Chapter 6

Dictionary Approximation

In this chapter, we revisit the topics of dictionary design and implementation. The results presented in previous chapters were all based on matching pursuit using a particular dictionary of separable 2-D Gabor functions. This dictionary was first introduced in [45] and was completely described in Chapter 2. Additional dictionary designs have been proposed by others in order to improve the coding efficiency [10] or reduce encoder complexity [55][15]. In this work, we introduce a new design methodology which incorporates both coding efficiency and complexity in a systematic way. The key to the method is an algorithm which takes an arbitrary 2-D dictionary and generates approximations of the dictionary which have fast 2-stage implementations. By varying the quality of the approximation, we can explore a systematic tradeoff between the coding efficiency and complexity of the resulting matching pursuit video encoder. As a practical result, we show that complexity reduction factors of up to 1000 are achievable with negligible coding efficiency losses of about 0.1 dB PSNR.

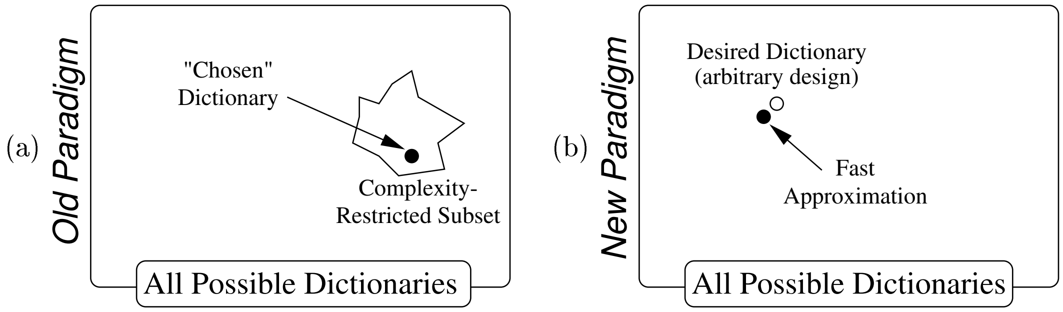
6.1 Introduction

Dictionary design is an important topic, affecting both the coding efficiency and complexity of the resulting matching pursuit encoder [17][14]. For coding efficiency, the dictionary should be designed so that structures in the motion residuals may be built using as few dictionary functions as possible. Complexity is determined in large part by the number of dictionary functions and their spatial extent, since matching pursuit encoding is based on exhaustively computing the inner products between the residual signal and all translates of the dictionary functions in a local area. The complexity of this inner product computation may be drastically reduced by assuming a computationally efficient dictionary structure such as 2-D separable functions, and then designing the dictionary within this restricted set. This design paradigm is illustrated in Figure 6.1a. As we will show in Section 6.2, this is the method used in all previously published matching pursuit video compression systems [45][15][55][10].

There is a problem with this method of dictionary design. As the figure illustrates, any initial restrictions reduce the size of the design space. Even if some "optimal" design procedure is used within this restricted set of possibilities, a better design may exist outside of the set. For example, restriction to separable 2-D functions is typical [45][15][12][10]. However, this excludes structures like diagonal edges and curves. The resulting dictionary is then unable to efficiently code such structures, which are both commonly present and perceptually important [7][13].

By lifting these initial restrictions and allowing arbitrary 2-D functions to exist in the dictionary, coding efficiency may be improved. In fact, well-known optimization

Figure 6.1: Basis design paradigms for matching pursuit. (a) Old paradigm based on initial computational restrictions. (b) New paradigm enabled by dictionary approximation.



techniques such as the Lloyd algorithm [31] might be adapted to optimize the coding efficiency of the dictionary.[1] Unfortunately, if no computational structure can be assumed, then the inner product computation requires full, nonseparable 2-D matching. This can be prohibitively expensive. For example, it was shown in [45] that full 2-D matching was 12.8 times more complex than an efficient separable implementation of the given dictionary.

In this work, we start with an arbitrary 2-D dictionary and develop a framework for generating approximations of that dictionary which have fast implementations. This enables a new design paradigm, as illustrated in Figure 6.1b. An initial dictionary is chosen without complexity restrictions, as shown by the "$\circ$" symbol in the figure. In choosing this dictionary, any design or optimization techniques may be used. The approximation procedure is then employed to generate a second dictionary, as shown by "$\bullet$". The second dictionary approximates the first, and is thus capable of similar coding efficiency. However, the second dictionary allows for a fast implementation. Specifically, it is built to accomodate an efficient 2-stage implementation introduced in [55].

A further advantage is gained by allowing the quality of the approximation to be

[1] This has effectively been done in [10], but there the optimization was restricted to 2-D separable dictionaries.

set by the user. At high quality, the approximated dictionary may be arbitrarily close to the original, and thus may approach the original in coding performance. By reducing the approximation quality, implementation complexity is also reduced. The framework thus allows a tradeoff between approximation quality and encoder complexity. As will be shown in Section 6.4, this leads to a systematic tradeoff between coding efficiency and complexity. For a given dictionary, we may adjust the approximation quality until the desired balance is reached.

The chapter is organized as follows. Section 6.2 reviews previous matching pursuit video coding systems, particularly those relevant to dictionary design. Section 6.3 develops the details of dictionary approximation. The method by which the approximations are generated is developed mainly in Section 6.3.2, and implementation issues for the resulting encoder are discussed in Section 6.3.3. Encoding results are shown in Section 6.4, followed by conclusions in Section 6.5.

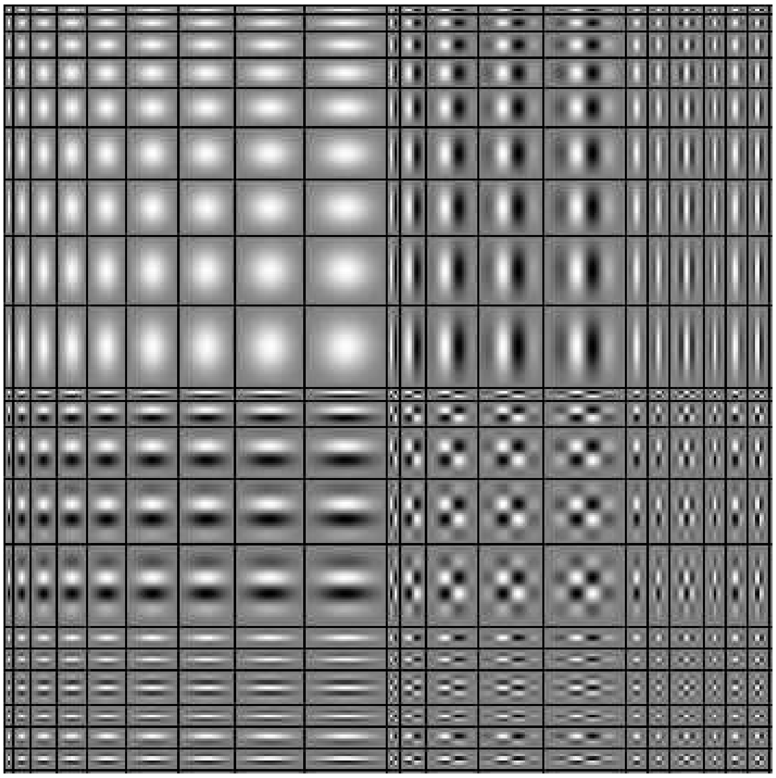
6.2 Review of previous work

We begin with a brief review of our original 2-D Gabor dictionary. Section 6.2.2 then discusses alternate dictionaries proposed by others. The 2-stage dictionary structure of Redmill, et.al. [55] is particularly relevant, so this will be reviewed in Section 6.2.3.

6.2.1 Our original dictionary

Recall that the dictionary used in the previous chapters was based on a 2-D separable Gabor design. The exact definition of this dictionary was given in Section 2.3.2. This

Figure 6.2: The 2-D separable Gabor dictionary.



dictionary is shown again here as Figure 6.2. Each atom is a single 2-D Gabor function selected from the figure and placed at some spatial location (x, y) in the coded residual image. Note that the "Old Paradigm" from Figure 6.1a was used to design this set. Specifically, the design was restricted to separable 2-D Gabor functions, and an ad-hoc training method was used to select 20 1-D filter functions to be applied both horizontally and vertically to define the dictionary. Note also that because the dictionary is separable, it contains no diagonally oriented shapes.

6.2.2 Other dictionaries

Several alternate dictionaries have been proposed by others. A recent work by Chou, et.al. [10] used a standard technique from Gain-Shape Vector Quantization to optimize a dictionary for coding efficiency. The dictionary was first restricted to separable 2-D functions, and so the "Old Paradigm" of Figure 6.1a applies. Within this restricted set, an iterative training algorithm was used to design a new dictionary, and PSNR improvement was shown in certain cases.

Several recent works have proposed dictionaries based on "factorized" separable filters. The dictionaries are designed so that large filter functions are realized by successive application of short kernal filters, thus reducing the complexity. A wavelet packet basis using Haar filters was proposed by de Vleeschouwer and Macq [15]. An alternate factorized dictionary which approximates a separable Gabor set was shown by Czerepiński, et.al. [12]. Because separability is assumed, these cases both follow the old paradigm of Figure 6.1a. Further, these methods are unable to implement arbitrary 2-D functions, and so are unsuitable for our current purpose.

A more flexible dictionary structure was proposed by Redmill, et.al. [55]. In the context of Figure 6.1a, Redmill's dictionary functions were restricted to simple oriented edge patterns, each constructed from weighted combinations of a few functions from a smaller elementary dictionary. This careful construction was then shown to enable an efficient implementation based on a 2-stage filtering structure. Because our method is based on a generalization of this 2-stage structure, we now review the original work in detail.

6.2.3 Fast 2-Stage Implementation

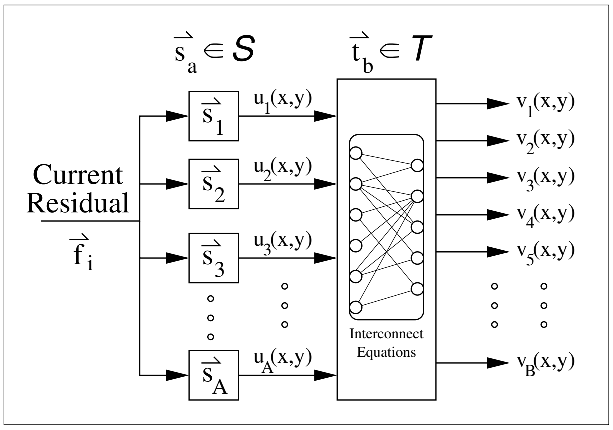
In Redmill, et.al. [55], each function in a target dictionary T is composed of a weighted summation of functions from a simpler elementary dictionary S. It then becomes possible to compute the target inner products as weighted summations of the elementary inner products. For example, suppose $\vec{t} \in T$ and $\vec{s}_1, \vec{s}_2 \in S$ are defined such that $\vec{t} = c_1\vec{s}_1 + c_2\vec{s}_2$. Then

$$\left\langle \vec{f}_i, \vec{t} \right\rangle = c_1 \left\langle \vec{f}_i, \vec{s}_1 \right\rangle + c_2 \left\langle \vec{f}_i, \vec{s}_2 \right\rangle \tag{6.1}$$

More generally, the inner products needed for matching pursuit decomposition may be computed using the 2-stage filtering structure shown in Figure 6.3. For a particular atom stage i, the inner products between $\vec{f}_i$ and the required (x, y) positional translates of the elementary functions $\vec{s}_a \in S$ are first computed. This produces A elementary inner product buffers $u_a(x, y)$. Weighted combinations of these elementary inner products are then used to form B target inner product buffers $v_b(x, y)$. These represent the inner products between $\vec{f}_i$ and the positional translates of each target dictionary function $\vec{t}_b$. The largest inner product among all the $v_b(x, y)$ may thus be used to determine the atom to code for stage i. Note that the interconnect box which forms the target inner products is defined by B equations, each a generalization of Equation (6.1).

The above structure has several advantages which may be exploited for computational efficiency. First, elementary dictionary S may be designed around a computationally efficient structure, even if target dictionary T does not share that structure. In Redmill's case, S was defined to consist of eight 2-D separable raised cosine filters at various scales (1×1, 3×3, 5×5, 7×7, 11×11, 15×15, 23×23, and 31×31). The target dictionary T was not

Figure 6.3: A two-stage filtering structure used by Redmill, et.al., for efficient implementation of a nonseparable dictionary.



2-D separable, since it contained oriented functions constructed from shifted copies of the elementary functions. The elementary inner products are thus computed separably, with the more complex target inner products computed efficiently from the elementary buffers.

A second advantage comes from the introduction of a progressive update step for the elementary inner product buffers. Rather than recomputing $u_a(x, y)$ for each atom stage i, the elementary filtering is done only once per frame. Each atom stage then consists of these steps:

1. *Find Atom:* Apply the interconnect equations to compute target inner products $v_b(x, y)$ over a local search area. The largest target inner product selects the i^{th} atom.
2. *Atom Update:* Subtract the selected atom from $\vec{f_i}$
3. *Elementary Update:* Recompute only those elementary inner product values in $u_a(x, y)$ which were affected by the atom update step.

For the third step, the range of affected inner products within each elementary buffer $u_a(x, y)$ will depend on the extent of the chosen atom as well as that of the associated elementary dictionary function $\vec{s}_a$. An efficient update procedure shown in [55] allows this step to be done at a cost of only one multiply-accumulate per affected elementary inner product. Additional details on this method will be given in Section 6.3.3.

A third complexity advantage arises from a special dependency structure that may be built into the dictionary. Suppose target dictionary functions $\vec{t_b}$ are ordered by index b, and the target inner product buffers are computed in the same order. When computing the inner products to fill a particular buffer $v_b(x,y)$, the previously computed inner products in buffers $v_{b'}(x,y), b' < b$ are available. Redmill's dictionary was carefully designed to construct each target $\vec{t_b}$ from a previous target $\vec{t_{b'}}$ with the addition of one or two functions from S. Each inner product in $v_b(x,y)$ is thus formed by combining a previously computed target inner product with one or two previously computed elementary inner products. By recycling inner product computation in this way, complexity is significantly reduced.

We now analyze the complexity of this fast 2-stage implementation. Such analysis is needed for two reasons. First, it gives insight into the design and later complexity analysis for dictionary approximation, which is built on the same 2-stage structure. Second, it expands on the analysis provided in Redmill et.al. [55]. The analysis in that work covered only the computation of target inner product buffers from the already computed elementary buffers. There are two additional sources of complexity which must be considered, as we now demonstrate.

The inner product computation complexity may be divided into three sources, as illustrated in Table 6.1. The initial elementary filtering complexity is called $\Gamma(IF)$. This covers initial computation of elementary buffers $u_a(x,y)$, which is done once per coded frame. This complexity depends on the elementary dictionary S, taking into account any available structure for fast computation. Because the initial filtering is done over all (x, y) positions in the motion residual image, $\Gamma(IF)$ depends on the image size as well. The units of $\Gamma(IF)$ are taken to be operations per frame, where total operation count is the sum of required additions and multiplications.

The other two sources of complexity are $\Gamma(FA)$ and $\Gamma(EU)$, which represent the Find Atom and Elementary Update steps. $\Gamma(FA)$ is the operation count due to applying the interconnect equations in order to compute the target inner products from the already computed elementary buffers. The atom search is done in a local area near an energy peak [45], and so the target inner products for each atom stage need only be computed within a local area of fixed size. $\Gamma(FA)$ thus depends on the size of this local search region, as well as on the interconnect equations.

$\Gamma(EU)$ is the operation count used to update the elementary buffers after the selected atom is subtracted from the remaining residual signal. Because only the values affected by the atom update are recomputed, the complexity depends on the extent of the selected atom. This makes $\Gamma(EU)$ signal dependent, unlike $\Gamma(IF)$ and $\Gamma(FA)$. Indeed, if the interconnect equations, the image size, and the size of the local atom search region are known, $\Gamma(IF)$ and $\Gamma(FA)$ may be exactly computed independently of the motion residual signals. Signal independent estimates of $\Gamma(EU)$ may also be computed, if some assumptions are made. This will be done for Redmill's dictionary below, and for our approximated dictionaries in Section 6.3.3.

We analyze the implementation complexity of Redmill, et.al. [55], by considering

Table 6.1: Sources of complexity for the fast 2-stage implementation.

Label	Description	Depends on	Counted
$\Gamma(IF)$	Initial Filtering $\rightarrow$ Compute $u_a(x,y)$	Image Size, Elementary set S	Per Frame
$\Gamma(FA)$	Find Atom $\rightarrow$ Compute $v_b(x,y)$ from $u_a(x,y)$	Interconnect equations, Local search extent	Per Atom
$\Gamma(EU)$	Elementary Update $\rightarrow$ Update $u_a(x,y)$ as needed	Atom Update Extent, Elementary set S.	Per Atom

each of the components shown in Table 6.1. Initial filtering complexity is computed as

$$\Gamma(IF) = 4P \sum_{a=1}^{A} L(a) \tag{6.2}$$

operations per frame. Here P is the total number of image pixels, and $L(a)$ is the length of the 1-D filter used to define each of the A separable 2-D elementary filters $\vec{s}_a$. This equation assumes that an $L \times L$ separable filter costs $2PL$ multiplies and an equal number of adds, for a total of $4PL$ operations [38]. The find atom complexity was estimated in [55] to be $\Gamma(FA) = 128 + 256\Omega$ operations per atom, where Ω is the number of pixels in the local atom search area. As stated before, the elementary update complexity is signal dependent, but we estimate it by assuming that an average atom update affects a 16×16 image region. Within elementary buffer $u_a(x,y)$, the atom update then affects the inner products in a square region $16 + (L(a) - 1)$ pixels on a side. To update all affected buffers using the efficient update method of [55] requires

$$\Gamma(EU) = 2 \sum_{a=1}^{A} (15 + L(a))^2 \tag{6.3}$$

operations per atom. If a particular frame is coded to N atoms, then the number of operations used to code that frame is:

$$\Gamma(IF) + N \cdot (\Gamma(FA) + \Gamma(EU)) \tag{6.4}$$

As an example, suppose we are coding a QCIF sequence at 24 kbit/s and 10 frame/s. The number of image pixels is $P = 38016$, and $N = 100$ atoms per frame is typical for our experiments. Redmill's dictionary contains $A = 8$ filters with lengths $L(a) \in \{1, 3, 5, 7, 11, 15, 23, 31\}$. Using a 16×16 local atom search as in [45], we get $\Omega = 256$. Using these values, Equation (6.4) gives a total of about 22.5 million operations per frame. A comparable frame encoding using the separable 2-D Gabor dictionary pictured in Figure 6.2 would cost 340 million operations, using the analysis provided in [45]. The Redmill dictionary is thus 15.1 times faster for this encoding scenario.

Because our method will be designed to take advantage of the same 2-stage structure detailed above, a similar complexity analysis will be used to evaluate the complexity of our approximated dictionaries in Sections 6.3.3 and 6.4. The above exercise also identified three distinct sources of complexity and outlined the factors on which each depends. This will provide insight into the development of the approximation system, as described in Section 6.3.2.

6.3 Dictionary Approximation

6.3.1 Overview of method

Consider an arbitrary initial dictionary $T_0 = \{\vec{t}_{(0,b)}, b = 1 \ldots B\}$ which has been designed without encoder complexity constraints. For example, T_0 might be optimized for coding efficiency using any available optimization technique. We wish to generate approximations of T_0 which have similar coding efficiency, but which have reduced complexity encoder implementations. In this section, we show a method for generating dictionaries

$T_q = \{\vec{t}_{(q,b)}, b = 1 \ldots B\}$, each of which approximates T_0 to some quality index q. Each index q has an associated target distortion $D(q)$ which has been set by the user in order to control the quality of the approximation. Further, T_q is designed to allow an efficient 2-stage implementation using the structure proposed in [55].

To generate T_q, we apply the matching pursuit algorithm in a novel way. Suppose each $\vec{t}_{(0,b)}$ is to be approximated as a weighted combination of simpler elements called "construction atoms." For a given b, each construction atom ψ has an associated weight $c_{(\psi,b)}$ and a unit norm basis shape $\vec{r}_{(\psi,b)}$, which is selected from an appropriately defined construction dictionary R_b. Matching pursuit itself is then used to decompose $\vec{t}_{(0,b)}$ on R_b. This produces a set of construction atoms Ψ_b which approximate the original dictionary function as:

$$\vec{t}_{(0,b)} \approx \vec{t}_{(q,b)} = \sum_{\psi \in \Psi_b} c_{(\psi,b)} \vec{r}_{(\psi,b)} \tag{6.5}$$

Because each $\vec{t}_{(0,b)}$ is expressed as a summation of simpler elements, the complexity advantages discussed in Section 6.2.3 may be exploited. Specifically, the collection $\{\Psi_b, b = 1 \ldots B\}$ is used to define the interconnect equations for a fast 2-stage matching pursuit encoder based on Figure 6.3.

A special dependency structure was built into the dictionary of Redmill, et.al. [55] in order to further reduce complexity. We build a similar structure into our approximated dictionary by constructing R_b in a special way. Consider the original dictionary functions $\vec{t}_{(0,b)}$ to be ordered by index b, with the approximations generated sequentially. For the first target with $b = 1$, we define R_b to be some pre-designed elementary dictionary S. For the later targets, R_b is defined as:

$$R_b = S \cup \{\vec{t}_{(q,b')}, b' < b\} \tag{6.6}$$

Later targets are thus constructed from the approximations of earlier targets. This allows inner product computation to be efficiently recycled in the 2-stage encoder implementation. The use of dependency leads to significant complexity reduction, as will be demonstrated in Section 6.3.2.

The generation algorithm decomposes each $\vec{t}_{(0,b)}$ on the appropriate dictionary R_b. Construction atoms are added to the approximation until the following distortion criterion is met:

$$\left\| \vec{t}_{(q,b)} - \vec{t}_{(0,b)} \right\|^2 \leq D(q) \tag{6.7}$$

In this way, $D(q)$ controls the approximation quality. A smaller value of $D(q)$ results in a higher quality approximation. However, this requires additional construction atoms, each of which adds complexity to the interconnect equations shown in Figure 6.3, and hence to the final 2-stage implementation. There is thus a tradeoff between the quality of the approximation and the complexity of the resulting video encoder.

A summary of the generation algorithm is given in Figure 6.4. Suppose that elementary dictionary S, together with all (x, y) positional translates, is at least complete. In this case, a solution of the form given in Equation (6.5) which satisfies distortion criterion (6.7) is guaranteed to exist for any $D(q)$, and matching pursuit will always converge to a valid solution [41].

The proposed system may be divided into two major stages as shown in Figure 6.5. Figure 6.5a shows the *generation* stage. In this stage, the generation algorithm is applied to

Figure 6.4: A summary of the generation algorithm.

Initialization:

Assume elementary set S, target set T_0, and target distortion $D(q)$ are given. Approximated set T_q is initially empty.

Approximation:

For $b = 1$ to B;

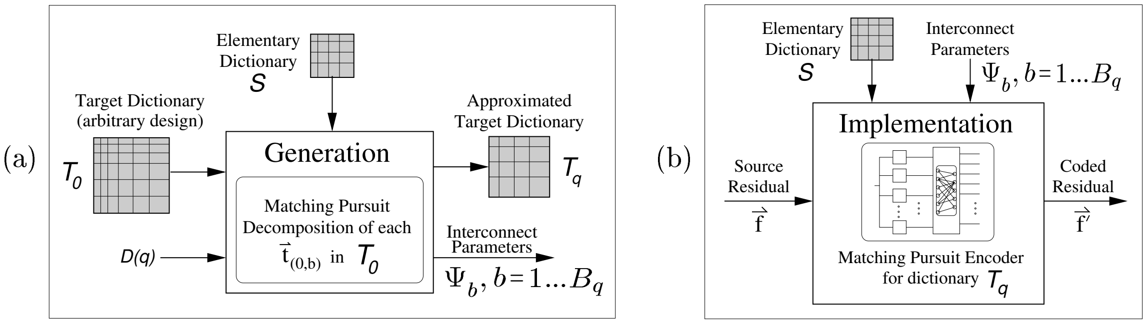
- Decompose $\vec{t}_{(0,b)}$ using matching pursuit with dictionary R_b as defined in Equation (6.6). Take each coded atom to be a construction atom ψ in an approximation of the form given in Equation (6.5). Stop when the constructed approximation $\vec{t}_{(q,b)}$ satisfies distortion criterion (6.7).
- Update $T_q = T_q \cup \{\vec{t}_{(q,b)}\}$

Next b;

dictionary T_0, and the interconnect parameters Ψ_b are generated for each target. The fast 2-stage encoder is then realized in an *implemenation* stage as shown in Figure 6.5b. This stage performs matching pursuit residual encoding using dictionary T_q, which approximates T_0 to distortion level $D(q)$. Because T_q is constructed according to Equation (6.5), the various advantages discussed in Section 6.2.3 are exploited to reduce the complexity. The encoding block described by Figure 6.5b is then used to encode motion residual signals in an efficient matching pursuit video encoder.

Figures 6.5a and 6.5b provide a summary of the dictionary approximation method, and identify the two major stages. Details of the generation stage will be given in Section 6.3.2, and implementation details will be discussed in Section 6.3.3.

Figure 6.5: The two stages of dictionary approximation. (a) Generation of the approximated dictionary, T_q. (b) Implementation of a fast 2-stage matching pursuit encoder which uses dictionary T_q.



6.3.2 Generation

In this section, we develop the generation stage of Figure 6.5a. We complete the earlier description by providing details on the design of elementary set S, and on the handling of dependency. We also introduce improvements which further reduce the complexity of the generated dictionaries. In the final subsection, sample generation results are presented and discussed.

Elementary Set Design

We first discuss the design of elementary set S. This has a direct impact on complexity, since both initial filtering complexity $\Gamma(IF)$ and elementary update complexity $\Gamma(EU)$ depend on S. In particular, $\Gamma(IF)$ may be reduced by basing S on a computationally efficient structure. The design of S also affects interconnect complexity $\Gamma(FA)$, since each approximated target must be constructed using elements of S. It is desirable to design S such that each $\vec{t}_{(0,b)} \in T_0$ may be sufficiently approximated using few construction atoms, since this in turn lowers $\Gamma(FA)$.

The problem of designing S to efficiently "encode" the functions in original dictionary T_0 is analogous to the problem of designing T_0 to encode motion residual signals $\vec{f}$. However, the inclusion of dependency makes any optimal design of S much more difficult. The reason is that the dictionary used for matching pursuit during the generation stage is not S, but construction dictionary R_b. While this dictionary depends on S, it also depends on T_0 and effectively grows throughout the generation process. For this reason, there is no straightforward method for designing S to minimize the complexity for a particular approximation T_q.

For our current experiments, we propose an ad-hoc elementary dictionary design based on a cascade filterbank structure. Each elementary buffer $u_a(x, y)$ is computed from a previously defined elementary buffer by application of a single 1-D filter in either the horizontal or vertical direction. These simple 1-D filters are defined in Figure 6.6a, and the application of these filters is specified by the filterbank diagram in Figure 6.6b. The filterbank effectively implements the 16 element 2-D dictionary pictured in Figure 6.6c. The elements of this dictionary range in size from 1×1 to 9×9 and approximate 2-D Gaussian functions. Note that each of the 1-D filters is designed to have only a small number of nonzero taps, nearly all of which are unity in order to reduce the number of required multiplications. The resulting implementation requires fewer than three operations per computed elementary inner product, as will be shown in the generation results at the end of this section.

Our ad-hoc design produces a reasonable elementary dictionary S with a low initial filtering complexity $\Gamma(IF)$. However, no optimization has been done in order to minimize the number of construction atoms needed to approximate a particular dictionary T_0. Further reduction in complexity $\Gamma(FA)$ may be achievable along these lines. Also note that the dictionary functions defined in Figure 6.6 are not unit norm, since the 1-D filters used to generate them have mostly unity taps. When used at the generation stage, these functions must be renormalized for compatibility with the matching pursuit algorithm. Additional adjustments are made at the implementation stage and will be described in Section 6.3.3.

As a final note, the design of S has a large impact on the memory requirements at the final 2-stage encoder. This is because each elementary inner product buffer $u_a(x, y)$ is essentially a frame buffer. Our encoder requires 16 such buffers. While memory requirements were not considered in our design, one could restrict the size of S if encoder memory is a concern.

Dependency and Ordering

Dependency allows approximated targets $\vec{t}_{(q,b)}$ to be built from previously approximated targets in some predetermined order. Inner products in $v_b(x, y)$ are then built from inner products in previously computed target buffers, and complexity is reduced. As described earlier, we build dependency into approximated dictionary T_q through construction dictionary R_b, which grows throughout the generation process to include all previously approximated targets. However, the complexity reduction achieved due to dependency is affected by the order in which the targets are handled at the generation stage. Essentially, the targets should be ordered so that later targets may be efficiently constructed from earlier, already computed targets.

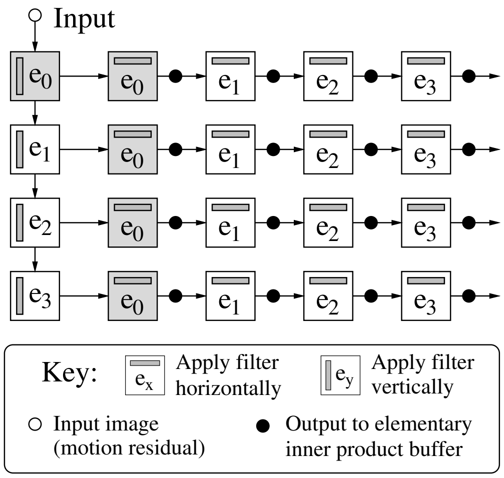
Optimization of the order is a difficult combinatorial problem. We instead propose a heuristic ordering algorithm which improves the dependency related complexity gains. The

Figure 6.6: Implementation of elementary dictionary S. (a) Definition of the simple 1-D filters. (b) Cascade filterbank in which the 1-D filters are applied in order to compute the elementary inner products. Trivial filtering operations are shaded. (c) Effective 2-D elementary dictionary.

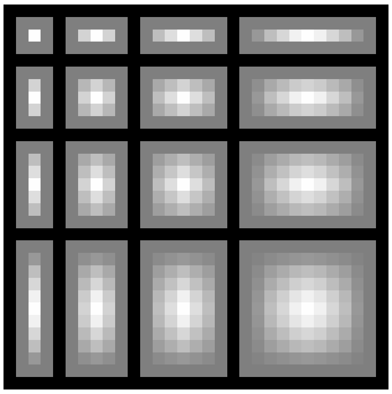
(a) 1-D Filters

e_0			1		
e_1		1	1.5	1	
e_2		1	0	1	
e_3	1	1	0	1	1

(b)



(c) Effective 2-D dictionary



idea is that larger targets should be constructed from smaller targets, and so the ordering is done by size. All pixels in each unit-norm 2-D target function $\vec{t}_{(0,b)}$ are compared to a constant threshhold, and absolute values exceeding the threshhold are considered significant. The targets are then re-ordered by increasing number of significant pixels. A threshhold of 0.25 is used for our experiments. With the targets ordered by size, later targets with larger extents are efficiently built from earlier, smaller targets. Fewer construction atoms are needed to construct each target, and complexity is reduced. This will be demonstrated in the generation results at the end of this section.

Natural Merging

Suppose that two or more original dictionary functions have identical representations in the approximated set. Although this seems unlikely, it is actually common when dependency is enabled. To see why, consider two similar functions in T_0. Call these $\vec{t}_{(0,b_1)}$ and $\vec{t}_{(0,b_2)}$, with $b_1 < b_2$. Now consider generating the approximation of the second function. Because the two functions are similar, it is likely that the first construction atom used to build $\vec{t}_{(q,b_2)}$ will be a dependent copy of the already approximated first function, $\vec{t}_{(q,b_1)}$. If this dependent construction atom satisfies

$$\left|\vec{t}_{(q,b_1)} - \vec{t}_{(0,b_2)}\right| < D(q) \tag{6.8}$$

then distortion criterion (6.7) is satisfied, and the approximation is complete. As a result, $\vec{t}_{(q,b_2)}$ is built as an exact copy of $\vec{t}_{(q,b_1)}$.

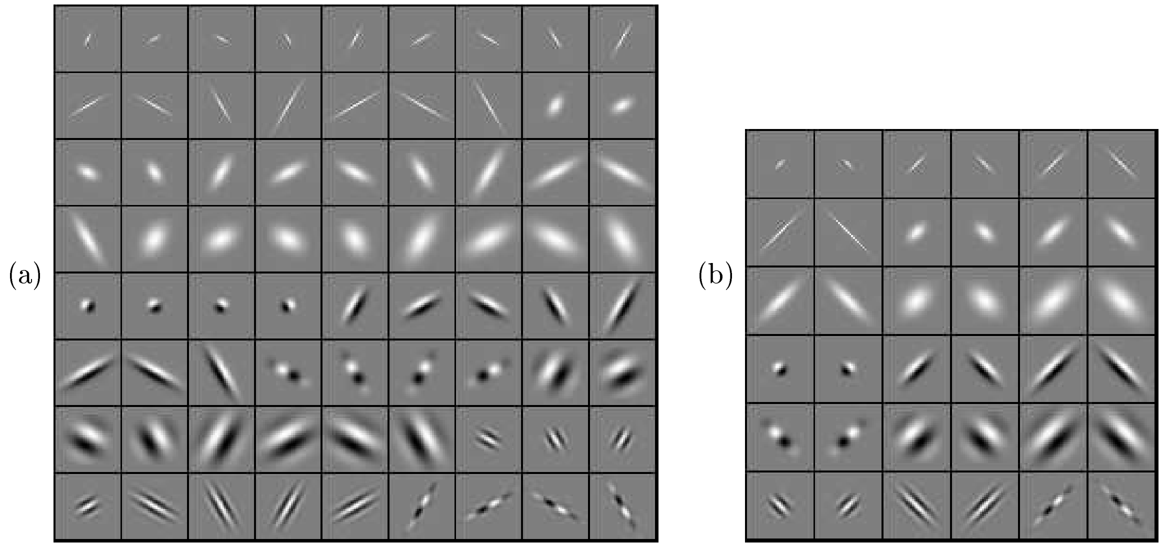
Since there is no advantage to having multiple copies of the same function in the approximated dictionary, such copies are detected and merged before the implementation stage. The approximated dictionary may thus contain fewer elements than the original. Equation (6.8) suggests that the likelihood of merging increases with large $D(q)$. Thus as $D(q)$ increases, more merging occurs and fewer functions remain in the 2-stage implementation of T_q. Merging in this way contributes to encoder complexity reduction.

Orthogonal Matching Pursuit

Orthogonal matching pursuit (OMP), as introduced in [50], may be applied at the generation stage to reduce the complexity at the 2-stage encoder implementation of T_q. To see why, consider the approximation error $(\vec{t}_{(q,b)} - \vec{t}_{(0,b)})$ as each construction atom is added to the approximation given in Equation (6.5). Because matching pursuit is greedy, this approximation error is guaranteed to be orthogonal only to the most recently added construction atom. With OMP, weights $c_{(\psi,b)}$ are recomputed each time a construction atom is added so that the approximation error is orthogonal to all construction atoms in the representation. The error reduction due to each construction atom is increased, and distortion criterion (6.7) may be satisfied with fewer construction atoms. Interconnect complexity $\Gamma(FA)$ at the final 2-stage encoder is in this way reduced for a given $D(q)$.

OMP is most effective when several nonorthogonal construction atoms overlap in the approximation. As we show in the next subsection, gains are only visible at small values of $D(q)$. Although such gains are small, they come at essentially no cost. The computation needed for OMP is low and occurs at the generation stage, where complexity is not a primary concern. Complexity is much more important at the implementation stage, where the OMP related gains are realized.

Figure 6.7: Oriented Gabor functions added to expand the original 2-D Gabor library shown in Figure 6.2. (a) Functions added at multiples of 30 degrees to form the "h30" set. (b) Functions added at multiples of 45 degrees to form the "h45" set.



Generation Results

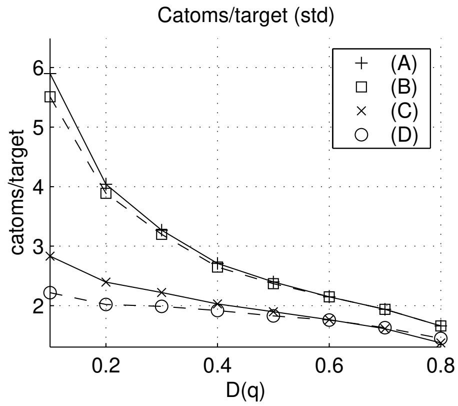
To illustrate the concepts discussed above, we applied the generation algorithm to three matching pursuit dictionaries. The first is the 2-D separable Gabor dictionary introduced in [45] and shown in Figure 6.2. We call this the standard dictionary, and label it as "std." Two additional dictionaries are formed by adding oriented Gabor functions to "std." The added functions are shown in Figure 6.7, and the resulting hybrid dictionaries are labeled "h30" and "h45" since they contain functions oriented at 30 and 45 degrees, respectively.

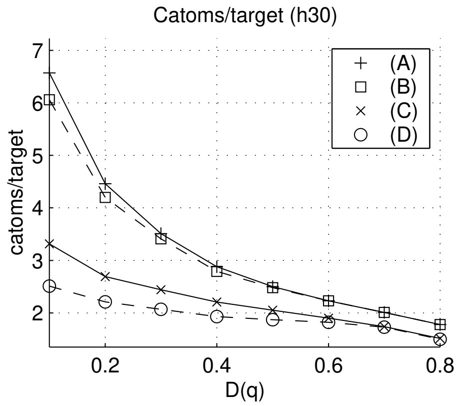
We use the generation algorithm to create approximate dictionaries T_q for each original dictionary over a wide range of $D(q)$. We repeat this process four times to illustrate the different generation options discussed in the previous subsections. Results are plotted in Figure 6.8 in terms of the average number of construction atoms per target. The number of construction atoms effectively determines the interconnect complexity $\Gamma(FA)$, and so fewer construction atoms in the plots leads to a lower implementation complexity.

The four variants of generation shown in Figure 6.8 may be described as follows. Case (A) shows the minimal set of options, using neither OMP nor dependency. Turning dependency off means $R_b = S$ throughout the generation process, and so each approximated target must be built entirely from elementary functions. For case (B), OMP is turned on. As a result, the number of construction atoms is reduced compared to case (A) at small values of $D(q)$. In case (C), dependency is turned on, and the targets are approximated in a random order. To reduce the likelihood of outliers, the experiment is repeated using five different random target orderings and the results are averaged to produce each plot point. We see that even when the approximation order is random, dependency yields significant improvement, reducing the number of construction atoms by nearly a factor of two for some values of $D(q)$. Finally in case (D), dependency is combined with our heuristic ordering algorithm. This further reduces the number of construction atoms by up to 25 percent. The curves for case (D) produce the fewest construction atoms per target, which in turn leads to the lowest interconnect complexity at the implementation stage. The case (D) options are thus used for the remainder of our experiments.

After generation, duplicate functions in T_q are eliminated using the natural merging step described earlier. This reduces the number of dictionary functions at the implementation stage from B to some number $B_q \leq B$. This effect is illustrated in Figure 6.9, which plots B_q against $D(q)$ for approximations of all three dictionaries. Note that for large values of $D(q)$ the number of remaining dictionary functions falls below 10 percent of the original number. Merging thus contributes greatly to complexity reduction when $D(q)$ is

Figure 6.8: Number of construction atoms (catoms) per approximated target as a function of $D(q)$. Each subplot shows a different original dictionary. Generation is done using various combinations of orthogonal matching pursuit (OMP), dependency, and target ordering.





"Options KEY"

	OMP	Dependency	Order
(A)	OFF	OFF	–
(B)	ON	OFF	–
(C)	ON	ON	Random
(D)	ON	ON	Heuristic

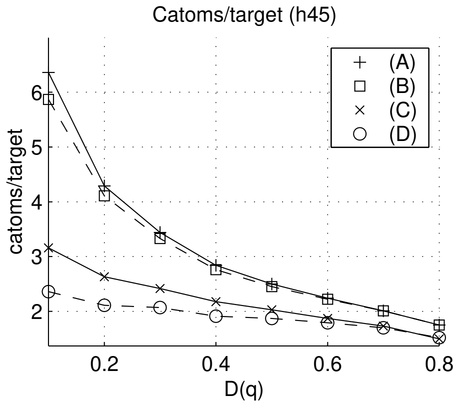
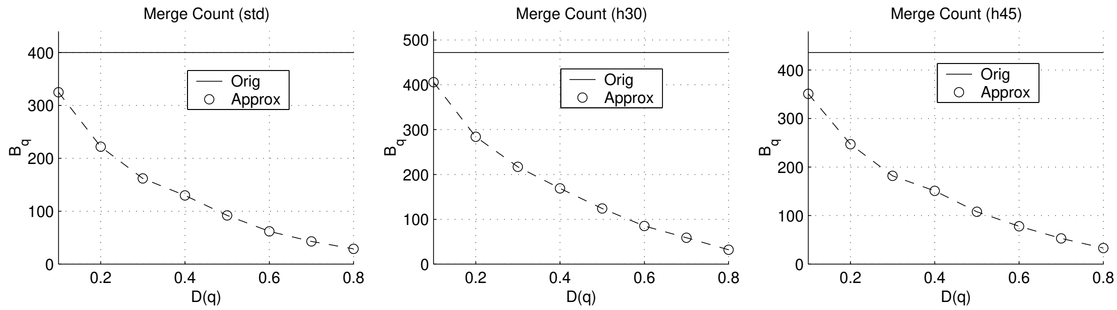


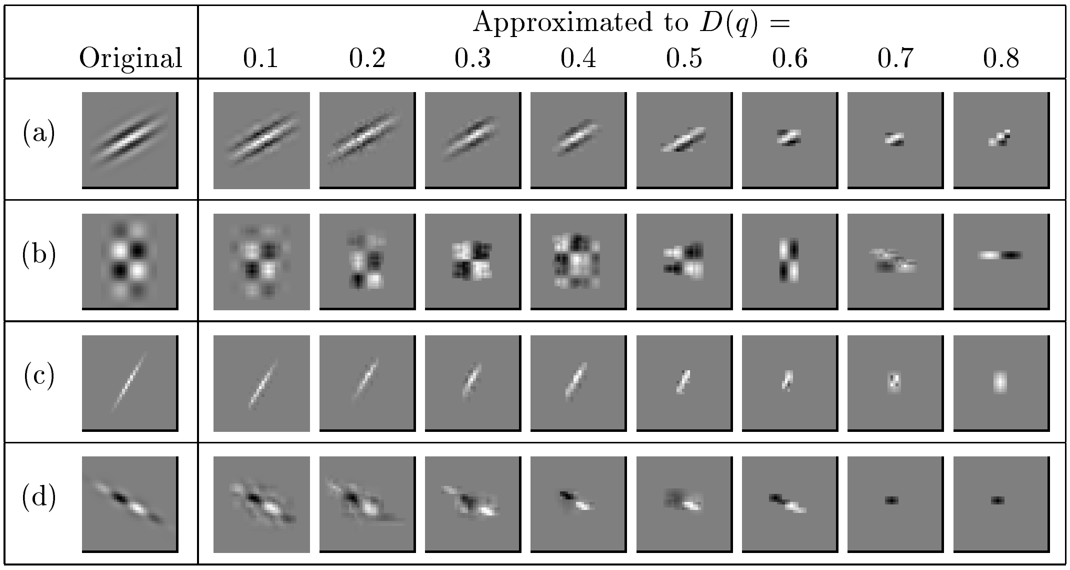
Figure 6.9: Number of approximated dictionary functions B_q remaining after the merging step, as a function of target distortion $D(q)$.



large.

Figure 6.10 shows four sample functions from the "h30" dictionary approximated to various $D(q)$. By examining these functions, much is revealed about the approximation process. The first column shows the original target functions, and the remaining columns show the approximated functions. As seen in the figure, small values of $D(q)$ result in accurate approximations of the original targets. As $D(q)$ increases, the approximated functions look less and less like the originals. An interesting side effect is that the functions shrink as $D(q)$ increases. This is explained by noting that the original targets are 2-D Gabor functions, each constructed from a Gaussian window. The energy of each target thus peaks in the center and trails off at the outer edges with the tail of the Gaussian window. The greedy matching method used for generation tends to code the high-energy central region of each Gabor function first, and the lower energy outer regions later. That is, $\vec{t}_{(q,b)}$ is built from the center outward. As $D(q)$ increases, fewer construction atoms are used, and the extent of the approximated function gets progressively smaller. As $D(q)$ approaches 1, it is often possible to satisfy distortion criterion (6.7) using only a single construction atom. At this point, the approximated functions bear little resemblance to the originals.

Figure 6.10: Some sample approximated functions from the "h30" dictionary.



The power of dependency may be illustrated by comparing the approximated functions in Figure 6.10 with the plots in Figure 6.8. Although intricate approximations are shown for $D(q) = 0.1$, the plots reveal that such functions are built with an average of only 2 to 3 construction atoms per target. This is possible only because functions are built progressively from previously approximated targets. For example, the diagonal line in Figure 6.10c is constructed from shorter lines at the same orientation. Such short diagonals may be found in the original dictionary shown in Figure 6.7a. Heuristic ordering ensures that such small building blocks are approximated early in the generation process. They are thus available to contribute to larger targets which are approximated later.

6.3.3 Implementation

We now describe the implementation stage, as shown earlier in Figure 6.5b. This stage represents the end matching pursuit residual encoder. The basic implementation steps

Figure 6.11: Procedure used at implementation to encode each motion residual. Steps which result in significant complexity are labeled in the right column.

Program Flow	Complexity
Initial elementary filtering (compute initial values for all $u_a(x, y)$)	$\Gamma(IF)$

Do {
    Apply interconnect equations to select next atom
    Subtract chosen atom from residual
    Update elementary buffers
} until encoding bit budget is exhausted.
	
	$\Gamma(FA)$
	
	$\Gamma(EU)$

are similar to that of Redmill, et.al. [55], and are summarized in Figure 6.11. The elementary filters are first applied to motion residual $\vec{f}$, and the resulting elementary inner products are placed in buffers $u_a(x, y)$. The algorithm then enters a loop which performs matching pursuit decomposition of $\vec{f}$ on the approximated dictionary T_q. Three steps are required to encode each atom. First, the interconnect equations are applied to the elementary buffers in order to compute target inner products $v_b(x, y)$ over a local $S \times S$ search area. The largest such inner product selects the next atom, which is then subtracted from the motion residual. Finally, the elementary inner products affected by atom subtraction are updated. Details of these steps differ somewhat from the original implementation in [55]. We now provide these details, and also discuss the complexity of each step.

Initial Filtering

In the initial filtering step, the elementary dictionary defined in Figure 6.6 is applied to the original motion residual $\vec{f}$. Inner products between $\vec{f}$ and all spatial translates of the elementary dictionary functions are computed. These values are placed in elementary buffers $u_a(x, y)$, where a ranges from 1 to A and (x, y) covers the entire image to be coded.

Table 6.2: Initial filtering complexity $\Gamma(IF)$ for typical frame sizes. The right column shows average complexity per computed elementary inner product.

Frame Size	Subimage Extent Luma	Chroma($\times$2)	$\Gamma(IF)$ ops/frame	ops per inner product
QCIF	176×144	88×72	1.40 million	2.31
CIF	352×288	176×144	5.47 million	2.25
SIF	352×240	176×120	4.58 million	2.26

The procedure must be applied to both luma and chroma subimages. Note that $A = 16$ for our elementary design.

Computation of these elementary inner products is done using the 1-D filters of Figure 6.6a applied according to the cascade filterbank of Figure 6.6b. The 1-D filters are defined with mostly unity taps. This greatly reduces the number of required multiplications, but prevents the effective 2-D filters from having unit norm. Matching pursuit decomposition requires a dictionary of unit norm elements, and so a renormalized version of S was used in the generation algorithm described in Figure 6.4. It is thus necessary to renormalize values in elementary buffers $u_a(x, y)$ when the interconnect equations are applied. We do this efficiently by building the renormalization factors into the interconnect equation weights. Details are given in the next subsection.

If the frame size is known, the initial filtering complexity may be exactly computed from the definitions in Figure 6.6. We summarize this complexity for typical frame sizes in Table 6.2. The final column measures the number of operations per computed elementary inner product averaged across all 16 elementary inner product buffers.

Finding Atoms

In this step, the interconnect equations are applied to compute target inner products $v_b(x, y)$. Each target inner product is a weighted combination of values from elementary inner product buffers and previously computed target inner product buffers. As a simple illustration, the following interconnect equations might be defined for a dictionary consisting of two target functions:

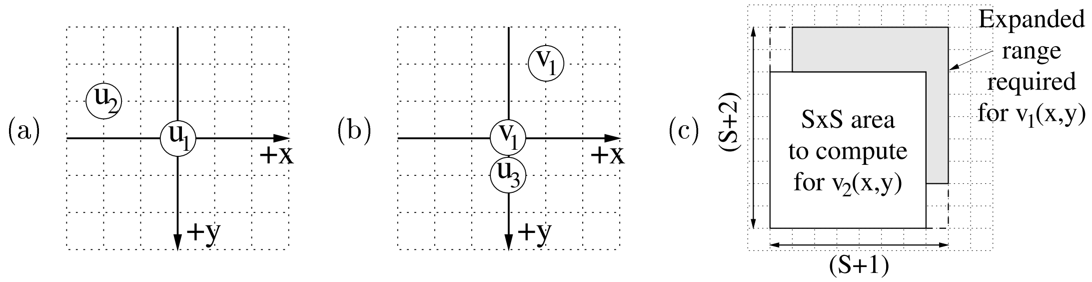
$$v_1(x, y) = u_1(x, y) + 0.5u_2(x - 2, y - 1) \tag{6.9}$$

$$v_2(x, y) = v_1(x, y) + 0.25v_1(x + 1, y - 2) + 0.25u_3(x, y + 1) \tag{6.10}$$

Note that target $v_1(x, y)$ is the weighted combination of two elementary functions. Target $v_2(x, y)$ illustrates dependency. It combines two translates of the first target with a third elementary function. The exact combination of dictionary functions, (x, y) displacements, and weights is found by orthogonal matching pursuit at the generation stage, as described in Figure 6.4.

Matching pursuit residual coding uses a local atom search. This requires target inner products to be computed in an $S \times S$ region around an energy peak. However, our generalized dependency structure requires an expanded range of inner products to be computed for some targets. This is illustrated in Figure 6.12. Figures 6.12a and 6.12b show the spatial arrangement of contributions to $v_1(x, y)$ and $v_2(x, y)$, respectively. For example, Figure 6.12a shows graphically that each value of $v_1(x, y)$ depends on $u_1(x, y)$ and $u_2(x - 2, y - 1)$. These values are always available, since elementary inner product buffers cover the entire residual image. The case is a bit more complicated for $v_2(x, y)$, since it requires both $v_1(x, y)$ and $v_1(x + 1, y - 2)$. Suppose the local atom search requires inner

Figure 6.12: An illustration of target range expansion. (a) Spatial contributions to $v_1(x, y)$. (b) Spatial contributions to $v_2(x, y)$. (c) Expansion of v_1 range required for computation of an $S \times S$ region of v_2.

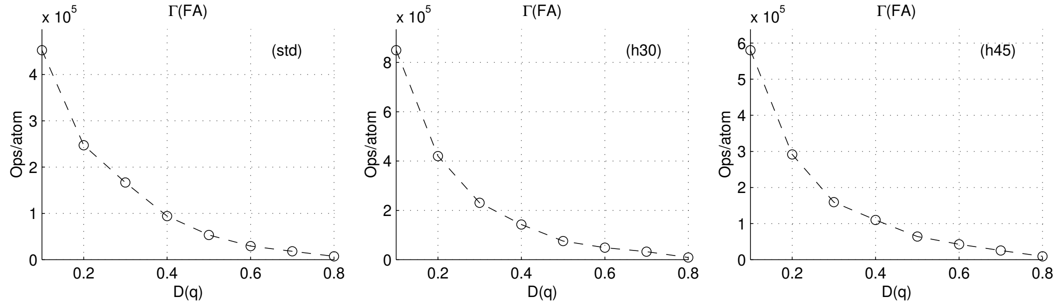


products for v_2 to be computed in the $S \times S$ region shown in Figure 6.12c. Each point within this area depends on $v_1(x + 1, y - 2)$, and so an expanded range of inner products for v_1 must be computed in order to provide values needed for computing the required v_2 range. Such range expansion is a consequence of dependency. We compute expanded ranges wherever needed at the implementation stage, and we count the resulting operations in our complexity analysis.

As mentioned in the previous subsection, the values in elementary buffers $u_a(x, y)$ must be renormalized for use in the interconnect equations. We do this efficiently by building the renormalization factors into the interconnect equation weights. This simply requires the weight for each elementary contribution in Equations (6.9) and (6.10) to be multiplied by $||\vec{s}_a||^{-1}$, where $\vec{s}_a$ is the appropriate 2-D elementary dictionary function.

A second weight adjustment is done to eliminate one multiplication from each interconnect equation. If the first weight in a particular equation is c, then we scale all weights by $1/c$. The first weight thus becomes unity and a multiplication is eliminated. However, the inner products produced by the equation and stored in $v_b(x, y)$ are now scaled by $1/c$. For atom finding, the winning position is selected within buffer $v_b(x, y)$ based on the

Figure 6.13: Interconnect complexity $\Gamma(FA)$ as a function of $D(q)$.



largest scaled inner product. This winning value is multiplied by c to correct the scale, and then compared to the scale-corrected winners from other target buffers in order to select the next atom. Rescaling thus reduces complexity. However, dependency requires an additional correction to be made. Because the values in $v_b(x, y)$ are scaled by some factor $1/c$, any contributions from $v_b(x, y)$ to later dependent targets must be adjusted by multiplying the corresponding interconnect weights by c. All weight adjustments are done in advance and thus do not adversely affect run-time encoder complexity.

Figure 6.13 plots interconnect complexity $\Gamma(FA)$ as a function of $D(q)$ for each of our three target dictionaries. A 16×16 local atom search area is used, as in [45]. Savings due to rescaling is considered in the analysis, as is the cost of all required target range expansion. As seen in the plot, $\Gamma(FA)$ for "std" ranges from a high of about 450,000 operations per atom at $D(q) = 0.1$ to a low of about 7200 for $D(q) = 0.8$. Although $\Gamma(FA)$ is not the only source of inner product complexity, it is the source which depends most greatly on $D(q)$. This property is important, since it allows approximation quality to be traded for complexity in the final encoder implementation.

Elementary Update

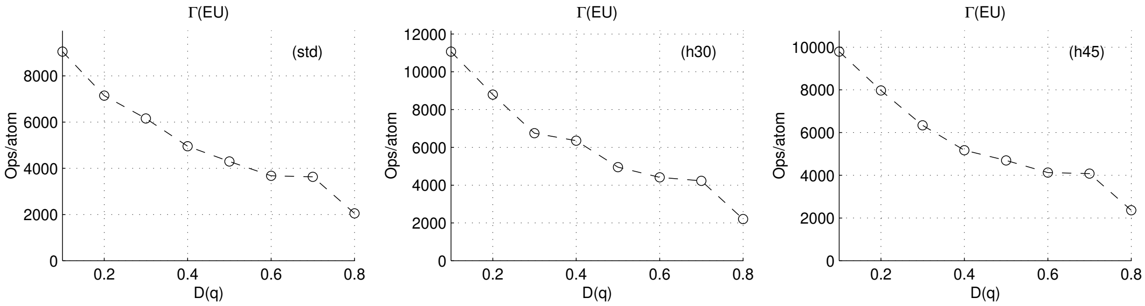
After each atom is found and subtracted, the affected elementary buffer values must be updated. The update is done using a method introduced in [55]. Each inner product is updated as

$$u_a(x, y) = u_a(x, y) - Q(p_i)\, C_{(\Delta x, \Delta y)}(\vec{s}_a, \vec{t}_{(q,b)}) \tag{6.11}$$

where $Q(p_i)$ is the quantized modulus of the current atom, $\vec{t}_{(q,b)}$ is the approximated target which was selected as the current atom, and $\vec{s}_a$ is the elementary function associated with buffer $u_a(x, y)$. The correlation function $C_{(\Delta x, \Delta y)}(\vec{s}_a, \vec{t}_{(q,b)})$ computes the inner product between $\vec{t}_{(q,b)}$ at the coded atom location, and a copy of $\vec{s}_a$ centered at location (x, y). We write $(\Delta x, \Delta y)$ to stress that the correlation function depends only on the relative position shift between $\vec{t}_{(q,b)}$ and $\vec{s}_a$, and not on the absolute location (x, y). It is thus reasonable to pre-compute the values of $C_{(\Delta x, \Delta y)}(\vec{s}_a, \vec{t}_{(q,b)})$ in advance for all relative position shifts. If this is done, the progressive update described by Equation (6.11) requires only one multiply-accumulate per updated inner product.

Because the range of affected elementary inner products depends on the extent of the coded atom, the resulting complexity $\Gamma(EU)$ depends on the motion residual signal $\vec{f}$. However, signal independent estimates for $\Gamma(EU)$ may be computed if assumptions are made. Assume that all targets in T_q are equally likely to be selected at each atom stage. The resulting elementary update complexity is shown in Figure 6.14. We see that $\Gamma(EU)$ decreases as $D(q)$ increases. This occurs because approximated targets tend to shrink as $D(q)$ increases, as shown earlier in Figure 6.10.

Figure 6.14: Estimated elementary update complexity $\Gamma(EU)$ as a function of $D(q)$. Values are computed assuming all targets in T_q are equally likely to be selected as coded atoms.



Implementation Complexity

In the previous subsections, complexity analysis was done separately for $\Gamma(IF)$, $\Gamma(FA)$, and $\Gamma(EU)$, with the results displayed in Table 6.2, Figure 6.13, and Figure 6.14, respectively. We now show that for a given coding scenario, this data may be combined in order to compute a signal independent complexity estimate. This is useful because it depends entirely on information known at the generation stage. $D(q)$ may thus be be adjusted at generation until the complexity estimate matches the desired encoder complexity.

To illustrate the complexity estimation, we return to the simple example used to evaluate Redmill's dictionary in Section 6.2.3. Suppose we are coding a QCIF sequence at about 100 atoms per frame using the "std" dictionary approximated to $D(q) = 0.5$. Initial filtering complexity $\Gamma(IF)$ is 1.4 million operations per frame, as given in Table 6.2. Interconnect complexity $\Gamma(FA)$ is read from Figure 6.13 as 53000 operations per atom. Likewise, elementary update complexity $\Gamma(EU)$ is read from Figure 6.14 as 4300 operations per atom. Total complexity may thus be computed as

$$\Gamma(IF) + 100 \cdot (\Gamma(FA) + \Gamma(EU)) = 7.1 \ million \ ops/frame \tag{6.12}$$

For the given coding scenario, the resulting dictionary is about 48 times faster than an efficient 2-D separable implementation of "std" [45]. It is also 3.16 times faster than Redmill's dictionary, according to the analysis given in Section 6.2.3.

While complexity estimates are useful at the generation stage, final evaluation of the method must be based on coding real sequences in order to measure both implementation complexity and coding efficiency. Such results are presented in the next section.

6.4 Experimental Results

In this section, we present encoding results for dictionary approximation. We test the method using the MPEG-4 test sequences and conditions as summarized in Table 1.2. The bit rate targeting method described in Chapter 2 is used to match the rate of each coded frame to the same experimental rate traces as were used for the coding efficiency comparisons in Chapters 2 and 3. For a given frame, atoms are coded until a target number of bits is reached, and the actual bit rate is typically within one or two percent of the target. Because the rate is equalized in this way, a fair quality comparison can be made across coding experiments using different approximated dictionaries.

We will test approximations of each of the three dictionaries introduced in Section 6.3.2. As a preliminary step, we show average Y-PSNR results for each original dictionary without approximation in Table 6.3. Recall that "h30" and "h45" are formed by adding oriented functions to the 2-D separable "std" dictionary. The differences shown in the final two columns of the table show that including these oriented functions produces small PSNR gains for most sequences. The gains are largest for sequences such as Mother and Foreman which contain many diagonally oriented edges. For these sequences, gains

Table 6.3: Average luminance PSNR for each sequence encoded using each original dictionary without approximation.

		Y-PSNR (dB)			Y-PSNR Diff (dB)	
Seq	Label	std	h30	h45	(h30-std)	(h45-std)
1	Cont-10	31.44	31.43	31.47	-0.01	0.03
2	Hall-10	31.86	31.86	31.86	0.00	0.00
3	Mom-10	33.22	33.42	33.42	0.20	0.20
4	Cont-24	34.41	34.32	34.32	-0.10	-0.09
5	Silent-24	32.33	32.41	32.42	0.08	0.09
6	Mom-24	35.89	36.02	35.86	0.13	-0.03
7	Coast-48	30.26	30.26	30.26	0.00	0.00
8	Fore-48	31.53	31.77	31.81	0.23	0.28
9	News-48	32.40	32.60	32.51	0.20	0.11
10	Coast-112	27.57	27.61	27.62	0.04	0.05
11	Fore-112	30.27	30.52	30.56	0.25	0.29
12	News-112	35.60	35.66	35.61	0.06	0.01
13	Mob-1024	27.15	27.22	27.21	0.08	0.06
14	Stef-1024	29.98	30.15	30.01	0.17	0.03

between 0.2 and 0.3 dB can be seen.

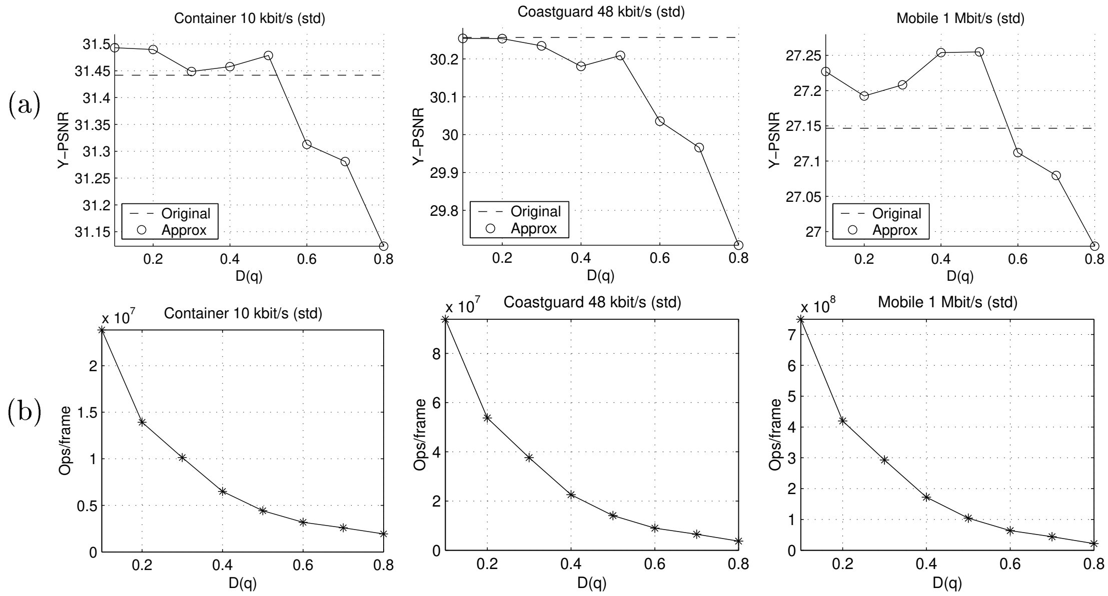
Each original dictionary was approximated to each value of $D(q)$ in $\{.1, .2, .3, .4, .5, .6, .7, .8\}$, resulting in 24 different approximated dictionaries. Each dictionary was then efficiently implemented as described in Section 6.3.3. Each resulting encoder was then used to encode all 14 MPEG-4 test sequences. Since each approximate encoder uses a different number of dictionary functions, a new set of VLC tables must be trained for each. We generate these VLC tables using a training set composed entirely of sequences from outside the MPEG-4 test set.

Figure 6.15 shows PSNR and complexity results for three sequences encoded using various approximations of the "std" dictionary. In Figure 6.15a, average Y-PSNR is plotted against $D(q)$ for each sequence. The result for the original dictionary without approximation is shown as a dotted line. The plots show that for small values of $D(q)$, the approximated dictionary performs about as well as the original. The approximate dictionary may slightly exceed the original in PSNR, as seen in the Container and Mobile plots. This occurs because the merging of similar target functions may increase the coding efficiency of the dictionary. The approximate dictionaries are close to the originals in performance for $D(q)$ values up to about 0.6. Beyond this point, Y-PSNR decreases more rapidly with $D(q)$.

Companion plots of complexity vs. $D(q)$ are provided in Figure 6.15b. Complexity is computed as average operations per frame required for inner product computation. Analysis is similar to that presented in Section 6.3.3, but complexity sources $\Gamma(IF)$, $\Gamma(FA)$ and $\Gamma(EU)$ are now computed from actual encoding statistics. As expected, complexity decreases as $D(q)$ increases. The complexity of the original "std" dictionary cannot be shown without greatly changing the plot scale. We instead note that using the efficient 2-D separable implementation in [45], the original complexity would be 1.66×10^8, 6.79×10^8, and 5.50×10^9 operations per frame for Container 10 kbit/s, Coast Guard 48 kbit/s, and Mobile 1 Mbit/s, respectively. In each case, approximation with $D(q) = 0.1$ reduces complexity by a factor of about 7 compared to an efficient implementation of the original "std" dictionary. Larger values of $D(q)$ result in still larger reductions in complexity.

The tradeoffs due to approximation are more clearly shown in Figure 6.16. The vertical axis of each plot shows the Y-PSNR loss due to approximation. This is found by subtracting the Y-PSNR of the original dictionary from that of each approximated dictionary. In the context of Figure 6.15a, the dotted line is subtracted from the solid line. On the horizontal axis of Figure 6.16, we plot the speedup factor due to the approximation. This is the complexity of the original dictionary divided by that of the approximated dic-

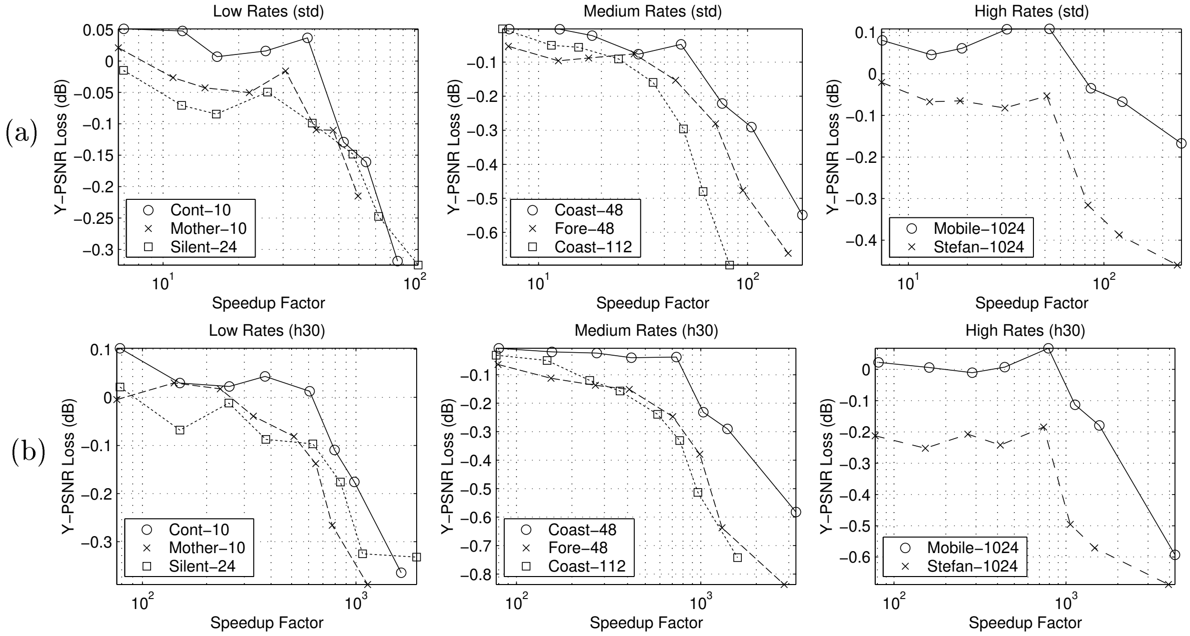
Figure 6.15: Encoding results for "std" dictionary approximated to various distortion levels $D(q)$. (a) Average Y-PSNR for original and approximated dictionaries. (b) Complexity of approximate dictionary implementations measured as average operation count per frame.



tionary. Each point on each plot represents the encoding of a single test sequence using one of our approximated dictionaries. Figure 6.16a shows the tradeoffs for the "std" dictionary. Here, original dictionary complexity is computed assuming the efficient 2-D separable implementation in [45]. The plots show that speedup factors above 40 are typically reached in exchange for PSNR losses of 0.1 dB. Larger speedup factors are achieved in exchange for additional PSNR loss. At higher rates, dictionary approximation is shown to reduce complexity by more than two orders of magnitude for large $D(q)$.

While the "std" dictionary is based on an efficient 2-D separable structure, dictionary approximation does not require any such structure of the original dictionary. In fact, the method may be used to generate efficient implementations of arbitrary dictionaries. We illustrate this by showing results for the "h30" dictionary. This dictionary is not 2-D separable, since it contains the oriented functions shown in Figure 6.7a. For the original "h30"

Figure 6.16: Loss in Y-PSNR due to approximation error plotted directly against speedup factor. (a) Coding results using the "std" dictionary. (b) Results using "h30" dictionary.



dictionary before approximation, we thus resort to computing full 2-D inner products. The original dictionary is thus expensive to implement, and so the complexity gains due to approximation are larger. This is reflected in Figure 6.16b, which shows the coding efficiency and complexity tradeoffs for various sequences encoded with "h30". The plots show that speedup factors greater than 500 are often seen in exchange for PSNR losses between 0.1 and 0.2. Speedup factors beyond 1000 are also possible, although the associated PSNR loss varies significantly by sequence.

Given curves of the form seen in Figure 6.16, it is possible to determine the largest speedup factor achievable for a given Y-PSNR loss. These values are presented in Table 6.4. The first set of three columns shows the speedup factors corresponding to a 0.1 dB approximation loss. Typical speedup factors for "std" range from 25 to 50. Larger factors between 150 and 800 are typical for the "h30" and "h45" dictionaries. The Mobile sequence

Table 6.4: Speedup factors attainable for approximation loss threshholds of 0.1 and 0.25 dB.

		0.1 dB loss			0.25 dB loss		
Seq	Label	std	h30	h45	std	h30	h45
1	Cont-10	49.6	780	665	76.3	1238	995
2	Hall-10	34.0	184	446	49.8	797	729
3	Mom-10	39.7	557	513	60.0+	759	677
4	Cont-24	25.8	796	765	70.3	1621	1571
5	Silent-24	39.5	637	548	73.0	960	909
6	Mom-24	15.5	373	925	59.1	1063	1536
7	Coast-48	56.4	830	766	87.5	1145	1098
8	Fore-48	33.8	135	459	64.3	711	666
9	News-48	9.6	77	155	35.5	266	343
10	Coast-112	25.7	220	283	44.6	606	546
11	Fore-112	23.2	144	171	41.0	420	493
12	News-112	8.2	307	472	43.5	705	669
13	Mob-1024	165.4	1104	1369	250.0+	2029	3470+
14	Stef-1024	56.8	–	781	75.0	808	918

is particularly resistant to approximation loss, allowing speedup factors beyond 1000 for less than a 0.1 dB loss. The second set of columns shows the result for a larger 0.25 dB loss threshhold. Associated speedup factors typically fall between 40 and 80 for the "std" dictionary, and between 400 and 2000 for "h30" and "h45".

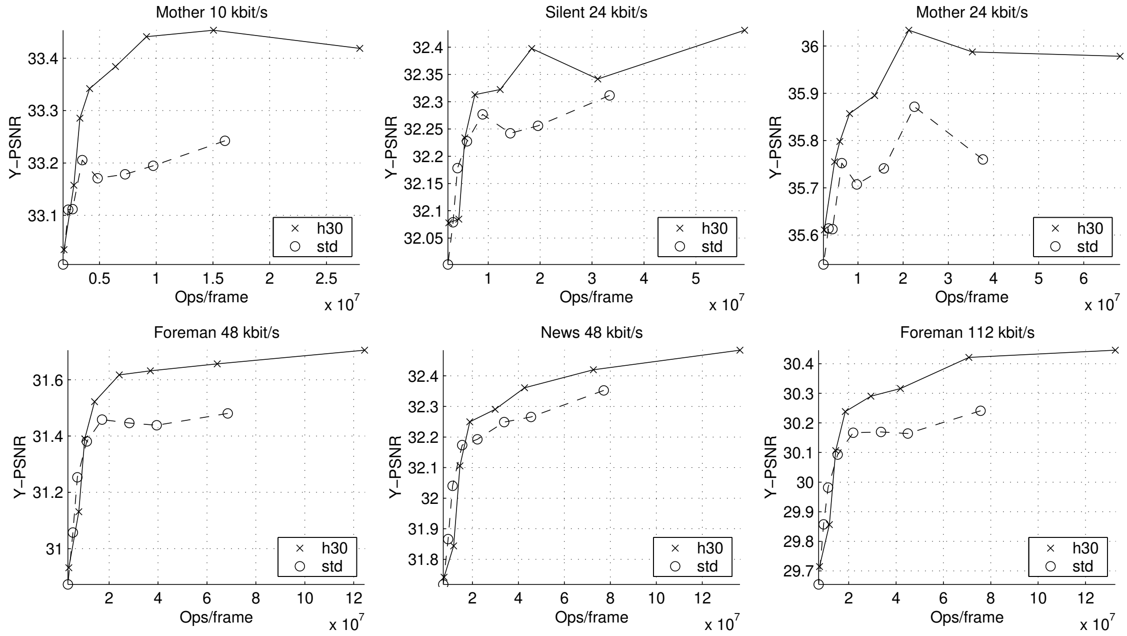
Note that in some cases, the Y-PSNR loss stays below the 0.25 dB threshhold for all tested approximation levels. This occurs for Mother-10 with the "std" dictionary, and also for Mobile-1024 with the "std" and "h45" dictionaries. A "+" symbol is used in Table 6.4 to distinguish these cases. This indicates that speedup factors beyond the listed values are likely attainable.

While Figure 6.16 and Table 6.4 provide a compact summary of the coding efficiency and complexity tradeoffs, they do not allow a direct comparison between approximations of different original dictionaries. For this purpose, we provide Figure 6.17. Each subplot of the figure plots Y-PSNR against complexity for the encodings of a single test sequence using approximations of the "std" and "h30" dictionaries. The full range of approximation levels is shown with the rightmost data point on each curve corresponding to $D(q) = 0.1$. We see that "h30" shows small Y-PSNR gains over "std" at the same complexity for mild approximation levels, e.g. $D(q) < 0.5$. This is because the oriented functions in "h30" are more effective in encoding the diagonal edge content in each sequence. The curves tend to merge beyond $D(q) = 0.5$, indicating similar PSNR vs. complexity performance. This can be attibuted to the fact that when $D(q)$ is large, few dictionary functions survive the pruning process. Approximations of "std" and "h30" tend to become more similar through pruning and approximation error, and so have similar tradeoffs. Note that the sequences in Figure 6.17 were selected because they contain significant diagonal edge content. For sequences like Container and Coast which lack diagonal edges, the "h30" dictionary shows no advantage over "std". This is evident from the earlier PSNR results given in Table 6.3.

We now illustrate the visual coding performance of the method for Foreman at 112 kbit/s. Six images of Frame 130 are shown in Figure 6.18. The rate targeting method described earlier in this section insures that the bit rate for a given frame is approximately constant regardless of dictionary or approximation level. For the particular frame detailed in the figure, the five different encodings are each within two percent of a common target budget of 8924 bits.

Figure 6.18a shows the original frame, and Figures 6.18b and 6.18c show encoded frames using original "std" and "h30" dictionaries, respectively. The additional oriented

Figure 6.17: Direct comparison of Y-PSNR vs. complexity between approximations of "std" and "h30" dictionaries. Data points for each curve represent $D(q)$ ranging from 0.1 to 0.8 increasing from right to left.



functions present in "h30" are seen to produce visual improvements in the coded frame. For example, the ear in Figure 6.18c is clearer and more detailed than that seen in 6.18b. Diagonal lines on Foreman's neck and shirt collar are also much improved. Ringing is reduced in the diagonal lines present on the building, and the shadow line above Foreman's right shoulder is much sharper. This visual improvement comes at a price in complexity. While the 2-D separable cost of the "std" dictionary encodes the sequence at an average of 5.08×10^8 operations per frame, the general nonseparable implementation of "h30" requires 1.02×10^{10} operations per frame. The implementation of "h30" is thus about 20 times more expensive than "std".

Encoding results using approximations of the "h30" dictionary are shown in Figures 6.18d-f. Figure 6.18d was encoded with $D(q) = 0.1$, and appears quite similar to the result from the original dictionary in Figure 6.18c. Figure 6.18e was encoded using $D(q) = 0.3$. Some small degradations due to the approximation can be seen, particularly in the mouth area. However, the advantages of the original "h30" dictionary are mostly still intact. These include sharp diagonal lines on the building, and more detail in the ear and collar regions. Finally, Figure 6.18f shows the result for $D(q) = 0.6$. By this point, some loss of quality is clearly seen on the face. However, many parts of the image are still visually better than the "std" encoding shown in Figure 6.18b.

Average complexities for the sequences shown in Figures 6.18d, 6.18e, and 6.18f are respectively 1.33×10^8, 4.19×10^7, and 1.86×10^7 operations per frame. The approximations are respectively 3.83, 12.1, and 27.4 times less complex than the 2-D separable implementation of "std" which produced Figure 6.18b. They are also respectively 77.0, 244 and 550 times less complex than the nonseparable implementation of "h30" which produced Figure 6.18c. Our results illustrate a simple application of the new design paradigm of Figure 6.1b. By allowing "h30" to contain nonseparable functions, both PSNR and visual quality were improved. The resulting dictionary is not based on an efficient computational structure. However, by applying dictionary approximation, low-cost approximate implementations of "h30" were automatically generated.

6.5 Conclusions

In this chapter, we developed a method for generating approximations of arbitrary dictionaries with fast 2-stage implementations. For a given dictionary, this allows coding efficiency to be traded for encoder complexity in a systematic way. It also enables a

Figure 6.18: Visual encoding results for frame 130 of Foreman at 112 kbit/s. (a) Original image. (b) Encoded using "std" dictionary. (c) Using "h30" dictionary. (d) Using "h30" dictionary approximated with $D(q) = 0.1$. (e) Using "h30" dictionary approximated with $D(q) = 0.3$. (f) Using "h30" dictionary approximated with $D(q) = 0.6$.

new dictionary design paradigm which is free from initial restrictions based on complexity concerns. We applied the method to the 2-D separable "std" dictionary, and showed that complexity reduction factors between 40 and 80 are typical in exchange for small reductions in coding efficiency. We showed that larger gains are possible when the method is applied to a more general dictionary without an efficient computational structure. For such dictionaries, complexity reduction factors ranging from 100 to 1000 were shown in exchange for a 0.1 dB approximation loss. Dictionary approximation may thus be employed to efficiently implement matching pursuit using dictionary functions of arbitrary design.