, 1, 2, 0)$



(h) $(2, 1, 2, 1)$

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.



(a) $m = 0$ (b) $m = 1$

Figure 2: Two mother shearlets.

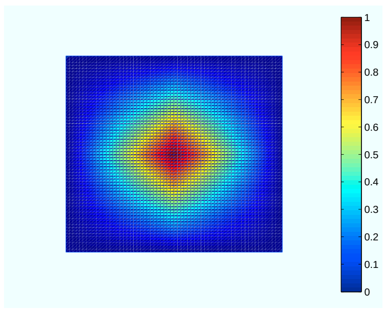


Figure 3: A level 0 mother shearlet.

3.2 The mother shearlet

The mother shearlets $\hat{\Psi}_m$ are usually chosen as tensor products $\hat{\Psi}_m(x) = \psi_m(x_1)\varphi_m(x_2)$, 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 *bi*linear on a rectangular mesh. Other choices are possible, but for our purposes we will restrict ourselves to linear $\psi.$ and $\varphi.$, 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 $\varphi(x) = -2|x|$, which is simply a hat function.

We (usually) use two different wavelets. They are

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

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 $\psi = \varphi$. TODO: Why? 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. We have already mentioned that if the reaction term κ is nonzero, there is no need for boundary conditions.

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 our shearlets do not necessarily do.

Thus, in the spirit of the partition of unity method ([4] and [1]), we multiply all basis functions with a global function $z(x)$ 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 $s(x) = (1,1)^T$, the inflow boundary consists of the edges along the x- and y-axes, and so $z(x) = x_1 x_2$ is a natural choice.

Furthermore, there is nothing to stop us from also investigating periodic boundary conditions. It is not hard to see that, by construction, the shearlets can be periodically extended and still be well-defined. The periodic case requires some special care in the programming, which will be dealt with where applicable in section 4.

3.4 Translations

It remains to specify the translation parameter c. Assume that the fundamental wavelets and scaling functions $\psi.$ and $\varphi.$ 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 then means that we can represent all tensor products of piecewise linear functions with such resolution by choosing translation steps spaced by s_x^{-j}/W in the first direction and s_y^{-j}/S in the second, whence

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

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

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

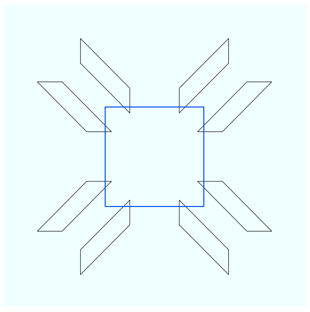


Figure 4: The extreme shearlets for $j = 1$, $s_x = 4$, $s_y = 2$.

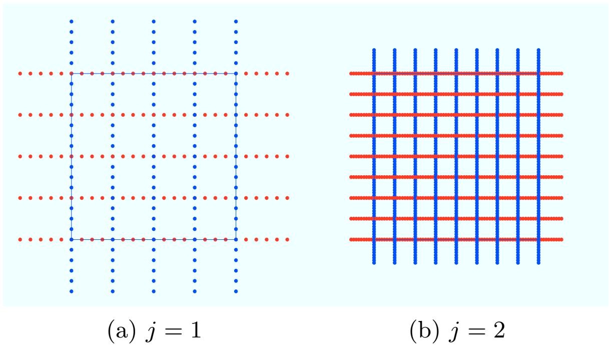


Figure 5: Translation points for the first two levels for $s_x = 4$, $s_y = 2$, $W = S = 4$. The blue points are for $r = 0$ and the red points are for $r = 1$.

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

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

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.

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 $\psi.$ and $\varphi.$ have different numbers of polynomial subparts (i.e. W and S depend on m). Our construction do not allow coupling between the translation 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.

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

Table 1: Number of shearlets 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 Numbers of shearlets

Asymptotically, the number of shearlets increases 16-fold for every added level. There will be four times as many translations in the left-right direction, twice as many in the up-down direction, as well as twice as many shears. The exact formula for $j > 1$ is (assuming s_x/s_y integral):

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

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 $j \to \infty$, 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 $j = 1, \ldots, 6$ 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 $\Psi^{j,k}_{m,c,r}$ is shearlet number N, then $\Psi^{j,k}_{m+1,c,r}$ is shearlet number $N+1$.

2. k comes next. Let M be the total number of mother shearlets. Then, if $\Psi^{j,k}_{m,c,r}$ is shearlet number N, then $\Psi^{j,k+1}_{m,c,r}$ is shearlet number $N + M$.
3. Next is the translation parameter c, with x-axis before y-axis. If there are K shears on level j, then if shearlet $\Psi^{j,k}_{m,c,r}$ is shearlet number N, then the corresponding shearlet immediately to the right is shearlet number $N+MK$, and the corresponding shearlet immediately over 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 are considered to be 1-indexed, while the parameters are 0-indexed.

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

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

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

Algorithm 2 Computes the shearlet number given the other parameters.

```
Require: $j$, $r$, $c = c(\alpha, \beta)$, $k$ and $m$
 1: $n \leftarrow 0$
 2: for $i \leftarrow 0, \ldots, j-1$ 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 $n$
```

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). A function called BaseTransform takes a shearlet and produces the transformation matrix A and the translation vector c.

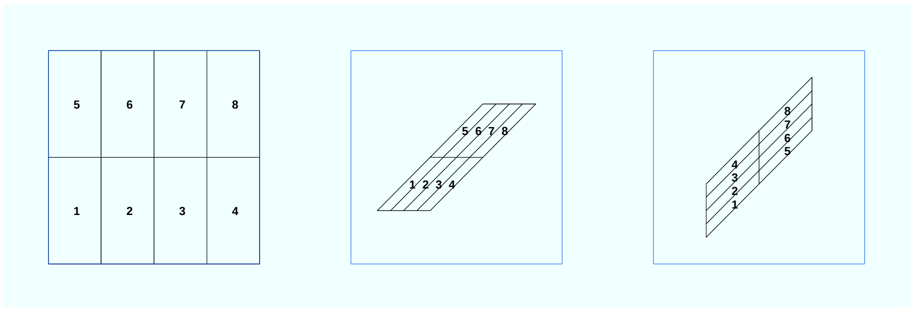
$$(A, c) \leftarrow \text{BaseTransform}(s).$$

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

The reason for the name BaseTransform is that it will be useful to have a more general function Transform that can produce mappings into subdomains of 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.

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 $2 \times n$



(a) Mother shearlet (b) $r = 0$ (c) $r = 1$

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

Algorithm 3 Transform computes A and c for a given shearlet and subdomain number.

```
Require: Shearlet $s$ and optional subdomain number $0 \le i \le 8$
1: $(A, c) \leftarrow \text{BaseTransform}(s)$
2: if $i$ is given and $i > 0$ then
3:     $d \leftarrow \left(\frac{1}{W}\left((i-1 \;\mathbf{mod}\; W) + \frac{1}{2}\right) - \frac{1}{2}, \quad \frac{1}{S}\left(\lfloor\frac{i-1}{W}\rfloor + \frac{1}{2}\right) - \frac{1}{2}\right)^T$
4:     $c \leftarrow A^{-1}d + c$
5:     $A \leftarrow \begin{pmatrix} W & \\ & S \end{pmatrix} \cdot A$
6: end if
7: return $(A, c)$
```

matrix of vertices $(p_1, p_2, \ldots, p_n)$ numbered counterclockwise, where $p_n \neq p_1$. 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: $(A, c) \leftarrow$ Transform$(s, i)$
 5: $p \leftarrow \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix}$
 6: $p \leftarrow A^{-1}p + c$
 7: if $s.r = 1$ then
 8:     Reverse the order of $p$
 9: end if
10: return $p$
```

4.3 Assembly strategy

4.3.1 Stiffness matrix

Now, given a bilinear form $\mathsf{a}(\cdot, \cdot)$ 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 fill out the coarser matrices. 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]

[1]It 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.

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 a. This is also where parallellization should enter the picture.

Algorithm 5 Assembly loop.

```
Require: Shearlet index set $I$.
1: for $i, j \in I$ do
2:     $A(i, j) \leftarrow \mathsf{a}(s_j, s_i)$
3:     $A(j, i) \leftarrow \mathsf{a}(s_i, s_j)$
4: end for
```

How, then, do we go about computing $\mathsf{a}(s_i, s_j)$? 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 poylgons 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 in hand, the integrand of the bilinear form a can be evaluated on the quadrature points and summed up accordingly.

There should be a basic choice of quadrature rule on 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.

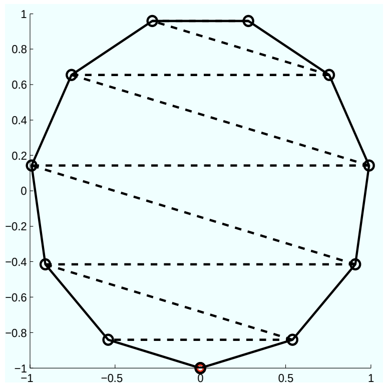


Figure 7: Splitting a polygon (here a regular 11-gon) into triangles.

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]

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]

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

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

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

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

```
Require: $2 \times N$ matrix $p$ denoting a polygon
1: start $\leftarrow$ 3, end $\leftarrow$ 1, right $\leftarrow$ false, $T \leftarrow \left\{ \begin{bmatrix} p_{\cdot,1} & p_{\cdot,2} & p_{\cdot,N} \end{bmatrix} \right\}$
2: while start + end $\leq N$ do
3:    if right then
4:       $T \leftarrow T \cup \left\{ \begin{bmatrix} p_{\cdot,\text{start}-1} & p_{\cdot,\text{start}} & p_{\cdot,N-\text{end}+1} \end{bmatrix} \right\}$
5:       right $\leftarrow$ true
6:       end $\leftarrow$ end + 1
7:    else
8:       $T \leftarrow T \cup \left\{ \begin{bmatrix} p_{\cdot,\text{start}-1} & p_{\cdot,N-\text{end}} & p_{\cdot,N-\text{end}+1} \end{bmatrix} \right\}$
9:       right $\leftarrow$ true
10:      end $\leftarrow$ end + 1
11:   end if
12: end while
```

Algorithm 7 BuildDoubleQuadrule 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:    $p \leftarrow$ Corners$(s, i)$
4:    $q \leftarrow$ Corners$(t, j)$
5:    $B \leftarrow$ SplitPolygon(ComputeIntersection$(p, q)$)
6:    for $c \in B$ do
7:       $r \leftarrow r \cup \{$TransformQuadrule$(q, c)\}$
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:    $q \leftarrow$ BuildDoubleQuadrule$(s, t, q)$
3:    $r \leftarrow 0$
4:    for quadrature point $x$ 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
```

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:     $p \leftarrow$ Corners$(s, i)$
4:     $B \leftarrow$ SplitPolygon$(p)$
5:     for $c \in B$ do
6:         $r \leftarrow r \cup$ TransformQuadrule$(q, c)$
7:     end for
8: end for
```

Algorithm 10 EvalLF evaluates a linear form.

```
Require: Shearlet $s$, basic quadrature rule $q$ and a function $f(s, x)$ for evaluating the
    integrand of a bilinear form at a point $x$.
1: $q \leftarrow$ BuildSingleQuadrule$(s, q)$
2: $r \leftarrow 0$
3: for quadrature point $x$ with weight $w \in q$ do
4:     $r \leftarrow r + w \cdot f(s, x)$
5: end for
6: return r
```

intersections, see for example [7], [6] and [2]. 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 [3].

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.*

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

If both a and b are vertices, consider vertex 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 b is not in L. If none of B intersects L, the edge r will be a separating line. Otherwise, consider the edge from b in direction of L. It must lie in the wedge between lines g and h. If it lies under g, 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 b is in L, it is not in U, and the argument works correspondingly. $\square$

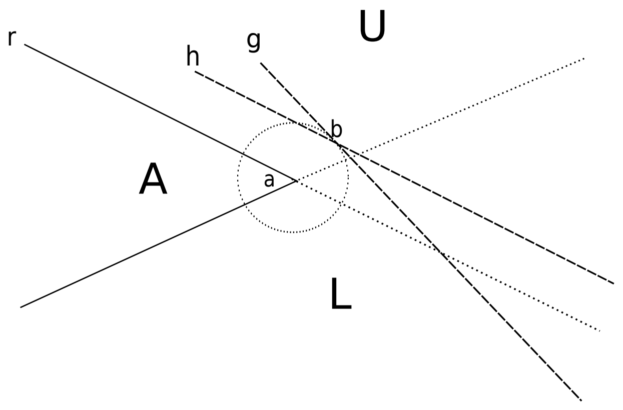


Figure 8: The separating axis theorem.

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.

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: $p \leftarrow$ Corners($s$), $q \leftarrow$ Corners($t$)
 2: for $e \in$ Edges($p$) do
 3:     $s \leftarrow$ true
 4:     for $v \in$ Vertices($q$) do
 5:         if $v$ is on the left side of $e$ then
 6:             $s \leftarrow$ 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
```

For actually *computing* the intersection between two polygons we have employed the General Polygon Clipper (GPC) library from the University of Manchester [5].

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 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 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 is strong enough. This will also be possible for the right hand side vector if the function f is simple enough. For most interesting experiments, however, these functions will vary, 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 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.

With piecewise linear shearlets, a quadrature rule that can integrate quadratics exactly is sufficient. On a rectangle of width h and height k, with $k > h$, such a quadrature rule would yield an error of $O(hk^3)$. Assuming this is an (unsheared) shearlet at level j, after l_j steps of additional subdivision, we have

$$h = \frac{1}{2^{l_j} \cdot W \cdot s_x^j}, \qquad k = \frac{1}{2^{l_j} \cdot S \cdot s_y^j}.$$

We want the errors from level j and i to be comparable. Thus

$$\frac{2^{l_i} \cdot W \cdot s_x^i \cdot 2^{3l_i} \cdot S^3 \cdot s_y^{3i}}{2^{l_j} \cdot W \cdot s_x^j \cdot 2^{3l_j} \cdot S^3 \cdot s_y^{3j}} = 1.$$

Some cancellation and algebra will reveal that the level relationship is

$$l_i = l_j + 4(j-i)\log_2\left(s_x \cdot s_y^3\right),$$

or, indeed, for a general quadrature rule of order n,

$$l_i = l_j + (n+1)(j-i)\log_2\left(s_x \cdot s_y^n\right).$$

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

For triangles, the relevant order 3 quadruature 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 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

$$\hat{\Psi}_m(x,y) = \psi(x)\varphi(y) \quad \Longrightarrow \quad \nabla\hat{\Psi}_m(x,y) = \left[\psi'(x)\varphi(y) \quad \psi(x)\varphi'(y)\right]^T,$$

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 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 by a deft programmer. An example can be found in section A.3.

Again, some modifications are necessary in the periodic case. Before transformation, a point x can be translated to any other point on a grid $\{x + m(1,0) + n(0,1)\}$, 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 ζ and ξ. Now, the problem is reduced to finding a translate $x + m\xi + n\eta$ in $[-1/2, 1/2]^2$. The key observation is that due to the nature of the transformation matrix 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

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

```
Require: Shearlet $s$, point $x$
 1: $(A, c) \leftarrow$ BaseTransform($s$)
 2: $y \leftarrow A(x - c)$
 3: if periodic then
 4:     $\eta \leftarrow A(\cdot, 1)$, $\xi \leftarrow A(\cdot, 2)$
 5:     if $\xi_2 \neq 0$ then
 6:         swap $\eta$ and $\xi$
 7:     end if
 8:     $n_{\min} \leftarrow \lceil -(x_2 + 1/2)/\eta_2 \rceil$, $n_{\max} \leftarrow \lfloor -(x_2 - 1/2)/\eta_2 \rfloor$, $n \leftarrow n_{\min}$
 9:     $m_{\min} \leftarrow \lceil -(x_2 + n\eta_1 + 1/2)/\xi_1 \rceil$, $n_{\max} \leftarrow \lfloor -(x_2 + n\eta_1 - 1/2)/\xi_1 \rfloor$, $m \leftarrow m_{\min}$
10:     return $y + m\xi + n\eta$
11:     (will be outside $[-1/2, 1/2]^2$ if $n_{\min} > n_{\max}$ or $m_{\min} > m_{\max}$)
12: end if
```

This algorithm can also be vectorized.

5 Stability

The shearlets form a frame, 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. The gap will generally be obvious.

In figures 9-14 we present spectra for various cases. 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.

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.

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

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,

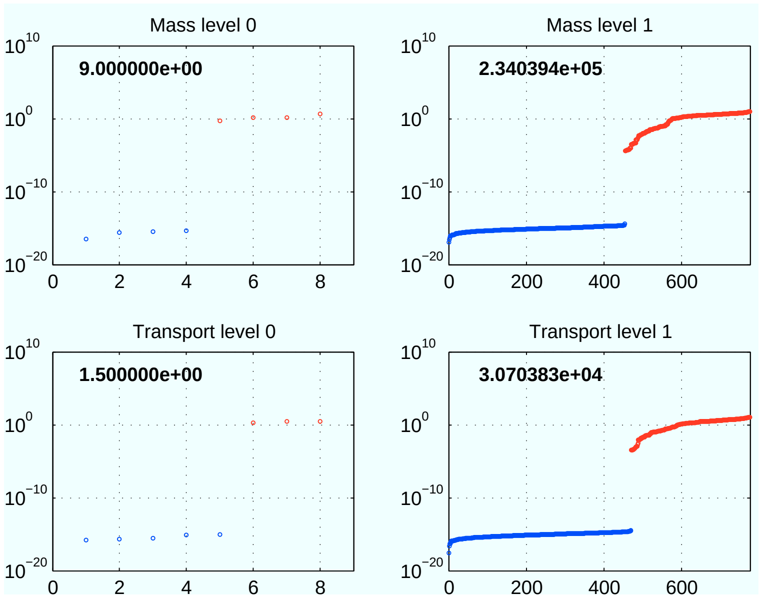


Figure 9: Eigenvalue plots for $s_x = 4$, $s_y = 2$, periodic with two mother shearlets.

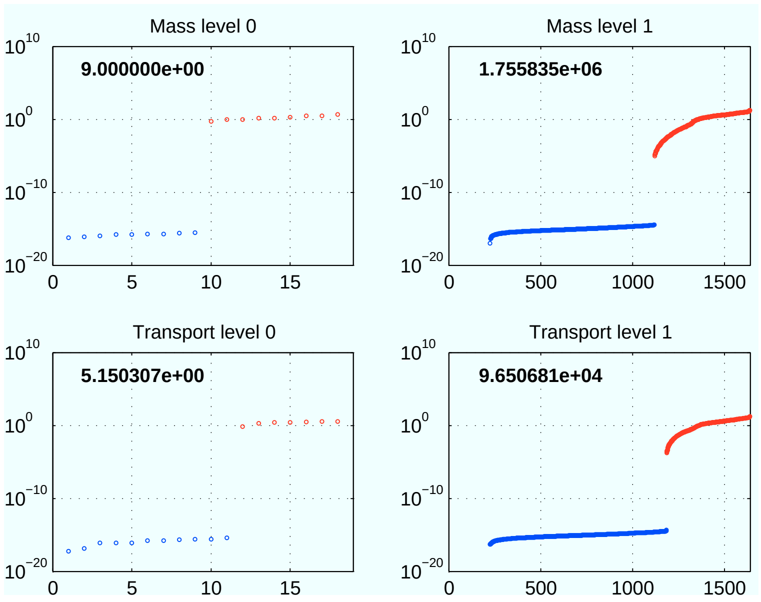


Figure 10: Eigenvalue plots for $s_x = 4$, $s_y = 2$, non-periodic with two mother shearlets.

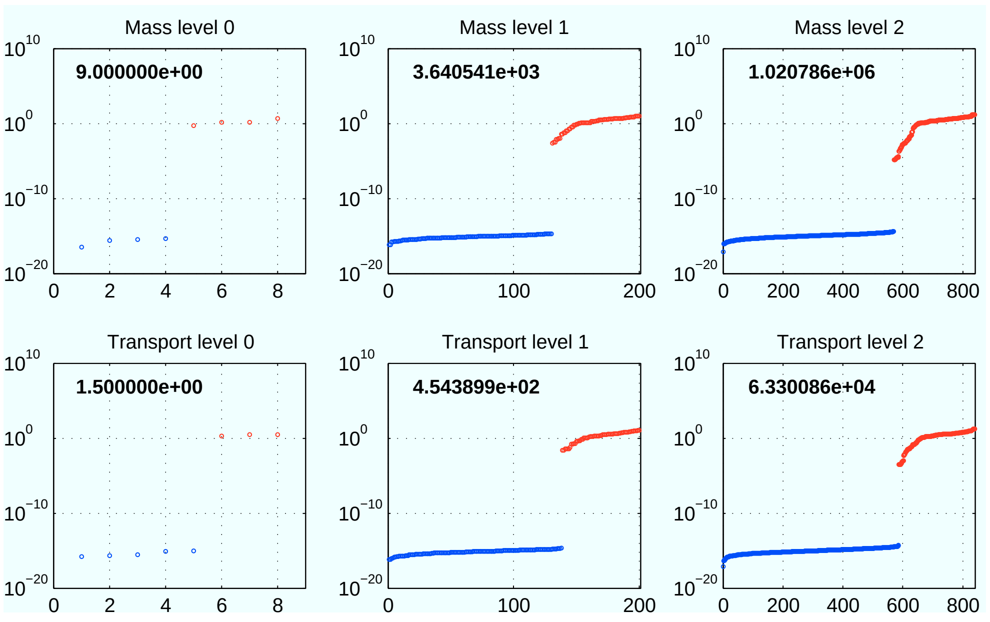


Figure 11: Eigenvalue plots for $s_x = 2$, $s_y = 1$, periodic with two mother shearlets.

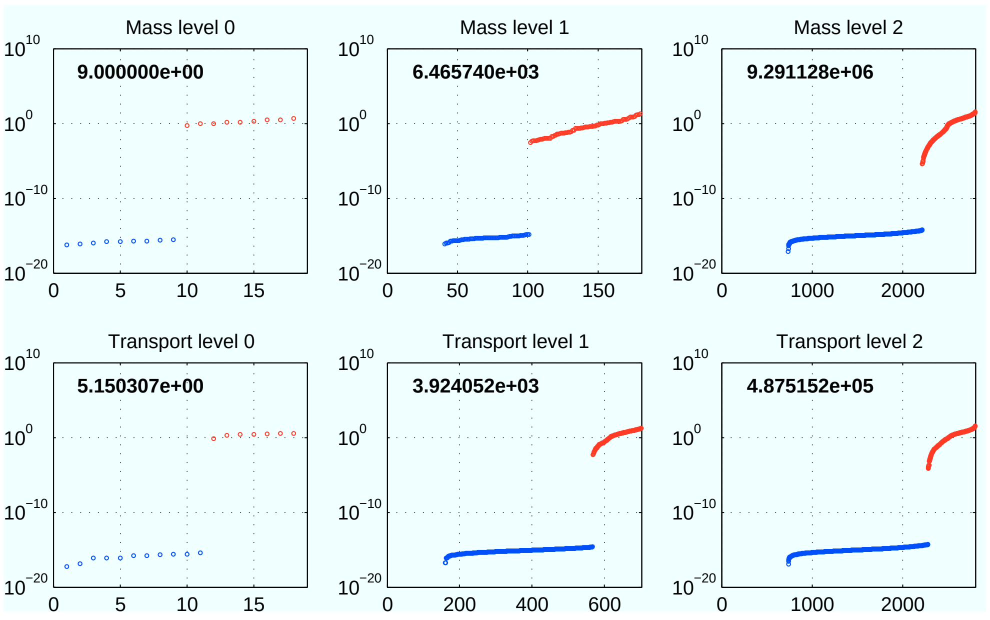


Figure 12: Eigenvalue plots for $s_x = 2$, $s_y = 1$, non-periodic with mother shearlets.

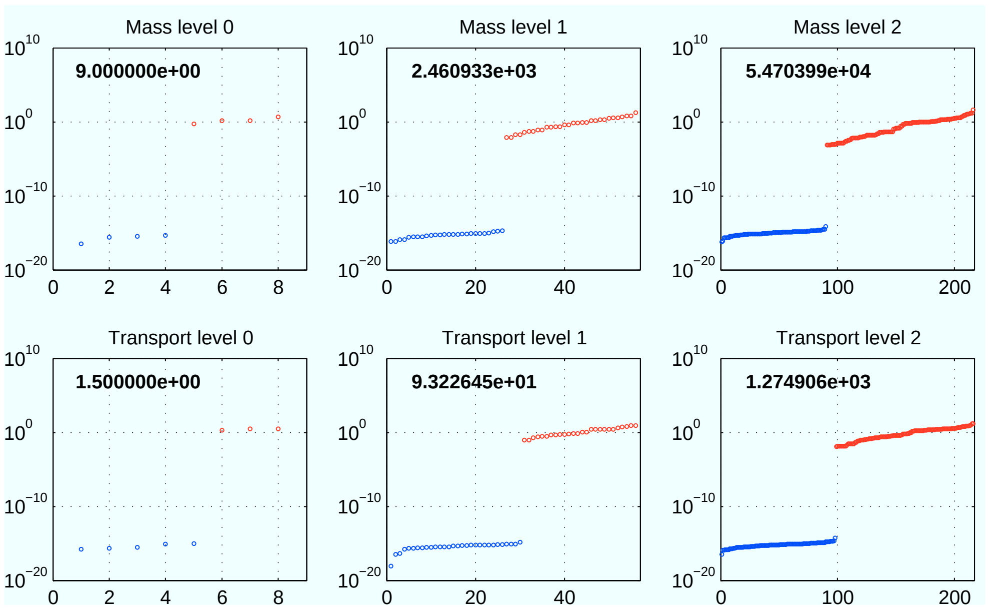


Figure 13: Eigenvalue plots for $s_x = 2$, $s_y = 1$, periodic with one mother wavelet (as in figure 3, but on all levels.)

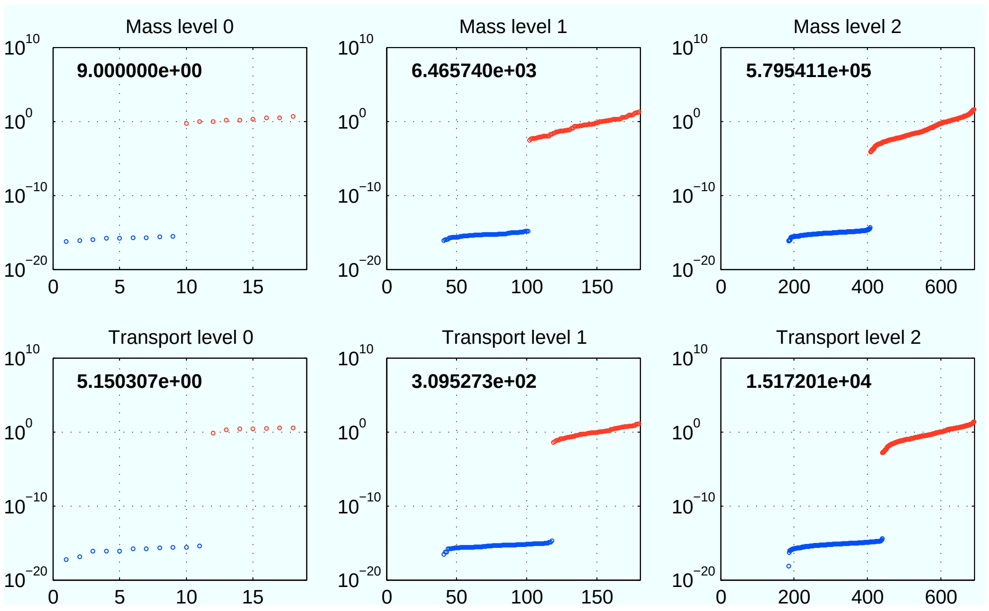


Figure 14: Eigenvalue plots for $s_x = 2$, $s_y = 1$, non-periodic with one mother wavelet (as in figure 3, but on all levels.)

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 parallellization, 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 parallell loops.

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

```
function out = Settings(varargin)

    persistent params
    persistent locked

    if ~exist('params') || isempty(params) || ((nargin > 0 &&
       strcmp(varargin{1},'clear')) && ~locked)

        params.let = 'shearlet';
        params.interior = 0;
        params.periodic = 0;
        params.qr = P706();
        params.k = @(x) ones(1,size(x,2));
        params.f = @(x) ones(1,size(x,2));
        params.s = @(x) [ones(1,size(x,2)); ones(1,size(x,2))];
        params.maxlevel = 2;
        params.numfunctions = 2 * ones(1,params.maxlevel+1);
        params.numfunctions(1) = 1;

        for l=2:params.maxlevel+1
            params.wvfunctions(l,1).xpts = [-.5, -.25, 0, .25, .5];
            params.wvfunctions(l,1).fvals = [0, -.5, 1, -.5, 0];
            params.wvfunctions(l,2).xpts = [-.5, -.25, 0, .25, .5];
            params.wvfunctions(l,2).fvals = [0, -sqrt(2)/4, 0,
                sqrt(2)/4, 0];
            params.scfunctions(l,1).xpts = [-.5, 0, .5];
            params.scfunctions(l,1).fvals = [0, 1, 0];
            params.scfunctions(l,2).xpts = [-.5, 0, .5];
            params.scfunctions(l,2).fvals = [0, 1, 0];
        end
        params.wvfunctions(1,1).xpts = [-.5, 0, .5];
        params.wvfunctions(1,1).fvals = [0, 1, 0];
        params.scfunctions(1,1).xpts = [-.5, 0, .5];
        params.scfunctions(1,1).fvals = [0, 1, 0];

        params.xscale = 4;
        params.yscale = 2;

        params = fixlevels(params);

        params.desc = 'Default';

        locked = 0;

    end

    if nargin > 0 && strcmp(varargin{1},'lock')
        locked = 1;
        disp('Settings locked');
    elseif nargin > 0 && strcmp(varargin{1},'unlock')
        locked = 0;
        disp('Settings unlocked');
    elseif locked && nargin > 0
        disp('Settings change interrupted');
    else

        for j = 1:2:nargin
            switch lower(varargin{j})
                case 'let'
                    if strcmp(varargin{j+1},'ridgelet')
                        params.let = 'ridgelet';
                    else
                        params.let = 'shearlet';
                    end
                case 'interior'
                    if varargin{j+1} == 1
                        params.interior = 1;
                    else
                        params.interior = 0;
                    end
                case 'periodic'
                    if varargin{j+1} == 1
                        params.periodic = 1;
                    else
                        params.periodic = 0;
                    end
                    params = fixlevels(params);
                case 'qr'
                    params.qr = varargin{j+1};
                case 'k'
                    params.k = varargin{j+1};
                case 'f'
                    params.f = varargin{j+1};
                case 's'
                    params.s = varargin{j+1};
                case 'glob'
                    params.glob = varargin{j+1};
                case 'globd'
                    params.globd = varargin{j+1};
                case 'maxlevel'
                    params.maxlevel = varargin{j+1};
                    for j = length(params.numfunctions):params.maxlevel;
                        params.wvfunctions(end+1,:) =
                            params.wvfunctions(end,:);
                    params.scfunctions(end+1,:) = params.scfunctions(end,:);
                    params.numfunctions(end+1) = params.numfunctions(end);
                end
                params = fixlevels(params);
            case 'numfunctions'
                params.numfunctions = varargin{j+1};
                params = fixlevels(params);
            case 'wvfunctions'
                params.wvfunctions = varargin{j+1};
            case 'scfunctions'
                params.scfunctions = varargin{j+1};
            case 'xscale'
                params.xscale = varargin{j+1};
                params = fixlevels(params);
            case 'yscale'
                params.yscale = varargin{j+1};
                params = fixlevels(params);
            case 'desc'
                params.desc = varargin{j+1};
        end
    end

end

out = params;

function params = fixlevels(params)

    params.rightskip = zeros(1,params.maxlevel+1);
    params.upskip = zeros(1,params.maxlevel+1);
    params.leftstart = zeros(1,params.maxlevel+1);
    params.downstart = zeros(1,params.maxlevel+1);
    params.numright = zeros(1,params.maxlevel+1);
    params.numup = zeros(1,params.maxlevel+1);
    params.numshears = zeros(1,params.maxlevel+1);
    params.numatlevel = zeros(1,params.maxlevel+1);
    params.numbelowlevel = zeros(1,params.maxlevel+1);

    for i = 0:params.maxlevel
        if i == 0
            params.rightskip(i+1) = 0.5;
            params.upskip(i+1) = 0.5;
            params.numshears(i+1) = 1;
        else
            params.rightskip(i+1) = params.xscale^(-i)/4;
            params.upskip(i+1) = params.yscale^(-i)/2;
            params.numshears(i+1) = 2^i + 1;
        end

        params.downstart(i+1) = 0;
        if params.periodic
```

```
                params.leftstart(i+1) = 0;
                params.numright(i+1) = 1/params.rightskip(i+1);
                params.numup(i+1) = 1/params.upskip(i+1);
            else
                if i == 0
                    params.leftstart(i+1) = 0;
                else
                    params.leftstart(i+1) = params.rightskip(i+1) -
                        0.5*(params.xscale^(-i) + params.yscale^(-i));
                end
                params.numright(i+1) =
                    round((1-2*params.leftstart(i+1))/params.rightskip(i+1))
                    + 1;
                params.numup(i+1) =
                    round((1-2*params.downstart(i+1))/params.upskip(i+1))
                    + 1;
            end

            params.numatlevel(i+1) = 2 * params.numfunctions(i+1) *
                params.numup(i+1) * params.numright(i+1) *
                params.numshears(i+1);
            params.numbelowlevel(i+1) = sum(params.numatlevel);
        end

    end

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.

```
function s = GetShearlet(i, params)

    if nargin < 2
        params = Settings;
    end

    orig = i;

    s.Level = 0;
    while i > params.numatlevel(s.Level+1)
        i = i - params.numatlevel(s.Level+1);
        s.Level = s.Level + 1;
    end

    s.Cone = 0;
    if i > params.numatlevel(s.Level+1) / 2
```

```
        i = i - params.numatlevel(s.Level+1) / 2;
        s.Cone = 1;
    end

    s.F = mod(i-1, params.numfunctions(s.Level+1));
    i = ceil(i/params.numfunctions(s.Level+1));

    s.K = mod(i-1, params.numshears(s.Level+1)) -
       (params.numshears(s.Level+1)-1)/2;
    i = ceil(i/params.numshears(s.Level+1));

    s.R = mod(i-1, params.numright(s.Level+1));
    i = ceil(i/params.numright(s.Level+1));

    s.U = i-1;

    s = RefreshShearletData(s, params);

    if s.N ~= orig
        disp(['Error in getting shearlet ' num2str(orig) '
            [GetShearlet.m]']);
    end

end
```

```
function s = Right(s, n, params)

    if nargin < 3
        params = Settings;
    end

    if nargin < 2 || isempty(n)
        n = 1;
    end

    s.R = s.R + n;
    if s.R >= params.numright(s.Level+1)
        s.R = params.numright(s.Level+1) - 1;
    elseif s.R < 0
        s.R = 0;
    end;
    s = RefreshShearletData(s, params);

end
```

```
function s = Up(s, n, params)

    if nargin < 3
        params = Settings;
    end

    if nargin < 2 || isempty(n)
        n = 1;
    end

    s.U = s.U + n;
    if s.U >= params.numup(s.Level+1)
        s.U = params.numup(s.Level+1) - 1;
    elseif s.U < 0
        s.U = 0;
    end;
    s = RefreshShearletData(s, params);

end
```

```
function s = Flip(s, params)

    if nargin < 2
        params = Settings;
    end

    s.Cone = 1 - s.Cone;
    s = RefreshShearletData(s, params);

end
```

```
function s = RefreshShearletData(s, params)

    if nargin < 2
        params = Settings;
    end

    if s.Cone == 0,
        s.X = params.leftstart(s.Level+1) + s.R *
            params.rightskip(s.Level+1);
        s.Y = params.downstart(s.Level+1) + s.U *
            params.upskip(s.Level+1);
    else
        s.Y = params.leftstart(s.Level+1) + s.R *
            params.rightskip(s.Level+1);
        s.X = params.downstart(s.Level+1) + s.U *
            params.upskip(s.Level+1);
    end;

    n = 0;
    if s.Level > 0
        n = params.numbelowlevel(s.Level);
    end
    if s.Cone == 1
        n = n + params.numfunctions(s.Level+1) *
            params.numup(s.Level+1) * params.numright(s.Level+1) *
            params.numshears(s.Level+1);
    end
    n = n + s.U * params.numfunctions(s.Level+1) *
        params.numright(s.Level+1) * params.numshears(s.Level+1);
    n = n + s.R * params.numfunctions(s.Level+1) *
        params.numshears(s.Level+1);
    n = n + params.numfunctions(s.Level+1) * (s.K +
        (params.numshears(s.Level+1)-1)/2);
    n = n + s.F;
    n = n + 1;

    s.N = n;

    r = GetShearletCorners(s, 0, params);
    s.Radius = max(sqrt(sum((r-repmat([s.X;s.Y],1,4)).^2, 1)));

end
```

```
function [P, c] = GetTransform(s, i, params)

    if nargin < 3
        params = Settings;
    end

    c = [s.X; s.Y];
    if isfield(s, 'K')
        a = params.xscale / params.yscale;
        P = [1, a*s.K; 0, 1] * [params.xscale^s.Level, 0; 0,
            params.yscale^s.Level];
    else
        P = [params.xscale^s.Level, 0; 0, params.yscale^s.Level];
    end;
    if s.Cone == 1, P = P * [0 1; 1 0]; end;

    if nargin > 1 && i > 0
        XS = length(params.wvfunctions(s.Level+1,s.F+1).xpts)-1;
        YS = length(params.scfunctions(s.Level+1,s.F+1).xpts)-1;

        Qi = [XS, 0; 0, YS];
        ci = [(mod(i-1,XS)+.5)/XS-.5; (floor((i-1)/XS)+.5)/YS-.5];
        c = P\ci+c;
        P = Qi*P;
    end

end
```

```
function p = GetShearletCorners(s, i, params)

    if nargin < 3
        params = Settings;
    end

    if nargin < 1
        p = [0, 1, 1, 0; 0, 0, 1, 1];
    else
        if nargin < 2
            i = 0;
        end
        p = [-.5, .5, .5, -.5; -.5, -.5, .5, .5];
        [P, c] = GetTransform(s, i, params);
        p = (P\p) + repmat(c, 1, size(p,2));
        if (s.Cone == 1), p = fliplr(p); end;
    end

end
```

A.3 Evaluating shearlets

The `PreparePoints` method takes care of all the transforming being done, as well as some work related to peroidicity. 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.

```
function [x, computefor] = PreparePoints(s, x, params)

    if nargin < 3
        params = Settings;
    end

    [P, c] = GetTransform(s, 0, params);
    x = P*(x-repmat(c, 1, size(x,2)));

    computefor = 1:size(x,2);

    if params.periodic
        dirs = P * [1 0; 0 1];
        if s.Cone == 1
            dirs = fliplr(dirs);
        end
        n_min = ceil((-x(2,:)-0.5)/dirs(2,2));
        n_max = floor((-x(2,:)+0.5)/dirs(2,2));

        computefor = setdiff(computefor, find(n_min>n_max));
        n = n_min;

        m_min = ceil((-x(1,:)-n*dirs(1,2)-0.5)/dirs(1,1));
        m_max = floor((-x(1,:)-n*dirs(1,2)+0.5)/dirs(1,1));

        computefor = setdiff(computefor, find(m_min>m_max));
        m = m_min;

        x = x + repmat(m,2,1).*repmat(dirs(:,1),1,size(x,2)) +
            repmat(n,2,1).*repmat(dirs(:,2),1,size(x,2));
    end

end
```

```
function out = EvaluateShearlet(s, x, params)

    if nargin < 3
        params = Settings;
    end

    xorig = x;
    [x, computefor] = PreparePoints(s, x, params);

    wv_xpts = params.wvfunctions(s.Level+1,s.F+1).xpts;
    wv_fvals = params.wvfunctions(s.Level+1,s.F+1).fvals;
    gm = @(x) inter(wv_xpts, wv_fvals, x);

    sc_xpts = params.scfunctions(s.Level+1,s.F+1).xpts;
    sc_fvals = params.scfunctions(s.Level+1,s.F+1).fvals;
    th = @(x) inter(sc_xpts, sc_fvals, x);

    out = zeros(1,size(x,2));
    out(computefor) = th(x(2,computefor)).*gm(x(1,computefor));
    out = 2^(-3*s.Level/2)*out;

    if isfield(params, 'glob')
        out = out .* params.glob(xorig(:,computefor));
    end

    function out = inter(xs, ys, x)
        out = interp1([min([xs,x])-1, xs, max([xs,x])+1], [0, ys, 0],
            x);
    end

end
```

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

    if nargin < 3
        params = Settings;
    end

    xorig = x;
    [x, computefor] = PreparePoints(s, x, params);

    wv_xpts = params.wvfunctions(s.Level+1,s.F+1).xpts;
    wv_fvals = params.wvfunctions(s.Level+1,s.F+1).fvals;
    gm = @(x) inter(wv_xpts, wv_fvals, x);
    gmd = @(x) interd(wv_xpts, wv_fvals, x);

    sc_xpts = params.scfunctions(s.Level+1,s.F+1).xpts;
    sc_fvals = params.scfunctions(s.Level+1,s.F+1).fvals;
    th = @(x) inter(sc_xpts, sc_fvals, x);
    thd = @(x) interd(sc_xpts, sc_fvals, x);

    [P, c] = GetTransform(s, 0, params);
    shld = zeros(2,size(x,2));
    shld(:,computefor) = [th(x(2,computefor)).*gmd(x(1,computefor));
       thd(x(2,computefor)).*gm(x(1,computefor))];
    shld = 2^(-3*s.Level/2)*P'*shld;

    if ~isfield(params, 'glob') || ~isfield(params, 'globd')
        out = shld;
    else
        shl = zeros(1,size(x,2));
        shl(computefor) = th(x(2,computefor)).*gm(x(1,computefor));
        shl = 2^(-3*s.Level/2)*shl;
        out = repmat(shl,2,1).*params.globd(xorig) +
            repmat(params.glob(xorig),2,1).*shld;
    end;

    function out = inter(xs, ys, x)
        out = interp1([min([xs,x])-1, xs, max([xs,x])+1], [0, ys, 0],
            x);
    end

    function out = interd(xs, ys, x)
        gpts = length(xs);
        vpts = length(x);

        loc = sum(repmat(x,gpts,1) > repmat(xs',1,vpts), 1);
        ys = [ys(1), ys, ys(end)];
        xs = [xs(1)-1, xs, xs(end)+1];

        out = (ys(loc+2)-ys(loc+1))./(xs(loc+2)-xs(loc+1));
    end

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.

```
function out = BLFKK(u, v, x, params)

    if nargin < 4
        params = Settings;
    end

    out = params.k(x).^2 .* EvaluateShearlet(u, x, params) .*
       EvaluateShearlet(v, x, params);

end
```

```
function out = BLFSS(u, v, x, params)

    if nargin < 4
        params = Settings;
    end

    out = sum(params.s(x) .* EvaluateShearletGradient(u, x, params), 1)
        .* sum(params.s(x) .* EvaluateShearletGradient(v, x, params),
        1);

end
```

```
function out = BLFSK(u, v, x, params)

    if nargin < 4
        params = Settings;
    end

    out = sum(params.s(x) .* EvaluateShearletGradient(u, x, params), 1)
        .* params.k(x) .* EvaluateShearlet(v, x, params);

end
```

```
function out = BLF(u, v, x, params)

    if nargin < 4
        params = Settings;
    end

    ux = EvaluateShearlet(u, x, params);
    vx = EvaluateShearlet(v, x, params);
    ud = EvaluateShearletGradient(u, x, params);
    vd = EvaluateShearletGradient(v, x, params);

    p = params;

    out = p.k(x).^2.*ux.*vx + sum(p.s(x).*ud, 1).*p.k(x).*vx +
        sum(p.s(x).*vd, 1).*p.k(x).*ux + sum(p.s(x).*ud,
        1).*sum(p.s(x).*vd, 1);

end
```

```
function out = LFK(v, x, params)

    if nargin < 3
        params = Settings;
    end

    out = EvaluateShearlet(v, x, params) .* params.f(x) .* params.k(x);

end
```

```
function out = LFS(v, x, params)

    if nargin < 3
        params = Settings;
    end

    out = sum(params.s(x) .* EvaluateShearletGradient(v, x, params), 1)
        .* params.f(x);

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 `GetIntersecton`, which actually computes the intersection between two shearlet (sub)supports, relying on the GPC library [5] 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).

```
function out = CheckPolygonIntersecton(p, q)

    if isempty(p) || isempty(q)
        out = 0;
        return;
    end

    cp = [p(:,end), p];
    cq = [q(:,end), q];

    % Loop through P
    for i = 2:size(cp,2)

        base_tmp = cp(:,i) - cp(:,i-1);
        base = [base_tmp(2); -base_tmp(1)];

        separator = 1;
        for j = 2:size(cq,2)
            if dot(cq(:,j) - cp(:,i), base) < 0
                separator = 0;
                break;
            end
        end

        if separator
            out = 0;
            return;
        end

    end

    for i = 2:size(cq,2)

        base_tmp = cq(:,i) - cq(:,i-1);
        base = [base_tmp(2); -base_tmp(1)];

        separator = 1;
        for j = 2:size(cp,2)
            if dot(cp(:,j) - cq(:,i), base) < 0
                separator = 0;
                break;
            end
        end

        if separator
            out = 0;
            return;
        end

    end

    out = 1;

end
```

```
function out = CheckShearletIntersection(u, v, params)

    if nargin < 3
        params = Settings;
    end

    p = GetShearletCorners(u, 0, params);
    q = GetShearletCorners(v, 0, params);

    Nvals = 0;
    Mvals = 0;

    if params.periodic
        deltax = v.X - u.X;
        deltay = v.Y - u.Y;
        rad = v.Radius + u.Radius;

        Nvals = ceil(-rad-deltax):floor(rad-deltax);
        Mvals = ceil(-rad-deltay):floor(rad-deltay);
    end

    for N = Nvals
    for M = Mvals
        if CheckPolygonIntersection(p, q + repmat([N;M],1,size(q,2)))
            out = 1;
            return;
        end
    end
    end

    out = 0;

end
```

```
function [np, p] = GetIntersection(u, i, v, j, params)

    if nargin < 5
        params = Settings;
    end

    if nargin < 4
        j = 0;
    end

    if nargin < 3
        v = [];
    end

    if nargin < 2
        i = 0;
    end

    sp = GetShearletCorners();
    S.x = sp(1,:); S.y = sp(2,:); S.hole = 0;
    U = BuildPolygon(u, i, params);

    R = PolygonClip(U, S, 1);

    if length(R) == 0
        np = 0;
        p = [];
        return;
    end

    if nargin > 2 && ~isempty(v)
        V = BuildPolygon(v, j, params);
        R = PolygonClip(R, V, 1);

        if length(R) == 0
            np = 0;
            p = [];
            return;
        else
            np = length(R);
            for k = 1:length(R)
                p(k).p = fliplr([R(k).x'; R(k).y']);
            end
            return;
        end
    else
```

```
        np = length(R);
        for k = 1:length(R)
            p(k).p = fliplr([R(k).x'; R(k).y']);
        end
        return;
    end

    function U = BuildPolygon(u, i, params)

        up = GetShearletCorners(u, i, params);
        U(1).x = up(1,:); U(1).y = up(2,:); U(1).hole = 0;

        if params.periodic
            [r, l, u, d] = CrossesBoundary(params, u, i);

            U = AddToPol(U, 1, -r, 0);
            U = AddToPol(U, 1, l, 0);
            U = AddToPol(U, 1, 0, -u);
            U = AddToPol(U, 1, 0, d);
            U = AddToPol(U, 1, -(r&u), -(r&u));
            U = AddToPol(U, 1, l&u, -(l&u));
            U = AddToPol(U, 1, -(r&d), r&d);
            U = AddToPol(U, 1, l&d, l&d);
        end

    end

    function pol = AddToPol(pol, ref, xadd, yadd)
        if (xadd ~= 0) | (yadd ~= 0)
            k = length(pol) + 1;
            pol(k).x = pol(ref).x + xadd;
            pol(k).y = pol(ref).y + yadd;
            pol(k).hole = pol(ref).hole;
        end
    end

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.

```
function out = SplitTriangle(trs, L)
```

```

    if L > 0

        out = zeros(6, 4*size(trs,2));
        i = 1;

        for tr = trs
            q = reshape(tr, 2, 3);
            out(:, i:i+3) = [reshape([q(:,1), avg(q(:,1),q(:,2)),
                avg(q(:,1),q(:,3))], 6, 1),...
                                reshape([q(:,2), avg(q(:,2),q(:,3)),
                                    avg(q(:,2),q(:,1))], 6, 1),...
                                reshape([q(:,3), avg(q(:,3),q(:,1)),
                                    avg(q(:,3),q(:,2))], 6, 1),...
                                reshape([avg(q(:,1),q(:,2)),
                                    avg(q(:,3),q(:,2)),
                                    avg(q(:,3),q(:,1))], 6, 1)];
            i = i + 4;
        end

        out = SplitTriangle(out, L-1);

    else

        out = trs;

    end

    function out = avg(a, b)
        out = 0.5 * (a+b);
    end

end
```

```
function r = BuildQuadRuleOnPolygon(p, L, params)

    if nargin < 3
        params = Settings;
    end

    r.x = [];
    r.w = [];

    if isempty(p)
        return;
    end

    idxs = [1, 2, size(p,2)];
    sw = 2; endidx = 1; startidx = 3;

    while true
        trs = SplitTriangle(reshape(p(:, idxs), 6, 1), L);

        for tr = trs
            temp = TransformQuadRule(params.qr, reshape(tr, 2, 3));
            if sum(temp.w) > 0
                r.x = [r.x, temp.x];
                r.w = [r.w, temp.w];
            end;
        end

        if (sw == 2)
            idxs = [idxs(2), size(p,2)-endidx, idxs(3)];
            endidx = endidx + 1;
            sw = 1;
        elseif (sw == 1)
            idxs = [idxs(1), startidx, idxs(2)];
            startidx = startidx + 1;
            sw = 2;
        end;

        if length(unique(idxs)) < 3,
            break;
        end;
    end;

end
```

```
function r = BuildDoubleQuadRule(u, v, params)

    if nargin < 3
        params = Settings;
    end

    if u.N == v.N
        r = BuildSingleQuadRule(u, 0, params);
    else
        r.x = [];
        r.w = [];

        numSectionsU =
            (length(params.wvfunctions(u.Level+1,u.F+1).xpts)-1) *
            (length(params.scfunctions(u.Level+1,u.F+1).xpts)-1);
        numSectionsV =
            (length(params.wvfunctions(v.Level+1,v.F+1).xpts)-1) *
            (length(params.scfunctions(v.Level+1,v.F+1).xpts)-1);

        for i = 1:numSectionsU
        for j = 1:numSectionsV

            [np,p] = GetIntersection(u, i, v, j, params);

            for k = 1:np
                rule = BuildQuadRuleOnPolygon(p(k).p, 0, params);
                r.x = [r.x, rule.x];
                r.w = [r.w, rule.w];
            end

        end; end;

    end

end
```

```
function r = BuildSingleQuadRule(v, L, params)

    if nargin < 3
        params = Settings;
    end

    r.x = [];
    r.w = [];

    if nargin < 2 || isempty(L)
        L = params.maxlevel - v.Level;
    end

    numSections = (length(params.wvfunctions(v.Level+1,v.F+1).xpts)-1)
        * (length(params.scfunctions(v.Level+1,v.F+1).xpts)-1);

    for i = 1:numSections

        [np,p] = GetIntersection(v, i, [], 0, params);

        for k = 1:np
            rule = BuildQuadRuleOnPolygon(p(k).p, L, params);
            r.x = [r.x, rule.x];
            r.w = [r.w, rule.w];
        end

    end

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.

```
function out = IntegrateBLF(blf, u, v, qr, params)

    if nargin < 5
        params = Settings;
    end

    out = sum(qr.w .* blf(u, v, qr.x, params));

end
```

```
function out = IntegrateLF(lf, v, qr, params)

    if nargin < 4
        params = Settings;
    end

    out = sum(qr.w .* lf(v, qr.x, params));

end
```

```
function out = EvaluateBLF(blf, u, v, params)

    if nargin < 4
        params = Settings;
    end

    if CheckShearletIntersection(u, v, params);

        [np,p] = GetIntersection(u, 0, v, 0, params);

        if np == 0
            out = 0;
            return;
        else
            qr = BuildDoubleQuadRule(u, v, params);
            if size(qr.x,2) > 0
                out = IntegrateBLF(blf, u, v, qr, params);
            else
                out = 0;
            end
        end

    else
        out = 0;
    end

end
```

```
function out = EvaluateLF(lf, v, params)

    if nargin < 3
        params = Settings;
    end

    if CheckPolygonIntersection(GetShearletCorners(),
       GetShearletCorners(v, 0, params))

        [np,p] = GetIntersection(v, 0, [], 0, params);

        if np == 0
            out = 0;
            return;
        else
            qr = BuildSingleQuadRule(v, [], params);
            if size(qr.x,2) > 0
                out = IntegrateLF(lf, v, qr, params);
            else
                out = 0;
            end
        end

    else
        out = 0;
    end

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 parallellization 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.

```
function ret = Stiffness(k, I, J, blf, sym, params)

    N = 10000;
    NNZ = 1000*N;

    persistent A;
    persistent nodes;
    persistent defined;

    if isa(k, 'char')
        if strcmp(k, 'clear')
            disp('Clearing...');
            clear A;
            clear nodes;
            clear defined;
            ret = 0;
            return;
        end
    end

    if ~exist('defined')
```

```matlab
        defined = [];
    end

    if (length(defined) < k) || (defined(k) == 0)
        disp('Allocating...');
        A{k} = spalloc(N, N, NNZ);
        nodes{k} = zeros(1, N);
        defined(k) = 1;
    end

    if nargin < 6
        params = Settings();
    end

    maxidx = max(max(I),max(J));
    if maxidx > length(nodes{k})
        nodes{k} = [nodes{k}, zeros(1, maxidx-length(nodes{k}))];
    end

    newI = find(nodes{k}(I) == 0);
    newJ = find(nodes{k}(J) == 0);
    newnodes = unique([I(newI), J(newJ)]);
    oldnodes = find(nodes{k} == 1);

    if size(newnodes, 1) == 0
        newnodes = [];
    end

    nums = 0;
    N = length(newnodes);
    tots = length(oldnodes)*N + N*(N+1)/2;
    disp(' ');
    if tots > 0
        progmeter(0, ['Computing ' num2str(tots) ' new elements of matrix ' num2str(k)]);
    end

    for i = newnodes,
        temph = zeros(1,length(oldnodes));
        tempv = zeros(length(oldnodes),1);
        parfor j = 1:length(oldnodes)
            temph(j) = EvaluateBLF(blf, GetShearlet(oldnodes(j), params), GetShearlet(i, params), params);
            if sym,
                tempv(j) = temph(j);
            else
                tempv(j) = EvaluateBLF(blf, GetShearlet(i, params), GetShearlet(oldnodes(j), params), params);
            end
        end

        A{k}(i,oldnodes) = temph;
        A{k}(oldnodes,i) = tempv;

        A{k}(i,i) = EvaluateBLF(blf, GetShearlet(i, params),
            GetShearlet(i, params), params);
        nums = nums + length(oldnodes) + 1; progmeter(nums/tots);

        nodes{k}(i) = 1;
        oldnodes = [oldnodes, i];
    end

    ret = A{k}(I,J);

    if tots > 0
        progmeter done
    end

end
```

```
function ret = Load(k, I, lf, params)

    N = 10000;

    persistent L;
    persistent nodes;
    persistent defined;

    if isa(k, 'char')
        if strcmp(k, 'clear')
            disp('Clearing...');
            clear L;
            clear nodes;
            clear defined;
            ret = 0;
            return;
        end
    end

    if ~exist('defined')
        defined = [];
    end

    if (length(defined) < k) || (defined(k) == 0)
        disp('Allocating...');
        L{k} = zeros(N, 1);
        nodes{k} = zeros(1, N);
        defined(k) = 1;
    end

    if nargin < 4
        params = Settings();
    end

    newI = find(nodes{k}(I) == 0);
    newnodes = I(newI);

    if size(newnodes, 1) == 0
        newnodes = [];
    end

    nums = 0;
    tempLk = L{k};
    tempnodesk = nodes{k};
    parfor i = newnodes,
        tempLk(i) = EvaluateLF(lf, GetShearlet(i), params);
        %nums = nums + 1;
        %progmeter(nums/length(newnodes));

        tempnodesk(i) = 1;
    end
    nodes{k} = tempnodesk;
    L{k} = tempLk;

    ret = L{k}(I);

end
```

References

[1] I. Babuška and J. M. Melenk. The partition of unity method. *International Journal for Numerical Methods in Engineering*, 40:727–758, 1997.

[2] Bernard Chazelle and David 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.

[3] S. Gottschalk. Separating axis theorem. Technical report, Department of Computer Science, UNC Chapel Hill, 1996.

[4] 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.

[5] A. Murta and T. Howard. See `http://www.cs.man.ac.uk/~toby/alan/software/`.

[6] 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.

[7] G. T. Toussaint. A simple linear algorithm for intersecting convex polygons. *The Visual Computer*, 1:118–123, 1985.