



Automation of the lifting factorisation of wavelet transforms

M. Maslen, P. Abbott *

Department of Physics, University of Western Australia, Nedlands, WA 6907, Australia

Abstract

Wavelets are sets of basis functions used in the analysis of signals and images. In contrast to Fourier analysis, wavelets have both spatial and frequency localization, making them useful for the analysis of sharply-varying or non-periodic signals. The *lifting scheme* for finding the discrete wavelet transform was demonstrated by Daubechies and Sweldens (1996). In particular, they showed that this method depends on the factorization of *polyphase matrices*, whose entries are Laurent polynomials, using the Euclidean algorithm extended to Laurent polynomials. Such factorization is not unique and hence there are multiple factorizations of the polyphase matrix. In this paper we outline a *Mathematica* program that finds all factorizations of such matrices by automating the Euclidean algorithm for Laurent polynomials. Polynomial reduction using Gröbner bases was also incorporated into the program so as to reduce the number of wavelet filter coefficients appearing in a given expression through use of the relations they satisfy, thus permitting exact symbolic factorizations for any polyphase matrix. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Wavelets; Lifting; Euclidean algorithm; Laurent polynomials; Gröbner bases; Polynomial reduction

AMS classification: 02.70; 07.05.K

PROGRAM SUMMARY

Title of program: LiftingFactorisation.nb

Version number: 1.0

Catalogue identifier: ADLE

Program Summary URL: <http://cpc.cs.qub.ac.uk/summaries/ADLE>

Program obtainable from: <ftp://ftp.physics.uwa.edu.au/pub/Wavelets/LiftingFactorisation.nb>

Programming language used: Mathematica 3.0

Platform: Any platform supporting Mathematica 3.0

Number of bytes in distributed program, including test data, etc.:
71 567

Distribution format: ASCII

Keywords: Wavelets, lifting, Euclidean algorithm, Laurent polynomials, Gröbner bases, polynomial reduction

Nature of physical problem

Spectral analysis and compression of signals or images.

Method of solution

Symbolic computer algebra is used to automate the Euclidean algorithm for Laurent polynomials [1] so as to factorize wavelet transforms yielding a sequence of simple arithmetic operations suitable for parallel, in-place, implementation [2].

* Corresponding author. E-mail: paul@physics.uwa.edu.au.

Limitations

The program requires that the Laurent polynomial quotients used in the algorithm have Laurent degree at most 1. The polyphase matrix must have unit determinant (though any polyphase matrix may be adjusted to satisfy this criterion [2]).

Unusual features

The program can find symbolic greatest common divisors (gcds) of two Laurent polynomials. Using Gröbner bases and polynomial reduction on the filter coefficients reduces the number of unknown coefficients appearing in expressions. There is also capability to

convert the lifting steps from mathematical notation to computer pseudocode, suitable for implementation in C or Fortran.

References

- [1] M.J. Maslen, Factoring Wavelet Transforms into Lifting Steps (University of Western Australia, 1997).
URL: <http://www.physics.uwa.edu.au/~paul/publications.html>.
- [2] I. Daubechies, W. Sweldens, J. Fourier Anal. Appl. 4 (1998) 247. URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#factor>

LONG WRITE-UP

1. Introduction

Scientists and engineers are familiar with Fourier analysis, which is an ideal tool for describing information such as waves, optical signals, and vibrations. It is the periodic nature of the Fourier basis which makes it so useful for these applications, for which the information is frequently oscillatory in nature. However, for highly non-periodic signals, the Fourier expansion requires many terms to achieve acceptable representation. In such cases an alternative is to use *wavelets*.

There are a variety of wavelet bases, each having particular characteristics such as symmetry, smoothness or orthogonality. As such they are far more versatile than Fourier analysis, and can be tailored to specific situations. Wavelets bases of *compact support* have excellent spatial localization (in contrast with the Fourier basis). Wavelets have been applied to areas such as geophysical survey analysis, image compression (FBI fingerprint database) [3], denoising, and encryption. Further details can be found in, for example, [1,4–8]. For a survey of applications of the wavelet transform to a wide range of physical fields – including astrophysics, turbulence, meteorology, plasma physics, atomic and solid state physics, and mathematical physics – see [9].

Wavelet research has proliferated over the last decade. Much of the relevant background theory has been developed, and the recent focus has been on applications and implementation techniques. One particularly powerful implementation method is that of lifting. This has been shown to be equivalent to the more traditional methods such as Mallat's algorithm and Fourier methods [10–13]. Lifting has several advantages over other methods. It is a very general technique, applicable to n -dimensional transforms and wavelets on manifolds. In some cases the method can yield integer-to-integer transforms [14] allowing for lossless compression. Computationally it allows a parallel, in-place implementation, which is faster by a factor of 2 for large datasets [2]. It reduces the implementation to a sequence of simple steps involving elementary algebraic operations, making it trivial to find the inverse transform.

A major recent development is the *factorization* of the lifting scheme [2]. This provides a simple method for deriving the sequence of lifting steps. Importantly, the factorization is fundamentally non-unique, so there exist several possible sequences of lifting steps for a given wavelet transform. This is an important feature as it allows a choice of which set of steps to use – some sets may be better suited to hardware implementation, numerical computation, or symmetric implementation, for example.

Two questions that arise from this factorization are (i) how many factorizations exist, and (ii) how does one find all factorizations. This paper addresses both of these issues, in the form of a *Mathematica* program that finds all gcds of two Laurent polynomials.

2. Mathematical preliminaries

We outline some of the background mathematical ideas relating to wavelets and the lifting scheme.

2.1. Laurent polynomials, finite filters, and z -transforms

A *Laurent polynomial* is an expression of the form

$$p(z) = \sum_{k=a}^b c_k z^{-k}.$$

We define the *degree* of a nonzero Laurent polynomial as $|p(z)| = b - a$, and the degree of the zero polynomial is defined to be $-\infty$.

A *finite filter* may be described as a set of coefficients $h = \{h_k : k \in \mathbb{Z}\}$, which are zero outside the finite range $k_b \leq k \leq k_e$. The z -transform of a finite filter is the Laurent polynomial whose coefficients are the filter coefficients, namely

$$h(z) = \sum_{k=k_b}^{k_e} h_k z^{-k}. \tag{1}$$

The limiting points k_e and k_b are arbitrary subject to the restriction that $k_e - k_b$ is the length of the filter. Equivalently, the z -transform is defined only up to a monomial factor.

2.2. The Euclidean algorithm

The *Euclidean algorithm* is a method for finding the greatest common divisor (gcd) of two integers. The *division algorithm* states that for any integers a and b , where $|a| > |b|$ (here $|\cdot|$ denotes absolute value) and $b \neq 0$, there exist integers q_1 and r_1 (the reason for the subscripts will become clear) such that $a = q_1 b + r_1$ with $0 \leq r_1 < b$.

The Euclidean algorithm can be used to find the gcd as follows. Firstly note that if $r_1 = 0$, then $\text{gcd}(a, b) = b$. If $r_1 \neq 0$, we see that any common divisor of b and r_1 is a divisor of a , and thus is a common divisor of a and b . Rewriting the above expression as $r_1 = a - q_1 b$, shows that any common factor of a and b is a factor of r_1 , and is thus a common factor of r_1 and b . We have shown that the common divisors of a and b are the same as those of r_1 and b , and hence $\text{gcd}(a, b) = \text{gcd}(r_1, b)$. The problem has thus been reduced to finding the gcd of the smaller pair of numbers (r_1, b) .

We may now apply the division algorithm to r_1 and b to get $b = q_2 r_1 + r_2$ with $0 \leq r_2 < r_1$, and repeating the argument above, we see that $\text{gcd}(r_1, b) = \text{gcd}(r_1, r_2)$. We iterate this process, each time obtaining a smaller remainder until eventually $r_i = 0$ for some i . We then have $r_{i-2} = q_i r_{i-1}$ and thus we have $\text{gcd}(a, b) = \text{gcd}(r_{i-2}, r_{i-1}) = r_{i-1}$. The algorithm has a simple computational implementation, in recursive form:

$$\begin{aligned} a_i &\leftarrow b_{i-1}, \\ b_i &\leftarrow a_{i-1} - a_i q_i. \end{aligned} \tag{2}$$

The Euclidean algorithm can be generalized to the Laurent polynomials. The procedure is similar to that shown above, with the generalization being that we require that $|r| < |b|$, where $|\cdot|$ is the Laurent degree.

2.3. An example

An important point is that the quotients are *not* necessarily unique in the Euclidean algorithm. That is, although the gcd is unique up to an invertible (monomial) factor, the *quotients* can be chosen in several nontrivially

distinct ways. To illustrate this we give an example of the Euclidean algorithm for Laurent polynomials. Let $a = 1/z + 3 - 2z$, and $b = -6/z^2 + 3/z$. We wish to find $\gcd(a, b)$. Note that $|a| = 1 - (-1) = 2$, while $|b| = -1 - (-2) = 1$. We set up the first step of the Euclidean algorithm:

$$\frac{1}{z} + 3 - 2z = q_1 \left(\frac{-6}{z^2} + \frac{3}{z} \right) + r_1.$$

Since the remainder must have degree less than that of b , it must be monomial. Rearranging this equation for r_1 , we get

$$r_1 = \left(\frac{1}{z} + 3 - 2z \right) - q_1 \left(\frac{-6}{z^2} + \frac{3}{z} \right).$$

We must choose q_1 so that r_1 will be monomial. One way to do this is to choose q_1 so that the highest monomials in each term in this expression are equal, and will thus cancel. This means that q_1 will need to be of the form $z^2(c + d/z)$ in order for the terms to have the same highest exponents. The parameters c and d are then found by the method of undetermined coefficients. Substituting, we find that the first remainder has the form

$$z(-3c - 2) + 6c - 3d + \frac{6d + 1}{z} + 3,$$

and, setting the coefficients of the z and constant terms to zero, we get $c = -2/3$, $d = -1/3$, so our first quotient is $-2z^2/3 - z/3$, and the first remainder is $-1/z$. We now apply the Euclidean algorithm to this remainder and b to get

$$b = q_2 r_1 + r_2 \Rightarrow r_2 = b - q_2 r_1.$$

Once again, the remainder r_2 must have degree less than that of r_1 . Since r_1 has degree zero, it follows that r_2 must be zero (recall that the degree of the zero polynomial is defined as $-\infty$). As before, we choose a quotient form so that both powers are matched, and find that the second remainder has the form

$$\frac{c + 3}{z} + \frac{d - 6}{z^2}.$$

We can set both terms, and thus the remainder, to zero to get $c = -3$, $d = 6$, so the second quotient is $6/z - 3$. Alternatively, we could have decided to eliminate the lowest two powers from the original expression, or even the lowest and the highest power. This gives three possible factorizations, each with different quotients. The gcd here is defined only up to a power of z : the gcd is ‘greatest’ in the sense that it has greatest possible Laurent degree. In this case, we found that the gcd was $-1/z$ (the last nonzero remainder). The other factorization branches lead to a monomial gcd also. In this case the original Laurent polynomial a can be retrieved via $a = q_1 b + \kappa$, where κ is the gcd, and the reader can easily verify that the three factorizations below are all valid:

$$a = \left(-\frac{2z^2}{3} - \frac{z}{3} \right) b - \frac{1}{z}, \quad a = \left(-\frac{2z^2}{3} - \frac{z}{6} \right) b - \frac{1}{2}, \quad a = \left(-\frac{7z^2}{12} - \frac{z}{6} \right) b - \frac{z}{4}.$$

The gcd is the same for each factorization up to a monomial factor, but the first quotients differ nontrivially. The last quotients differ only by monomial factors, because at this stage we have no freedom over which powers to eliminate.

Note that multiplying a by z and b by z^2 converts these Laurent polynomials into (ordinary) polynomials in z . Polynomial division then gives

$$(az) = -2z^2 + 3z + 1 = \left(-\frac{2z}{3} - \frac{1}{3} \right) (bz^2) - 1,$$

which is equivalent to the first factorization above, providing a simple computational check. Similarly, the last factorization can be obtained by dividing a/z by bz (since these are ordinary polynomials in $1/z$), i.e.,

$$\frac{a}{z} = -2 + \frac{3}{z} + \frac{1}{z^2} = \left(-\frac{7}{12} - \frac{1}{6z} \right) (bz) - \frac{1}{4}.$$

Furthermore, there are fast algorithms for polynomial division and these can be used for these computations. However, the second factorization cannot be obtained in this way.

Later, we will factorize certain 2×2 matrices whose entries are Laurent polynomials (specifically z -transforms as defined above) by executing the Euclidean algorithm on two of the entries. The factors are matrices whose entries are determined by the quotients resulting from the algorithm. Thus, using the different sets of quotients, we can obtain several different matrix factorizations.

2.4. Gröbner bases and polynomial reduction

The Euclidean algorithm can yield complicated algebraic expressions when carried out on z -transforms with symbolic filter coefficients. It is desirable to reduce such expressions as much as possible, using any relations that exist between the filter coefficients. A useful way of carrying out such simplifications is to use *Gröbner bases* and *polynomial reduction*. *Mathematica* has built-in functions to generate Gröbner bases and to carry out polynomial reduction.

The formal definition of a Gröbner basis is somewhat involved, making use of several concepts in the theory of rings and multivariate polynomials (the interested reader is referred to [1]). The essential point is that Gröbner bases encode the information contained in the constraints on the filter coefficients, which can then be exploited to simplify factorizations by the technique of polynomial reduction. The utility of Gröbner bases may be demonstrated by considering an example. In the general case of DN orthogonal wavelets (D denotes Daubechies and N here denotes the number of non-zero filter coefficients), the following relations are satisfied by the filter coefficients h_k [5]:

$$\sum_k h_k = \sqrt{2}, \quad \sum_k h_k h_{k-2m} = 2\delta_{0,m} \quad \text{for } m = 0, 1, 2, \dots, \frac{N}{2} - 1,$$

$$\sum_k (-1)^k k^{p-1} h_k = 0, \quad \text{where } p = 1, 2, \dots, \frac{N}{2}.$$

These conditions arise from applying certain approximation and orthonormality constraints. The equations may be solved for the coefficients of a given DN basis. In general the above conditions give $N + 1$ equations in N unknowns, however the system of equations is consistent; square normalization can be derived from the other conditions. They can be solved exactly when $N \leq 6$, but for $N > 6$ they must be solved numerically. Solving for $N = 6$ gives

$$h_0 = \frac{\sqrt{2}}{32} (1 + \sqrt{10} + \sqrt{5 + 2\sqrt{10}}), \quad h_1 = \frac{\sqrt{2}}{32} (5 + \sqrt{10} + 3\sqrt{5 + 2\sqrt{10}}),$$

$$h_2 = \frac{\sqrt{2}}{32} (10 - 2\sqrt{10} + 2\sqrt{5 + 2\sqrt{10}}), \quad h_3 = \frac{\sqrt{2}}{32} (10 - 2\sqrt{10} - 2\sqrt{5 + 2\sqrt{10}}),$$

$$h_4 = \frac{\sqrt{2}}{32} (5 + \sqrt{10} - 3\sqrt{5 + 2\sqrt{10}}), \quad h_5 = \frac{\sqrt{2}}{32} (1 + \sqrt{10} - \sqrt{5 + 2\sqrt{10}}).$$

A Gröbner basis is a set of polynomial expressions which are constructed using the zero conditions satisfied by the coefficients. An important feature is that the resulting Gröbner basis always has exactly the same collection of common roots as the original set of polynomials and is a canonical form from which many properties can conveniently be deduced. *Mathematica* has a built-in command for computing a Gröbner basis:

$$\text{In}[1] := \mathbf{g} = \text{GroebnerBasis}\left[\left\{\sum_{i=0}^5 h_i - \sqrt{2}, \sum_{i=0}^5 (-1)^i h_i, \sum_{i=0}^5 i(-1)^i h_i, \sum_{i=0}^5 i^2 (-1)^i h_i, \sum_{i=0}^3 h_i h_{i+2}, \sum_{i=0}^1 h_i h_{i+4}\right\}, \text{Table}[h_i, \{1, 0, 5\}]\right]$$

$$\begin{aligned} Out[1] = & \{65536h_5^4 - 8192\sqrt{2}h_5^3 - 3072h_5^2 - 96\sqrt{2}h_5 + 9, \\ & - 256h_5^3 + 32\sqrt{2}h_5^2 + 6h_5 + 3h_4, \\ & - 4096h_5^3 + 512\sqrt{2}h_5^2 + 192h_5 + 24h_3 - 3\sqrt{2}, \\ & 8h_2 + 16h_5 - 3\sqrt{2}, 4096h_5^3 - 512\sqrt{2}h_5^2 - 168h_5 + 24h_1 - 9\sqrt{2}, \\ & 2048h_5^3 - 256\sqrt{2}h_5^2 - 96h_5 + 24h_0 - 3\sqrt{2}\}. \end{aligned}$$

This basis will eliminate $h_0, h_1, h_2, h_3,$ and h_4 from any polynomial expression, replacing it with terms in h_5 . Now consider the following expression:

$$In[2] := \mathbf{expr} = \mathbf{h_1 h_5 + h_0 h_3 + h_4 h_2};$$

This can be reduced to an expression involving only a single coefficient, here h_5 , using polynomial reduction with the above Gröbner basis:

$$In[3] := \mathbf{reduced} = \mathbf{PolynomialReduce[expr, g, Table[h_i, \{i, 0, 5\}]]/Last}$$

$$Out[3] = \frac{64}{3}\sqrt{2}h_5^3 - \frac{25h_5^2}{3} - \frac{5h_5}{4\sqrt{2}} - \frac{1}{64}.$$

Observe that the form of these filter coefficients makes it difficult to give this reduced form of $expr$ using elementary algebra. Note also that we do not need to know the exact values of all of the coefficients, only their defining relationships. For example, it can be shown that finding the exact form of the D8 coefficients is equivalent to solving an irreducible sixth degree polynomial. However, polynomial reduction can still be carried out on the D8 coefficients, giving rise to *exact* expressions containing only *one* coefficient. This may be more satisfactory for numerical implementations, as we only need to substitute one numerical value into exact expressions.

2.5. Mallat's algorithm

We briefly describe here an established algorithm for computing the discrete wavelet transform. For a sampled signal, considered for simplicity to be of length $l = 2^n$ (this is easily generalized), we can use the relations satisfied by the wavelets to compute the wavelet expansion coefficients. We define matrices called *quadrature mirror filters* (QMF) in terms of the filter coefficients, and multiply by our data vector to produce a step of the wavelet transform. After each matrix multiplication, half of the vector is 'left alone' – it is one component of the wavelet transform. Before the next step, a new QMF matrix of half the original dimensions is constructed. Eventually we reach a point where we are operating on a two component vector (only if $l = 2^n$), and the result of this step is the last component of the discrete wavelet transform. This is called *Mallat's algorithm* [5].

More precisely, we denote our sampled signal by the vector $s^{[n]}$; these sampled values correspond to the highest resolution level. With the coefficients $h_k (g_k = (-1)^k h_{1-k})$ forming a low(high)-pass filter, we define the QMF matrices as

$$(H)_{k,l} = h_{l-2k}, \quad (G)_{k,l} = g_{l-2k}. \quad (3)$$

Then we have the relations;

$$s^{[j]} = Hs^{[j+1]}, \quad d^{[j]} = Gs^{[j+1]}, \quad (4)$$

where the vectors $s^{[j]}$ and $d^{[j]}$ are the 'signal' and 'difference' terms at resolution j . In other words, the components of these vectors are the coefficients in the expansion of a signal into the wavelet basis. This defines a step of the discrete wavelet transform in terms of multiplication by H and G , the QMF matrices. The (in the $l = 2^n$ case) one-component vector $s^{[0]}$ is the average or 'DC' term of our signal. After the first step the QMF matrices are redefined at half the dimensions, and we carry out the step (4) on the signal term found at the previous level. Thus we are able to find the expansion coefficients at *all* levels beginning with our sampled values $s^{[n]}$.

As a simple example of this, consider an explicit matrix implementation of the D4 wavelet transform. It is, of course, highly inefficient to implement Mallat’s algorithm using matrix multiplication: this implementation is presented here only to introduce some notation and indicate the algorithm’s inherent parallelizability. For an initial data vector $s^{[3]} = \{s_0, d_0, s_1, d_1, s_2, d_2, s_3, d_3\}$ (the reason for the artificial structure of this vector will become clear later), the first step,

$$\begin{pmatrix} s^{[2]} \\ d^{[2]} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_0 & h_1 & h_2 & h_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 \\ h_2 & h_3 & 0 & 0 & 0 & 0 & h_0 & h_1 \\ g_2 & g_3 & 0 & 0 & 0 & 0 & g_0 & g_1 \\ g_0 & g_1 & g_2 & g_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & g_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ d_0 \\ s_1 \\ d_1 \\ s_2 \\ d_2 \\ s_3 \\ d_3 \end{pmatrix},$$

can be split into the parallel operations involving only h ;

$$s^{[2]} \leftarrow \begin{pmatrix} h_0 & h_2 & 0 & 0 \\ 0 & h_0 & h_2 & 0 \\ 0 & 0 & h_0 & h_2 \\ h_2 & 0 & 0 & h_0 \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} h_1 & h_3 & 0 & 0 \\ 0 & h_1 & h_3 & 0 \\ 0 & 0 & h_1 & h_3 \\ h_3 & 0 & 0 & h_1 \end{pmatrix} \cdot \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix}, \tag{5}$$

and g :

$$d^{[2]} \leftarrow \begin{pmatrix} g_2 & 0 & 0 & g_0 \\ g_0 & g_2 & 0 & 0 \\ 0 & g_0 & g_2 & 0 \\ 0 & 0 & g_0 & g_2 \end{pmatrix} \cdot \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} + \begin{pmatrix} g_3 & 0 & 0 & g_1 \\ g_1 & g_3 & 0 & 0 \\ 0 & g_1 & g_3 & 0 \\ 0 & 0 & g_1 & g_3 \end{pmatrix} \cdot \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix}. \tag{6}$$

The *lifting scheme* described in §3 presents an alternative, highly efficient, method for computing the wavelet transform – only involving simple multiplication and addition operations – which is faster than Mallat’s algorithm by a factor of 2 for large datasets [2]. Moreover, lifting is more general, applicable to n -dimensional transforms and wavelets on manifolds.

3. Lifting

In this section we outline the general theory of the *lifting scheme*. Further details can be found in [2], and broader information in [10–13]. The lifting scheme has been shown to be equivalent to the traditional method of implementing the wavelet transform via the QMF matrices [2]. It has several advantages over Mallat’s algorithm, as will be shown below. It turns out that the procedure is equivalent to the factorization of a matrix whose components are related to the z -transform of the filter. This matrix factorization is non-unique, and we can find all factorizations through automation of the Euclidean algorithm for Laurent polynomials.

3.1. Polyphase representations

We define the *polyphase representation* of a filter;

$$h(z) = h_e(z^2) + z^{-1}h_o(z^2), \tag{7}$$

where h is the z -transform of the filter as defined in Eq. (1). Here h_e (h_o) is the *even* (*odd*) *polyphase component* of h , defined by

$$h_e(z^2) = \frac{h(z) + h(-z)}{2}, \quad h_o(z^2) = \frac{h(z) - h(-z)}{2z^{-1}}. \tag{8}$$

We see that h_e and h_o contain the even and odd filter coefficients, respectively. We now define the *polyphase matrix*:

$$P(z) = \begin{pmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{pmatrix}, \quad (9)$$

where g_e and g_o are defined in terms of the high pass filter g analogously to (7). Note that the determinant of this matrix is always monomial [2]. The program requires it to be unity, but any polyphase matrix can be suitably adjusted to satisfy this requirement [2]. Here z is not a value to be substituted, but a symbolic entity representing a data shift. In general for a filter f , we have the rule:

$$z^m f_l \rightarrow f_{l-m}, \quad (10)$$

where the direction of the shift is a choice of convention.

The polyphase matrix provides an alternative method of executing the lifting scheme. For a data vector $s^{[j+1]}$, we begin by performing the *lazy wavelet transform*, which simply involves subsampling the data at even and odd locations. That is;

$$\begin{aligned} s_l^{[j]} &\leftarrow s_{2l}^{[j+1]}, \\ d_l^{[j]} &\leftarrow s_{2l+1}^{[j+1]}. \end{aligned} \quad (11)$$

In analogy with Mallat's algorithm, the vector $(s^{[j]}, d^{[j]})$ then premultiplies the polyphase matrix to give one stage of the discrete wavelet transform, bearing in mind that the components $s^{[j]}$ and $d^{[j]}$ are *themselves* vectors, and the polyphase matrix elements are Laurent polynomials containing the z -operator (shift). That is, a general step may be written as

$$(s^{[j]}, d^{[j]}) \leftarrow (s^{[j+1]}, d^{[j+1]}) \cdot \begin{pmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{pmatrix}. \quad (12)$$

The result of the multiplication is the signal and detail at the next level, and the next step is to repeat the lazy wavelet transform (11) on $s^{[j]}$ and proceed as in (12). The length of $s^{[j]}$ and $d^{[j]}$ is reduced by a factor of 2 at each step, and thus the algorithm must always terminate after n steps.

3.2. The factorization of the polyphase matrix

The factorization of the polyphase matrix is obtained by using Theorem 7 of [2]. We first apply the Euclidean algorithm to divide $h_e(z)$ by $h_o(z)$. That is, we start by setting $a_0 = h_e(z)$, $b_0 = h_o(z)$, and then iterate the scheme as in (2),

$$\begin{aligned} a_i &= b_{i-1}, \\ b_i &= a_{i-1} - a_i q_i, \end{aligned}$$

choosing quotients q_i such that the degrees of the remainders b_i are reduced at each step. The factorization of the polyphase matrix is given in terms of these quotients by

$$P(z) = \left(\prod_{i=1}^{(n/2)-1} \begin{pmatrix} 1 & q_{2i-1}(z) \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ q_{2i}(z) & 1 \end{pmatrix} \right) \cdot \begin{pmatrix} 1 & \kappa^2 s(z) \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \kappa & 0 \\ 0 & 1/\kappa \end{pmatrix}, \quad (13)$$

where κ is a constant *scaling factor* which is simply the gcd, and $s(z)$ is a *lifting term*, found by comparison of the factorized and unfactorized forms of the polyphase matrix. We refer to the matrices in the initial product as *quotient matrices*, and to the last two matrices as the *lifting matrix* and the *scaling matrix*, respectively.

The alternating upper and lower triangular structure of this factorization is what allows an ‘in-place’ implementation of the wavelet transform. Consider the effect of premultiplication of $P(z)$ in *factorized* form by a two-element row vector $(s^{[j]}, d^{[j]})$ as in (12). The first factor here, $\begin{pmatrix} 1 & q_1(z) \\ 0 & 1 \end{pmatrix}$, is upper triangular with diagonal entries equal to 1. The *lifting step* corresponding to this matrix factor is thus

$$(s^{[j]}, d^{[j]}) \cdot \begin{pmatrix} 1 & q_1(z) \\ 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} s^{[j]} \leftarrow s^{[j]} & \\ d^{[j]} \leftarrow d^{[j]} + q_1(z)s^{[j]} & \end{pmatrix}. \tag{14}$$

Similar steps result from subsequent matrices in the factorization. This illustrates the fact that the lifting method is in-place and trivially invertible because

$$\begin{pmatrix} 1 & x \\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & -x \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 0 \\ x & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 \\ -x & 1 \end{pmatrix}.$$

Note that in general the quotients contain powers of z , which will give rise to shifting of the lists. The lifting scheme can thus be written as a sequence of steps of the form (14) with alternating s and d symbols, followed by the scaling step. The initial values of s and d are the results of the lazy wavelet transform as in (11).

When we reach the end of the factorization, that is, when the scaling steps have been executed, we record the final values of $s^{[j]}$ and $d^{[j]}$. As before, $d^{[j]}$ represents the high frequency component at scale j ; this is one component of the wavelet transform. We conduct the lazy wavelet transform on the result of scaling $s^{[j]}$, and follow the same lifting steps as before. These ideas are illustrated by the examples in §5.

The scaling matrix, $\begin{pmatrix} \kappa & 0 \\ 0 & 1/\kappa \end{pmatrix}$, is implemented as $s^{[j]} \leftarrow \kappa s^{[j]}$ and $d^{[j]} \leftarrow 1/\kappa d^{[j]}$. For implementation purposes it is preferable to have a factorization with *constant* gcd. The gcd is always monomial [2], and, for a general gcd, we can factorize the scaling matrix into lifting steps followed by a constant scaling matrix. To see this, note that

$$\begin{aligned} \begin{pmatrix} x & 0 \\ 0 & 1/x \end{pmatrix} &= \begin{pmatrix} 1 & x-x^2 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ -1/x & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & x-1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \\ &\equiv \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1-1/x \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ x & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1/x^2-1/x \\ 0 & 1 \end{pmatrix}, \end{aligned} \tag{15}$$

where x represents $z^{\pm 1}$ – only in this case do the matrices in these factorizations have degree 1 entries. So for an arbitrary gcd, αx^m , $m > 0$, we could factorize the scaling matrix as

$$\begin{aligned} \begin{pmatrix} \alpha x^m & 0 \\ 0 & 1/(\alpha x^m) \end{pmatrix} &= \begin{pmatrix} x & 0 \\ 0 & 1/x \end{pmatrix}^m \cdot \begin{pmatrix} \alpha & 0 \\ 0 & 1/\alpha \end{pmatrix} \\ &= \left(\begin{pmatrix} 1 & x-x^2 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ -1/x & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & x-1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \right)^m \begin{pmatrix} \alpha & 0 \\ 0 & 1/\alpha \end{pmatrix}, \end{aligned}$$

and still have alternating upper and lower triangular matrices with degree 1 entries followed by a constant scaling matrix. Similarly we could factorize the matrix using the second elementary factorization in (15), or indeed a combination of the two. Hence there are 2^m ways of factorizing the scaling matrix using these elementary factorizations, quite apart from the fact that we can choose the quotients in different ways.

3.3. The number of factorizations

It is important to consider the number of possible factorizations of a given wavelet transform. This is equivalent to the question of how many factorizations exist for two Laurent polynomials. In the following analysis, we only consider quotients of degree at most 1.

Suppose a_0 and b_0 are our starting polynomials. We first consider the case where $|a_0| = |b_0| + 1$. Consider the first step of the Euclidean algorithm as in (2),

$$\begin{aligned} a_1 &= b_0, \\ b_1 &= a_0 - a_1 q_1. \end{aligned} \tag{16}$$

The aim is to choose q_1 so that the degree of b is reduced (i.e., so $|b_1| < |b_0|$). Since we are using degree 1 quotients, we have that $|b_0 q_1| = |b_0| + |q_1| = |a_0|$, that is, a_0 and $b_0 q_1$ have the *same* number of terms. We must choose the quotient so that the highest (or equivalently the lowest) powers of a_0 and $b_0 q_1$ are the same. If this were not so, there would be extra powers introduced by the subtraction, and we could not reduce the degree. Since a generic degree 1 quotient has the form $z^n(c + d/z)$, this is simply a matter of choosing n appropriately.

In order to reduce the degree, we may choose to eliminate the two highest or two lowest powers, or each extreme power, by choice of the quotient coefficients c and d . Clearly there is no point eliminating intermediate terms, since this will have no effect on the degree. Thus we have three possibilities at this step. After this step, we set $a_1 = b_0$ and proceed similarly to (16) to find b_2 . The degrees of a and b have each been reduced by one, so we proceed as in the first step. Thus we have three possibilities at each step *except* the last one. This is because at the last step, a has degree 1, and all options are *equivalent* – there are only two terms that can be eliminated. Since we reduce the degree of b by 1 at each step, the number of steps in the Euclidean algorithm is $|b| + 1$, and hence the number of factorizations in this case is $3^{|b|}$.

Now let us consider the case where $|a| = |b|$. Considering again (16), we see that the degree can be reduced if q_1 has degree 0 (i.e., is monomial) under certain circumstances. We can cover all cases by defining a generic quotient as before. This time, however, $b_0 q_1$ has one more term than a_0 . We can still eliminate each extreme power, but this time we must choose whether we want to match the highest *or* lowest powers in a_0 and $b_0 q_1$ – these choices are no longer equivalent. We eliminate the highest two powers or lowest two powers as before. The degree 0 quotients arise when we set c or d to zero when eliminating terms. Hence there are now *four* possible elimination options at the first step. After the first step, we set $a_1 = b_0$ as above. This time, however, the degree of b has been reduced by 1, but the degree of a has *not* changed. Thus we are in the situation where $|a| = |b| + 1$, and we can proceed as above. The number of factorizations in this case is thus $4 \times 3^{|b|-1}$.

If a and b have degrees differing by more than 1, we *cannot* execute the Euclidean algorithm using only quotients of degree at most 1. To see this, consider again the first step (16), and suppose that $|a_0| > |b_0| + 1$. For degree 1 quotients, we can cancel two terms from a_0 , so $|b_1| = |a_0| - 2 > |b_0| - 1$, that is $|b_1| \geq |b_0|$ (since we are dealing with positive integers) so we have *not* reduced the degree of b . This is not a great limitation, since we are not dealing with *arbitrary* Laurent polynomials, but *even and odd components of polyphase filters*. Usually the degrees of the even and odd components will have degrees differing by 0 or 1. The only exception would be if one of the filter coefficients in the range $k_b < k < k_e$ was zero.

Note that we do not count any extra factorizations introduced by the non-unique factorization of the scaling matrix when the gcd is a non-constant monomial; these are not fundamentally important or enlightening.

4. Description of the program

We turn now from the general theory of lifting to the automation of polyphase factorizations and lifting using *Mathematica*.

4.1. Overview

The program, consisting of two broad parts, automates the factorization of the polyphase matrix, and can find all factorizations. The first part is an implementation of the Euclidean algorithm for Laurent polynomials, and the

second part is concerned with transforming the results into a sequence of lifting steps. The Euclidean algorithm part takes two Laurent polynomials, and runs the Euclidean algorithm, thus finding the quotients necessary to construct the matrix factorization (13). All the factorizations are found this way, although the program restricts the form of the quotients to have degree one. As indicated in §3.3, this is not a great loss of generality. It is noted in [2] that quotients of degree 1 are more favorable for hardware implementations than higher degree quotients.

The task of transforming the result of the Euclidean algorithm into a sequence of lifting steps is divided into smaller parts again. The first part converts to the matrix factorization (13), which involves arranging the quotients and gcd into matrix factors, and also finding the lifting term $s(z)$. This factorization in itself is not the final aim, although it is helpful to be able to see what the factorization looks like. The next part converts the matrix factorization into the sequence of lifting steps suitable for hardware or software implementation. This is done by simply carrying out the matrix multiplication of the symbolic vectors (s, d) by each matrix factor. The inverse of a given transform is also easily found. The final part converts these steps into *Mathematica* input. Implementing the factorization results permits comparison between the package and the results of different factorizations of the same transform.

The following subsections outline the main components of the program, and §5 gives several examples of its use.

4.2. *SingleIteration* module

This module simply takes as arguments the initial a and b on which to execute a step of the Euclidean algorithm, along with an elimination option. Its output is the quotient, the remainder, and the second Laurent polynomial (to be used in the next step). This module may be called repeatedly by a while loop whose stopping condition is that the remainder is identically zero. A generic quotient of degree one is defined as $z^n(c + d/z)$, where n is determined by matching extreme powers. This is substituted to give the form of the remainder, and c and d are found by the method of undetermined coefficients. Effectively these parameters are determined by the elimination option. Based on the analysis of §3.3, the four possible elimination options are:

EliminateEndsMatchHigh: This option means that the highest and lowest powers of the expression are to be eliminated, but the highest powers are to be matched by suitable choice of n .

EliminateEndsMatchLow: Again the extreme power terms will be eliminated, but this time the lowest two powers will be matched.

EliminateLow: In this case the lowest two power terms will be matched and eliminated.

EliminateHigh: Similarly, the highest two power terms will be matched and eliminated.

As indicated in §3.3, the first two options are frequently equivalent. However, since these options cover all cases, we do not need to test when the distinction is actually necessary.

4.3. *FindAll* module

This module generates every possible elimination branch, and runs *SingleIteration* on each of them until a zero remainder is found. It takes as input the two Laurent polynomials to be factorized and an optional simplification or transformation rule (see below). Note that at the last stage of the algorithm, one is dealing with a remainder which is monomial, and so all elimination options are equivalent. This is accounted for in the module, and saves three *Solve* operations occurring at each of a large number of factor branch nodes.

The structure of the module is such that it generates all possible choices that could be made at the first stage, solves for each choice, and stores the result. This is repeated at the next level, and so on, keeping the results at every stage. In this way, a ‘tree’ of factor branches is found. It is important to be aware of which quotients from different levels may be combined to form a branch of quotients. That is, since we are simply generating the lists of quotients at each level, these need to be arranged into a set of branch structures.

In summary, the process occurs in such a way that every possible equation is solved *once* and stored, and then recombined appropriately. The output is a list, each element of which corresponds to one factorization. The

elements themselves have a substructure – their first element is the list of quotients, and their second (and last) element is the gcd.

Neither of the above modules need be directly called by the user. The user calls the *EuclideanFactorisation* module, which checks which elimination option has been specified, and delegates the task to one of the above modules as appropriate. The module takes as input the starting Laurent polynomials (in a 2-element list), along with an elimination rule. The elimination rule is of the form *EliminationBranch* \rightarrow $\{opt1, opt2, \dots\}$, where $\{opt1, opt2, \dots\}$ is a list of elimination options if the user wishes to find a specific factorization, or simply *EliminationBranch* \rightarrow *All* if the user wants to find every factorization. Note that the user can specify an optional third argument to simplify or transform the results in some way. For example, this is how the Gröbner reduction is done, and if the user wants numeric factorizations, then *Chop* (which sets to zero any quantity less than 10^{-10}) is a suitable argument. If no third argument is given then it defaults to *Simplify*.

4.4. *PolynomialReduction* and *DaubechiesZeros* modules

PolynomialReduction takes as arguments the expression to be reduced and the filter coefficients along with the zero conditions they satisfy. It generates a Gröbner basis by calling the built-in *GroebnerBasis* command, and this is dynamically stored to avoid recomputation. It then simply uses the built-in *PolynomialReduce* command with the Gröbner basis. The Gröbner bases for *DN* can be computed using the *DaubechiesZeros* command, which automatically generates the zero conditions satisfied by Daubechies coefficients.

4.5. *PolyphaseFactorisation* module

This module gives the factorization of a polyphase matrix, taking as arguments the polyphase matrix, a factorization as found by the *EuclideanFactorisation* module, and an optional simplification function. It returns the matrix factorization as in (13). The given polyphase matrix must have determinant 1, though any matrix can be suitably modified in order to satisfy this criterion [2].

Note that optional simplification rules can be entered for this module as well as in the *EuclideanFactorisation* module. If one wishes to have a matrix factorization simplified by polynomial reduction, for example, the rule *must* be re-entered when calling this module. Although the quotients were simplified during the Euclidean algorithm, when finding the *matrix* factorization we need the lifting term $s(z)$. This depends on the *unfactorized* polyphase matrix, which has not undergone any simplification.

The module first checks to see if the number of steps required is odd, and if so, it prepends a zero to the quotient list found in the Euclidean algorithm. This will effectively produce an identity matrix as the first factor, and so the only effect this step has is to cause the following matrix factors to be transposed. This produces results consistent with those from [2], and is a valid factorization of the polyphase matrix. In fact it is equivalent to changing the order of the initial Laurent polynomials and setting the first quotient to zero. The lifting term $s(z)$ is found by equating the factorized and unfactorized forms of the polyphase matrix and solving appropriately. Note also that the module uses s to describe $\kappa^2 s(z)$.

The factorization is then given by constructing the appropriate matrices with the quotients, the lifting term, and the scaling matrix which is determined by the gcd. Any identity matrices in the factorization are discarded, and the placeholder command *CenterDot* is applied to those remaining to prevent them being multiplied during output.

If the factorization has a non-constant gcd, the user can apply the function *ScalingTransform* to κ to replace the scaling matrix. The user can also specify a list according to the sequence of elementary factorizations (15) they wish to use. For example, if the gcd was z^2 , and the user wished only to use the second of the factorizations (15), they would provide the list $\{2, 2\}$. If they wished to use the second followed by the first they would specify $\{2, 1\}$ and so on. If no list is provided, the default is $\{1, 1, \dots, 1\}$, corresponding to the use of the first factorization only.

4.6. ToLifting module

This converts the polyphase factorization into the sequence of lifting steps. Recall that the polyphase factorization is a sequence of alternating upper and lower triangular matrices, followed by a diagonal scaling matrix. The diagonal entries of the triangular matrices are all equal to unity. In general, the first matrix can be upper or lower triangular, and the module works in each case. After converting the factorization into a list of factors, it tests to see whether the first matrix is upper triangular. If so, the first element to be changed is s , and the symbol to be changed alternates until we reach the scaling steps (the diagonal matrix). We simulate this by defining the list $components = \{s, d\}$. A generic step similar to (14) is defined with x being the first or last component of $components$. The variable $offset$ is defined to offset arguments of *Part* as applied to $components$, according to whether s or d is the variable to be changed. When all the intermediate steps have been found, the lazy steps are prepended, and the scaling steps are found using the final diagonal matrix. If one wishes to implement the lifting steps in another language, the output of *ToLifting* can serve as pseudocode.

A *Mathematica* replacement rule is defined to simulate the z -operator as defined in (10). This rule is applied to all the steps of the transform, after some algebraic simplification to ensure the replacement rule will function properly.

A slightly different notation from that used in [2] was developed, in order to be more conducive to programming, since superscripts are difficult for computer algebra systems to distinguish from exponents. The notation ‘ s ’ and ‘ d ’ is still used to denote ‘signal’ and ‘difference’ terms respectively. Also in [2] subscripts were used to denote intermediate steps. As we have seen, these are unnecessary, because we can overwrite the old values at every step.

The *InverseLifting* module finds the steps of the inverse wavelet transform given the forward steps. We do not solve any equations, but instead use *pattern matching* and *replacement* [16]. The module first reverses the order of the steps and converts them to equations, dropping the lazy steps. This is done because it is syntactically more difficult to carry out replacements on rules. It works by recognizing typical intermediate steps and typical scaling steps, and replacing them with their inverse. The final operation is to replace steps of the form $s_{l+1} \rightarrow s_l$ with the equivalent step $s_l \rightarrow s_{l-1}$, for the sake of form and programming. Note that the replacement is applied to all the steps, although it will only be needed for the last two (the inverse scaling steps). This is not a serious problem, since even for the longest practical filters the number of lifting steps is relatively small.

4.7. ToMathematica module

Once the lifting steps have been found, it is helpful to be able to test them by carrying out wavelet transforms as required. It is a relatively straightforward matter to convert to them into *Mathematica* form. The first is simply a replacement of the symbols used in the general notation to that used by *Mathematica*. In particular, there is no need to refer explicitly to list positions, since we can carry out operation on all list elements in parallel, and shifts are indicated by the *RotateLeft* and *RotateRight* commands. The user can then copy the output from this command and create a new module from it to carry out one stage of the algorithm. The wavelet transform can be carried out on a list by using *NestList* combined with the resulting module.

When the replacements have been made, we only need to add in the trivial steps that occur in the forward or inverse transform. Specifically, if we are carrying out the forward transform, we add the preliminary lazy steps, and if we are carrying out the inverse transform, we must include a final joining step.

The syntax to convert the inverse to *Mathematica* form is `ToMathematica[expr, Inverse \rightarrow True]`, and where there is no second argument, the default conversion setting is to find the forward transform.

5. Examples

In this section the use of the program is demonstrated with examples.

5.1. D4

Consider the factorization of D4, the basis given in the example of §2.5. We create the list of four nonzero coefficients, needed for polynomial reduction procedures, as:

$$\text{In}[4] := \mathbf{H} := \text{Table}[\mathbf{h}_i, \{\mathbf{i}, \mathbf{0}, \mathbf{3}\}]$$

The filters are

$$\text{In}[5] := \mathbf{h}[\mathbf{z}_-] = \mathbf{h}_0 + \frac{\mathbf{h}_1}{\mathbf{z}} + \frac{\mathbf{h}_2}{\mathbf{z}^2} + \frac{\mathbf{h}_3}{\mathbf{z}^3};$$

$$\text{In}[6] := \mathbf{g}[\mathbf{z}_-] = \text{Collect}[\mathbf{z}^{-1}\mathbf{h}[-\mathbf{z}^{-1}], \mathbf{z}]$$

$$\text{Out}[6] = -h_3z^2 + h_2z - h_1 + \frac{h_0}{z}$$

and so the polyphase matrix is

$$\text{In}[7] := \mathbf{P}[\mathbf{z}_-] = \text{Polyphase}[\mathbf{z}]$$

$$\text{Out}[7] = \begin{pmatrix} h_0 + \frac{h_2}{z} & -h_1 - zh_3 \\ h_1 + \frac{h_3}{z} & h_0 + zh_2 \end{pmatrix}$$

which can be compared with Eqs. (5) and (6). We run the Euclidean algorithm on the low-pass polyphase components – the entries of the first column of $P[z]$. Finding all factorizations gives;

$$\text{In}[8] := (\text{AllFacts} = \text{EuclideanFactorisation}[\mathbf{P}[\mathbf{z}][[\mathbf{1}, \mathbf{1}], \mathbf{P}[\mathbf{z}][[\mathbf{2}, \mathbf{1}]]], \text{EliminationBranch} \rightarrow \text{All},$$

$$\text{PolynomialReduction}[\#, \text{DaubechiesZeros}[\mathbf{H}], \mathbf{H}] \&)]//$$

ColumnForm//Simplify

$$\text{Out}[8] = \left\{ \left\{ \left\{ 1 + \frac{1}{2\sqrt{2}h_3}, \frac{z - 4\sqrt{2}(z+1)h_3 - 1}{4z} \right\}, -\frac{1}{4h_3} \right\}, \left\{ \left\{ \frac{\sqrt{2} - 4h_3}{2\sqrt{2} - 4h_3}, \frac{1}{4}(9z - 1) + \sqrt{2}(1 - 3z)h_3 \right\}, \frac{1}{2\sqrt{2}z - 4zh_3} \right\}, \left\{ \left\{ \frac{-8\sqrt{2}h_3z + 5z + 4}{9z - 12\sqrt{2}zh_3}, \frac{3z(\sqrt{2}(1 - 7z) + 4(5z - 1)h_3)}{16h_3} \right\}, \frac{4h_3}{3z^2(4\sqrt{2}h_3 - 3)} \right\}, \left\{ \left\{ \frac{-4z + 8\sqrt{2}h_3 + 1}{4\sqrt{2}h_3 + 1}, \frac{\sqrt{2}(z+1) + 4(z+3)h_3}{8z^2(\sqrt{2} - 2h_3)} \right\}, \frac{2z(\sqrt{2} - 2h_3)}{4\sqrt{2}h_3 + 1} \right\} \right\}$$

where the first elements are the lists of quotients and the last elements are the gcds.

Note that although these expressions are fairly complicated, only one of the filter coefficients (h_3) appears. The number of factorizations is in agreement with the argument in §3.3, viz. $4 \times 3^{|b|-1} = 4$. Note that although the gcds appear quite different in each case, they are all monomials in z , and so are all the same up to an invertible factor. Each of the four factorizations above would give rise to a different set of lifting steps, but all are implementations of the same transform.

Let us now construct the lifting implementation of the first factorization. We find the polyphase matrix factorization:

**In[9] := PolyphaseFactorisation[P[z], AllFactors[[1]],
PolynomialReduction[#, DaubechiesZeros[H], H]&]**

$$Out[9] = \begin{pmatrix} 1 & 1 + \frac{1}{2\sqrt{2}h_3} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ -\sqrt{2}h_3 + \frac{-\sqrt{2}h_3 - \frac{1}{4}}{z} + \frac{1}{4} & 1 \end{pmatrix} \\ \times \begin{pmatrix} 1 & z\frac{1}{2\sqrt{2}h_3} + \frac{1}{16h_3^2} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -\frac{1}{4h_3} & 0 \\ 0 & -4h_3 \end{pmatrix}$$

To simplify the factorization we substitute the exact value of h_3 [5]:

**In[10] := MatrixFactorisation =
%/h3 -> $\frac{1 - \sqrt{3}}{4\sqrt{2}}$ //Simplify//RootReduce//ToRadicals**

$$Out[10] = \begin{pmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ \frac{\sqrt{3}z + \sqrt{3} - 2}{4z} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & z \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \sqrt{2 + \sqrt{3}} & 0 \\ 0 & \sqrt{2 - \sqrt{3}} \end{pmatrix}$$

and from this we get the lifting implementation:

In[11] := (LiftingSteps = MatrixFactorisation//ToLifting)//ColumnForm

Out[11] = $s_l \leftarrow x_{2l}$

$$d_l \leftarrow x_{2l+1}$$

$$d_l \leftarrow d_l - \sqrt{3}s_l$$

$$s_l \leftarrow \frac{\sqrt{3}d_l}{4} + \left(-\frac{1}{2} + \frac{\sqrt{3}}{4}\right)d_{l+1} + s_l$$

$$d_l \leftarrow d_l + s_{l-1}$$

$$s_l \leftarrow \sqrt{2 + \sqrt{3}}s_l$$

$$d_l \leftarrow \sqrt{2 - \sqrt{3}}d_l$$

5.2. Inverse D4 transform

We find the inverse of the above lifting implementation of the D4 wavelet transform:

In[12] := LiftingSteps//InverseLifting//ColumnForm

$$Out[12] = d_l \leftarrow \frac{d_l}{\sqrt{2 - \sqrt{3}}}$$

$$s_l \leftarrow \frac{s_l}{\sqrt{2 + \sqrt{3}}}$$

$$d_l \leftarrow d_l - s_{l-1}$$

$$s_l \leftarrow -\frac{1}{4}\sqrt{3}d_l - \left(-\frac{1}{2} + \frac{\sqrt{3}}{4}\right)d_{l+1} + s_l$$

$$d_l \leftarrow d_l + \sqrt{3}s_l$$

$$x_{2l+1} \leftarrow d_l$$

$$x_{2l} \leftarrow s_l$$

Observe that the inverse is just as simple as the forward transform.

5.3. D10

Computing the factorization of D10 numerically clearly shows the benefit of using this program. With the D10 coefficients as

In[13] := D10Coefficients

Out[13]={0.160102, 0.603829, 0.724309, 0.138428, -0.242295,
-0.0322449, 0.0775715, -0.00624149, -0.0125808, 0.00333573}

we again construct the polyphase filters and matrix:

In[14] := h[z_] = D10CoefficientsTable[z⁻¹, {i, 0, 9}];

In[15] := g[z_] = Collect[z⁻¹h[-z⁻¹], z];

In[16] := P[z_] = Polyphase[z];

Note that there are a large number of factorizations;

**In[17] := (AllFactors = EuclideanFactorisation[{P[z][[1, 1]], P[z][[2, 1]]},
EliminationBranch → All, Chop])//Length**

Out[17]=108

again in agreement with §3.3 viz. 4.3^{4-1} . We examine two of these, each with constant gcd. The following factorization would be numerically unstable;

In[18] := AllFactors[[2]]

Out[18]={ { -0.265145, -0.499843 - $\frac{0.878163}{z}$,
- 212910. - $\frac{45.7871}{z}$, 4.69682×10^{-6} - $\frac{1.00854 \times 10^{-9}}{z}$,
- 4.16681×10^{10} - $\frac{1.25669 \times 10^{11}}{z}$ }, - 2.4687×10^{-6} }

whereas the following factorization should be stable:

In[19] := AllFactors[[54]]

Out[19]={ { 3.77152, $0.0698843z - 0.247729$, $\frac{3.03369}{z^2} - \frac{7.59758}{z}$,
 $0.0157993z^3 - 0.0503964z^2$, $\frac{0.172573}{z^4} - \frac{1.10315}{z^3}$ }, 0.347389 }

In general, it is preferable to have as many coefficients as possible close to unity. It is clear that in this case there are several factorizations that do not satisfy this criterion; this shows the advantage of non-unique factorization.

5.4. Cohen–Daubechies–Feauveau (4, 2) filter

We now demonstrate that the program works for non-orthogonal filters. The Cohen–Daubechies–Feauveau (4, 2) (CDF (4, 2)) filters are given by

$$\begin{aligned} In[20] := \mathbf{h}[z_-] &= \frac{\sqrt{2}}{32} (40 + 5(z + z^{-1}) - 12(z^2 + z^{-2}) + 3(z^3 + z^{-3})); \\ In[21] := \mathbf{g}[z_-] &= \frac{\sqrt{2}}{16} (z^{-3} - 4z^{-2} + 6z^{-1} - 4 + z). \end{aligned}$$

The polyphase matrix is guaranteed to have monomial determinant, in fact we require it to be unity [2]. This is achieved by dividing the entries of the second column by the determinant. Substitution of the appropriate coefficients will verify that the D4 and D10 polyphase matrices had determinant 1. For the CDF (4, 2) filter, the determinant is -1 , so we reconstruct the polyphase matrix as

$$\begin{aligned} In[22] := \mathbf{P}[z_-] &= \mathbf{Polyphase}[z] \\ Out[22] &= \begin{pmatrix} -\frac{3z}{4\sqrt{2}} + \frac{5}{2\sqrt{2}} - \frac{3}{4\sqrt{2}z} & -\frac{1}{2\sqrt{2}} - \frac{1}{2\sqrt{2}z} \\ \frac{3z^2}{16\sqrt{2}} + \frac{5z}{16\sqrt{2}} + \frac{5}{16\sqrt{2}} + \frac{3}{16\sqrt{2}z} & \frac{z}{8\sqrt{2}} + \frac{3}{4\sqrt{2}} + \frac{1}{8\sqrt{2}z} \end{pmatrix}. \end{aligned}$$

We note that in this case h_o has degree greater than that of h_e . In this case the first quotient is zero, [2]; this is allowed for in the program.

$$\begin{aligned} In[23] := & \mathbf{EuclideanFactorisation}[\{\mathbf{P}[z][[1, 1]], \mathbf{P}[z][[2, 1]]\}, \\ & \mathbf{EliminationBranch} \rightarrow \{\mathbf{EliminateHigh}, \mathbf{EliminateEndsMatchLow}, \\ & \mathbf{EliminateEndsMatchHigh}, \mathbf{EliminateHigh}\}] \end{aligned}$$

$$Out[23] = \left\{ \left\{ 0, -\frac{z}{4} - \frac{1}{4}, -1 - \frac{1}{z}, \frac{3z}{16} + \frac{3}{16} \right\}, 2\sqrt{2} \right\}.$$

This gives rise to the polyphase matrix factorization:

$$\begin{aligned} In[24] := & \mathbf{PolyphaseFactorisation}[\mathbf{P}[z], \%] \\ Out[24] &= \begin{pmatrix} 1 & 0 \\ -\frac{z}{4} - \frac{1}{4} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -1 - \frac{1}{z} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ \frac{3z}{16} + \frac{3}{16} & 1 \end{pmatrix} \cdot \begin{pmatrix} 2\sqrt{2} & 0 \\ 0 & \frac{1}{2\sqrt{2}} \end{pmatrix}. \end{aligned}$$

We find the total number of factorizations:

$$\begin{aligned} In[25] := & \mathbf{EuclideanFactorisation}[\{\mathbf{P}[z][[1, 1]], \mathbf{P}[z][[2, 1]]\} // \mathbf{N}, \\ & \mathbf{EliminationBranch} \rightarrow \mathbf{All}, \mathbf{Chop}] // \mathbf{Length} \end{aligned}$$

$$Out[25] = 9$$

Given that we have effectively reversed the order of the starting Laurent polynomials, this agrees with the analysis of §3.3, viz. $3^2 = 9$.

6. Conclusion

The program finds all factorizations of two Laurent polynomials using the Euclidean algorithm with degree 1 quotients, and can display the matrix factorization of a polyphase matrix. Conversion of this factorization into a

sequence of lifting steps was automated, as was conversion of the lifting steps into *Mathematica* input form. The inverse of a given wavelet transform can be found either in general form or in *Mathematica* form.

The ability to provide exact factorizations for bases such as DN was enabled through the use of polynomial reduction with Gröbner bases. This uses the relations between the filter coefficients, and, as a result, we can obtain an exact factorization for a basis such as $D8$ in terms of only one of the filter coefficients, even though the coefficients themselves cannot be calculated exactly. Future research efforts in this area could investigate the use of quotients whose degree is greater than one. This would preclude in-place implementation, but may lead to factorizations which are favorable in other ways. As noted previously, higher degree quotients would be necessary to factorize two Laurent polynomials whose degrees differed by more than 1. Integer to integer transforms could also be incorporated into this new implementation of lifting [14]. Choosing the optimum factorization for a given purpose was not investigated in detail, and the question of what makes a ‘good’ factorization is very much an open one. For example, it is preferable to have as many coefficients as possible of the order of unity, but there are no well-defined criteria for deciding whether the given coefficients are too big or too small. This issue may best be investigated by those applying this method to a particular task.

References

- [1] M.J. Maslen, Factoring Wavelet Transforms into Lifting Steps (University of Western Australia, 1997). URL: <http://www.physics.uwa.edu.au/~paul/publications.html>.
- [2] I. Daubechies, W. Sweldens, J. Fourier Anal. Appl. 4 (1998) 247. URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#factor>.
- [3] J.N. Bradley, C.M. Brislawn, T. Hopper, Visual Info. Process. II (SPIE, 1961) (1993).
- [4] P.C. Abbott, Introduction to Wavelets (University of Western Australia, 1998). URL: <http://www.physics.uwa.edu.au/Physics/Courses/Honours/Modules/wavelets.html>.
- [5] I. Daubechies, Ten Lectures on Wavelets (SIAM, 1992).
- [6] T.H. Koornwinder, Wavelets: An Elementary Treatment of Theory and Applications (World Scientific, 1993).
- [7] D.E. Newland, Introduction to Random Vibrations, Spectral and Wavelet Analysis (Longman, 1993).
- [8] G. Strang, SIAM Review 31 (1989) 614.
- [9] J.C. van den Berg, Wavelets in Physics (Cambridge University Press, 1998).
- [10] W. Sweldens, Wavelets and the lifting scheme: a 5 minute tour. URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#iciam95>.
- [11] W. Sweldens, The lifting scheme: a custom-design construction of biorthogonal wavelets (1995). URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#lift1>.
- [12] W. Sweldens, The lifting scheme: a new philosophy in biorthogonal wavelet construction. URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#spie95>.
- [13] W. Sweldens, P. Schröder, Building your own wavelets at home. URL: <http://cm.bell-labs.com/who/wim/papers/papers.html#athome>.
- [14] A.R. Calderbank, I. Daubechies, W. Sweldens, B.-L. Yeo, Appl. Comp. Harm. Anal., in press. URL: <http://cm.bell-labs.com/who/wim/papers/index.html#integer>.
- [15] I. Daubechies, Comm. Pure Appl. Math. 41 (1988) 909.
- [16] S. Wolfram, The *Mathematica* Book, 3rd edn. (Wolfram Media/Cambridge University Press, 1996).