SMPTE ST 2042-1 (VC-2) Bit Widths

Release 0.1.6

BBC

Mar 19, 2021

CONTENTS

CHAPTER

ONE

PREFACE

The *vc2_bit_widths* (page 33) Python package provides routines for computing how many bits of numerical precision are required for implementations of the SMPTE ST 2042-1:2017[1] VC-2 professional video codec[2]. In addition it also provides routines for producing test pictures which produce large signal values in actual video codecs.

This manual is split into three parts. In *User's Manual* (page 5) a general introduction to the purpose, terminology and usage of this module is given. In *Theory and Design* (page 51), the underlying theory and mathematical approach are described and evaluated. Finally, *Internals and Low-Level API* (page 79) gives a more detailed overview of the implementation and lower-level features of this software.

Finally, you can find the source code for *vc2_bit_widths* (page 33) on GitHub[3].

Note: This documentation is also available to browse online in HTML format[4].

[1] https://ieeexplore.ieee.org/document/7967896
[2] https://www.bbc.co.uk/rd/projects/vc-2
[3] https://github.com/bbc/vc2_bit_widths/
[4] https://bbc.github.io/vc2_bit_widths/

Part I

User's Manual

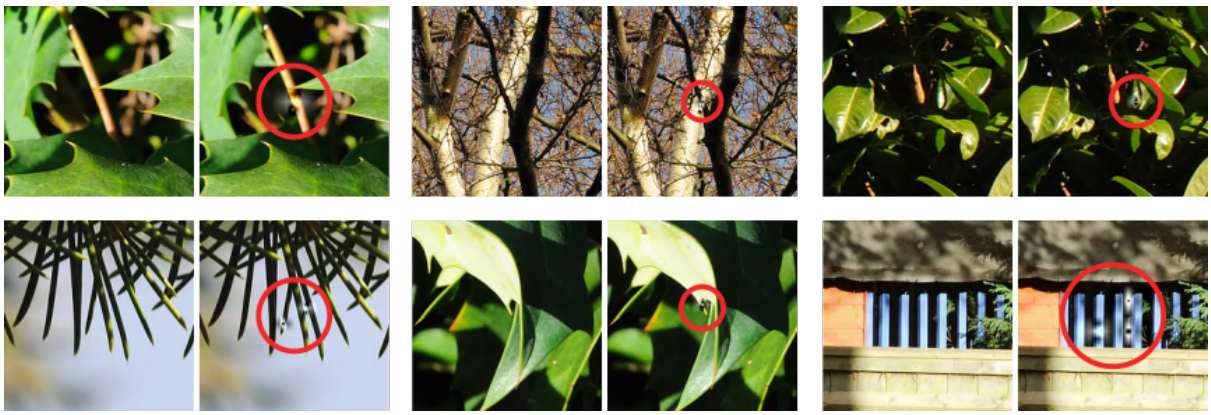
CHAPTER

TWO

INTRODUCTION

The VC-2 standard defines the video decoding process using infinite precision integer arithmetic. For practical implementations, however, fixed-width integers must be used to achieve useful performance.

If a codec is built with too few bits of precision, artefacts may be produced in the event of integer wrap-around or saturation (see examples below). If too many bits are used, however, the implementation will consume more resources than necessary.



Perhaps surprisingly, the question 'how many bits do I need?' is not a simple one to answer. This software attempts to provide useful estimates for these figures based on mathematical analyses of VC-2's filters

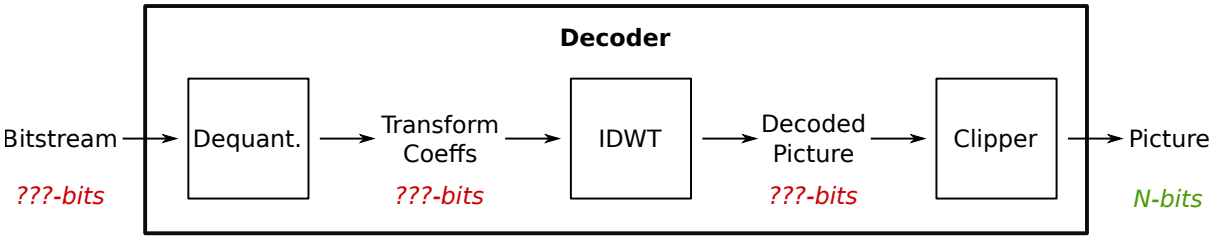
Before introducing this software it is important to understand its limitations and the terminology it uses. These will be introduced in the next few sections before the command line and Python library interfaces of *vc2_bit_widths* (page 33) is introduced.

2.1 Caveats

While this software aims to produce robust bit width figures, it can only go as far as the VC-2 specification and current mathematical techniques allow.

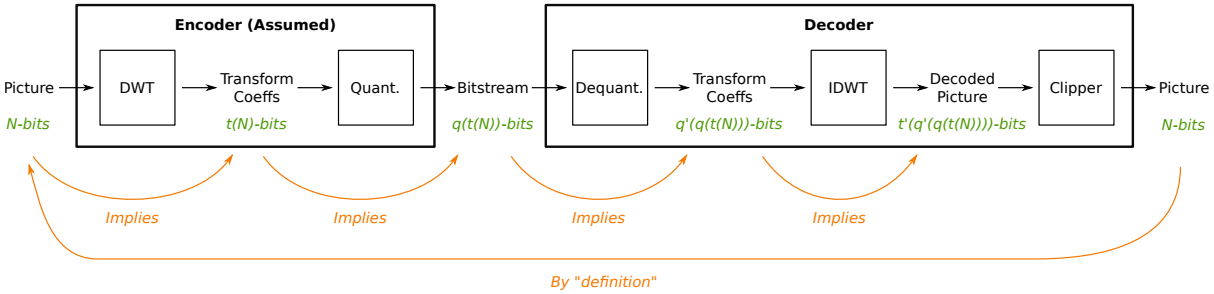
2.1.1 Assumed encoder behaviour

The VC-2 standard only defines the behaviour of a decoder. Unfortunately, due to the clipper at the output, it is not possible to work backwards from the output bit width to calculate the input or intermediate signal ranges (as illustrated below).



The standard also does not define legal ranges for values in a bitstream (which would allow us to work forwards through the decoder). Instead we must make some assumptions about the behaviour of VC-2 encoders which will allow us to determine the ranges of values which could appear in real bitstreams.

This software makes the assumption that all VC-2 encoders consist of a (forward) discrete wavelet transform followed by a dead zone quantiser implemented as informatively suggested by the standard. Once this assumption has been made it becomes possible to determine the bit widths of every part of a VC-2 encoder and decoder.



In principle, VC-2 encoder implementations are free to diverge from this assumed behaviour and so may produce bitstreams with different signal ranges to those predicted by this software. In practice, it is relatively unlikely to be the case. Nevertheless, you should be aware that this software relies on this assumption.

2.1.2 Non-linearity

Though VC-2 is based on the (linear) wavelet transform, its use of integer arithmetic and quantisation makes VC-2 a non-linear filter. Due to the difficulty of analysing large non-linear filters, this software is generally unable to give exact answers about the numbers of bits required and instead are given as a range of possible values.

The upper-bounds on signal levels and bit widths are computed by this software using robust mathematical models of VC-2's filters and are guaranteed not to be under-estimates. However, these upper bounds can be (sometimes significant) over-estimates of the signal levels produced by true worst-case signals.

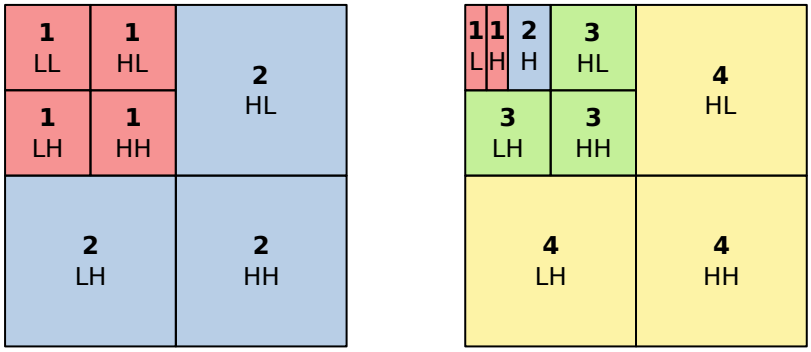
Likewise, lower-bounds on signal level ranges are provided based on the signal levels exhibited during the encoding and decoding of specially created test patterns. Since these test patterns are valid pictures (or the result of encoding valid pictures), they represent concrete lower-bounds on signal levels. Despite producing more extreme signals than natural pictures or noise in general, the test patterns are not guaranteed to elicit worst-case signal levels. As such the signal levels produced merely set a lower-bound on the true worst-case.

2.2 Terminology

This software uses the following naming conventions to identify the numerous intermediate values within VC-2 analysis (encoding) and synthesis (decoding) filters.

2.2.1 Levels and subbands

Levels and subbands are numbered similarly to the VC-2 specification with the exception of the DC-band being listed as part of level '1', and not level '0'.



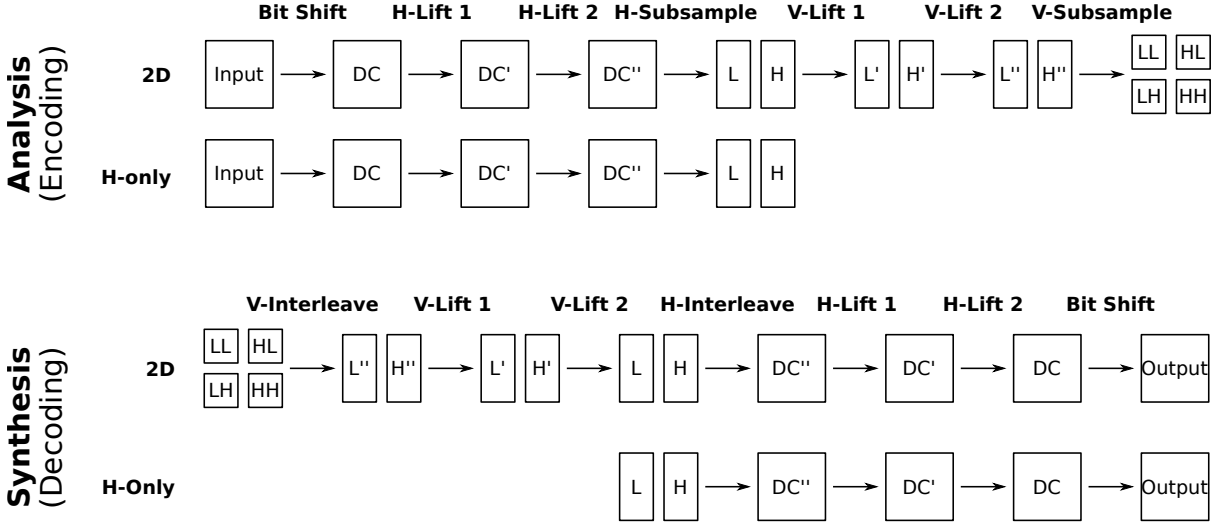
dwt_depth = 2
dwt_depth_ho = 0

dwt_depth = 2
dwt_depth_ho = 2

For a given transform level, the filtering process is broken down into a series of steps which transform an array of input values into several (subsampled) arrays during analysis (encoding) and the reverse during synthesis (decoding).

2.2.2 Intermediate value arrays

The naming convention used for transform levels and subbands is extended to give names to every intermediate array of transform values in a multi-level transform. Within a single transform level, the arrays are named as illustrated below:



Note: In diagrams above two lifting stages are shown for each filter. For filters with more than two lifting stages, the outputs of these stages follow the same pattern. For example, for a Daubechies (9, 7) filter, which has four lifting stages, the additional arrays are named `DC'''`, `DC''''`, `L'''`, `L''''`, `H'''` and `H''''`.

For example, in a 2-level 2D analysis transform, we can use these names like so:

- The original input picture would be completely identified as 'Analysis, Level 2, Input'
- The array following the initial bit-shift operation is 'Analysis, Level 2, DC'.
- The final results after horizontal and vertical lifting filters would be identified as 'Analysis, Level 2, LL', 'Analysis, Level 2, LH', 'Analysis, Level 2, HL' and 'Analysis, Level 2, HH'.

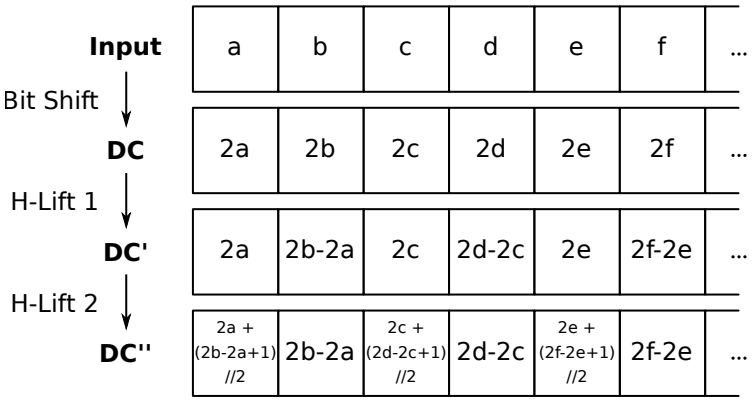
- The input to the next analysis transform is 'Analysis, Level 1, Input' and is exactly the same array as 'Analysis, Level 1, LL', just renamed for consistency.

The synthesis transform is analogous. For example, the final decoded picture would be identified as 'Synthesis, Level 2, Output' in a 2-level transform.

2.2.3 Phases

Each of the intermediate arrays of values within a synthesis or analysis transform may be considered to be the result of some a 2D polyphase filter on the original input picture or transform coefficients. As such, each value in the array may be thought of as being the result of a particular phase of the filter, each with its own filtering characteristics. Since there are only a finite number of filter phases, we can completely characterise the process which led to a particular array using only one value per filter phase.

For example, consider a horizontal only 1-level Haar analysis transform on a 1D image:



It should be easy to see the repeating patterns of operations in the different arrays in the example.

For instance, in the DC' array, even numbered entries take the form $2\ \text{Input}[n]$ while odd numbered entries take the form $2\ \text{Input}[n] - 2\ \text{Input}[n-1]$. As a result, the DC' array may be described as having two phases or, alternatively, as having a period of two. By contrast the DC array has a period of one, since all entries are alike (taking the form $2\ \text{Input}[n]$).

In the more general case of 2D filters processing 2D pictures, the period of a particular array is given as two numbers, e.g. (2, 4), giving the period in the X dimension and the period in the Y dimension respectively. Likewise, individual phases are also identified by an X and Y value.

Using these conventions, a particular phase of a particular intermediate array of values may be named. For example, 'Analysis, Level 1, DC', 0, 1' identifies the filter phase X=0, Y=1 at the output of the bit shift operation of the final transform level of the analysis filter.

2.2.4 Targets

For the purposes of analysing the filter behaviour leading to a particular array, it is sufficient to analyse one example of each filter phase, and this is exactly what this software does. When it comes to generating test pictures, however, this software may not always pick the top-leftmost example of a particular filter phase to test. The specific example chosen is referred to as the 'target' value.

Since all instances of a particular filter phase are alike, using a different example is still equivalent if it has the same phase.

There are various reasons why a test picture might not pick the top-left most example of a particular filter. For example, consider the problem of generating a test picture which tests both phases of the DC" array in the Haar example above. To test phase 0 of the DC" array we pick the zeroth element as our target and use inputs 'a' and 'b' to test filter. If we wish to simultaneously test phase 1 of this array, we can't use element 1 as the target because this also depends on inputs 'a' and 'b', so instead we must pick element 3 (which uses inputs 'c' and 'd') as the target instead.

CHAPTER

THREE

TUTORIAL

Use of this software is generally divided into two steps:

- Performing static filter analysis
- Calculating bit width requirements or generating test pictures

In the static filter analysis step, a mathematical analysis of a specific wavelet and transform depth is performed. The result of this analysis is a series of test patterns and abstract mathematical descriptions of the expected signal range.

In the second step, the output of the static analysis is specialised further for a particular picture bit width and quantisation matrix from which concrete signal ranges, test patterns and other information can be computed.

This software provides both command line tools and a Python API (*vc2_bit_widths* (page 33)) with which these tasks may be performed. The command line interface will be demonstrated in the examples below.

3.1 Static filter analysis

Static filter analysis is typically performed using the *vc2-static-filter-analysis* (page 15) command line utility like so:

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json
```

In the example above, the static analysis is performed for a 2-level LeGall (5, 3) wavelet transform and written to `static_analysis.json`.

Asymmetric transforms can also be specified using the `--wavelet-index-ho` and `--dwt-depth-ho` arguments.

For most wavelets and transform depths (as in the example above), the static analysis process completes after a few seconds. For very large wavelets and transform depths, however, the process can take several hours. To monitor the progress the `--verbose` argument may be used.

The JSON files produced by `vc2-static-filter-analysis` follow the format described in *JSON file format* (page 16). This file contains theoretical worst-case signal levels in algebraic form along with test patterns for every intermediate value array and filter phase. In this form, however, everything is quite abstractly defined and we must process this file further to produce more useful results.

3.2 Calculating bit-width requirements

The *vc2-bit-widths-table* (page 20) command line utility may be used to compute a table of concrete signal ranges and bit width requirements:

```
$ vc2-bit-widths-table \
    static_analysis.json \
    --picture-bit-width 10 \
    --output bit_widths_table.csv
```

In this example, we use results of the static analysis produced in the previous step to find the signal ranges and bit widths when 10 bit input pictures and the default quantisation matrix are used. (Custom quantisation matrices may be specified using the `--custom-quantisation-matrix` argument.)

The `bit_widths_table.csv` file produced may be opened using any spreadsheet package (e.g. Microsoft Excel) and contains a table as illustrated below:

type	level	array_name	lower_bound	test_pattern_min	test_pattern_max	upper_bound	bits
analysis	2	Input	-512	-512	511	511	10
analysis	2	DC	-1024	-1024	1022	1022	11
analysis	2	DC’	-2047	-2046	2046	2047	12
analysis	2	DC”	-2047	-2046	2046	2047	12
analysis	2	L	-1537	-1535	1534	1535	12
analysis	2	H	-2047	-2046	2046	2047	12
analysis	2	L’	-3071	-3069	3069	3071	13
analysis	2	H’	-4094	-4092	4092	4094	13
…	…	…	…	…	…	…	…
synthesis	2	L’	-26806	-4888	4888	26806	14-16
synthesis	2	H’	-6929	-5167	5167	6929	14
synthesis	2	L	-26806	-4888	4888	26806	14-16
synthesis	2	H	-9513	-4345	4345	9513	14-15
synthesis	2	DC”	-26806	-4888	4888	26806	14-16
synthesis	2	DC’	-30271	-4888	4888	30271	14-16
synthesis	2	DC	-30271	-4888	4888	30271	14-16
synthesis	2	Output	-15136	-2444	2444	15136	13-15

Each row in the table describes a different intermediate value array within a VC-2 analysis or synthesis filter, as identified by the ‘type’, ‘level’ and ‘array_name’ columns (see *Terminology* (page 6)). (If required, the table can be broken down further into individual phases using the `--show-all-filter-phases` argument.)

The ‘lower_bound’ and ‘upper_bound’ columns give an estimate of the worst-case signal levels which could appear in that array. This estimate is guaranteed not to under-estimate the true worst case signal levels but, in the case of synthesis filters, can sometimes be a significant over-estimate.

The ‘test_pattern_min’ and ‘test_pattern_max’ columns give actual signal values resulting from passing the test patterns generated during static analysis through a VC-2 encoder and decoder. In the case of the synthesis transform, the values reported are for whichever quantisation index produces the most extreme value.

The final column gives the number of bits required for a correct VC-2 implementation. This value may be given as a range in the case where the test patterns and theoretical worst case differ significantly. The true bit width

requirement is guaranteed to lie somewhere within that range. (See *Caveats* (page 5)).

3.3 Bounding quantisation indices

The VC-2 specification does not put an upper bound on the quantisation indices which might be used. The *vc2-maximum-quantisation-index* (page 25) utility uses the theoretical bounds of the analysis filter (encoder) outputs to determine the largest quantisation index which could sensibly be used for a particular picture bit depth and quantisation matrix:

```
$ vc2-maximum-quantisation-index \
    static_analysis.json \
    --picture-bit-width 10
55
```

As before, custom quantisation matrices may be specified using the `--custom-quantisation-matrix` argument, otherwise the default quantisation matrix will be assumed.

3.4 Optimising synthesis test patterns

The *vc2-optimise-synthesis-test-patterns* (page 26) command attempts to enhance the synthesis filter test patterns produced by `vc2-static-filter-analysis` to produce even larger signal values.

The test patterns produced by `vc2-static-filter-analysis` are the result of a heuristic designed to be likely to elicit extreme signal values, but worst-case signal levels are not guaranteed. For analysis transforms (encoding), this heuristic performs very well, however synthesis transforms (decoding) are more challenging due to the non-linearity introduced during quantisation (see *Non-linearity* (page 6)).

A stochastic optimisation algorithm is used by `vc2-optimise-synthesis-test-patterns` to manipulate the initial test pattern. This process repeatedly encodes, quantises and then decodes modified test patterns using a full implementation of the VC-2 integer filtering process. As a consequence, the optimised test signals are able to exploit quirks of the integer rounding and quantisation errors introduced by a particular codec configuration. As a result, the optimised test patterns are very tightly matched to that particular configuration, but can achieve substantial worst-case signal level increases.

The command may be used with its default parameters like so:

```
$ vc2-optimise-synthesis-test-patterns \
    static_analysis.json \
    --picture-bit-width 10 \
    --output optimised_patterns.json
```

In this example, the test patterns will be optimised for codecs operating on 10 bit pictures and using the default quantisation matrix, with the resulting test patterns being written to `optimised_patterns.json`. The `--verbose` argument may be used to give a greater indication of progress.

The level of improvement achieved, and the algorithm runtime, are highly dependent on the careful tuning of the search parameters (see *Choosing parameters* (page 28)). It may be expected that to produce useful improvements several hours of optimisation will be required.

The `vc2-bit-widths-table` command may be provided with the optimised test patterns to generate a table showing the signal ranges and bit widths reached by the optimised test signal:

```
$ vc2-bit-widths-table \
    static_analysis.json \
    optimised_patterns.json \
    --output optimised_bit_widths_table.csv
```

3.5 Generating test pictures

The *vc2-bit-width-test-pictures* (page 22) command may be used to generate a series of test pictures containing test patterns suitable for passing through a VC-2 encoder or decoder.

As a simple example, set of test patterns generated as above may be turned into a collection of HD test pictures like so:

```
$ mkdir test_pictures
$ vc2-bit-width-test-pictures \
    static_analysis.json \
    1920 1080 \
    --picture-bit-width 10 \
    --output-directory test_pictures
```

The generated test pictures contain test patterns packed together as illustrated in the example below:

The test pictures are split into analysis and synthesis test pictures.

The analysis test pictures may be fed directly to an encoder.

The synthesis test pictures are further split up into groups which should be quantised to different levels. These pictures should be individually encoded such that every picture slice is quantised with the specified quantisation index. These encoded pictures may then be fed to a decoder implementation.

See *Test picture format and usage* (page 23) for a more detailed explanation of how these test pictures should be used.

3.6 Bundling analyses and test pattern data files

When a large number of analyses have been performed (using, e.g. *vc2-static-filter-analysis* (page 15)), a correspondingly large set of analysis JSON files will also accumulate. These can be bundled together, along with any optimised synthesis test patterns test (from *vc2-optimise-synthesis-test-patterns* (page 26)) into a compressed bundle file.

As well as substantially reducing the diskspace required to store the analysis files, specific analyses and optimised test patterns may be extracted on demand using a built-in index.

The *vc2-bundle* (page 29) command may be used to create and query bundle files. For example, if you have a number of static filter analyses with filenames like `static_filter_analysis_*.json` and optimised synthesis test patterns with filenames like `optimised_synthesis_test_patterns_*.json`, a bundle can be produced using `vc2-bundle create` like so:

```
$ vc2-bundle create bundle.zip \
    --static-filter-analyses static_filter_analysis_*.json \
    --optimised-synthesis-test-patterns optimised_synthesis_test_patterns_*.json
```

Individual analyses may be extracted like so:

```
$ vc2-bundle extract-static-filter-analyses \
    bundle.zip \
    --wavelet-index haar_with_shift \
    --dwt-depth 1 \
    --output extracted.json
```

And so too can optimised synthesis test patterns:

```
$ vc2-bundle extract-optimised-static-synthesis-test-patterns \
    bundle.zip \
    --wavelet-index haar_with_shift \
    --dwt-depth 1 \
    --picture-bit-width 10 \
    --output extracted.json
```

CHAPTER

FOUR

COMMAND-LINE UTILITIES REFERENCE

This package provides a suite of command line utilities which may be used to perform all of the essential tasks involved in computing bit width requirements and generating test pictures in a stand-alone manner. Refer to *Tutorial* (page 9) for an introduction to these tools. More detailed instructions are provided below.

4.1 vc2-static-filter-analysis

This command statically analyses a VC-2 filter configuration to determine mathematical expressions for worst-case signal ranges and generate test patterns.

4.1.1 Example usage

A simple 2-level LeGall (5, 3) transform:

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json
```

A more complex asymmetric transform:

```
$ vc2-static-filter-analysis \
    --wavelet-index haar_with_shift \
    --wavelet-index-ho le_gall_5_3 \
    --dwt-depth 1 \
    --dwt-depth-ho 2 \
    --output static_analysis.json
```

4.1.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-static-filter-analysis [-h] --wavelet-index WAVELET_INDEX
                                  [--wavelet-index-ho WAVELET_INDEX_HO]
                                  [--dwt-depth DWT_DEPTH]
                                  [--dwt-depth-ho DWT_DEPTH_HO]
                                  [--num-batches NUM_BATCHES]
                                  [--batch-num BATCH_NUM] [--output OUTPUT]
                                  [--verbose]

This command statically analyses a VC-2 filter configuration to determine
mathematical expressions for worst-case signal ranges and generate test
patterns. Writes the output to a JSON file.
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --wavelet-index WAVELET_INDEX, -w WAVELET_INDEX
                        The VC-2 wavelet index for the wavelet transform. One
                        of: 0 or deslauriers_dubuc_9_7, 1 or le_gall_5_3, 2 or
                        deslauriers_dubuc_13_7, 3 or haar_no_shift, 4 or
                        haar_with_shift, 5 or fidelity, 6 or daubechies_9_7.
  --wavelet-index-ho WAVELET_INDEX_HO, -W WAVELET_INDEX_HO
                        The VC-2 wavelet index for the horizontal parts of the
                        wavelet transform. If not specified, assumed to be the
                        same as --wavelet-index/-w.
  --dwt-depth DWT_DEPTH, -d DWT_DEPTH
                        The VC-2 transform depth. Defaults to 0 if not
                        specified.
  --dwt-depth-ho DWT_DEPTH_HO, -D DWT_DEPTH_HO
                        The VC-2 horizontal-only transform depth. Defaults to
                        0 if not specified.
  --num-batches NUM_BATCHES, -B NUM_BATCHES
                        If the analysis is to be performed in a series of
                        smaller batches, the number of batches to split it
                        into.
  --batch-num BATCH_NUM, -b BATCH_NUM
                        When --num-batches is used, the specific the batch to
                        run.
  --output OUTPUT, -o OUTPUT
                        The name of the file to write the computed JSON output
                        to. Defaults to stdout.
  --verbose, -v         Show more detailed status information during
                        execution.
```

4.1.3 JSON file format

The output of `vc2-static-filter-analysis` is a JSON file which has the following structure:

```
{
    "wavelet_index": <int>,
    "wavelet_index_ho": <int>,
    "dwt_depth": <int>,
    "dwt_depth_ho": <int>,
    "analysis_signal_bounds": [<signal-bounds>, ...],
    "synthesis_signal_bounds": [<signal-bounds>, ...],
    "analysis_test_patterns": [<test-pattern-specification>, ...],
    "synthesis_test_patterns": [<test-pattern-specification>, ...],
}
```

Signal bounds

The `"analysis_signal_bounds"` and `"synthesis_signal_bounds"` lists define algebraic expressions for the upper- and lower-bounds for each analysis and synthesis filter phase (see *Terminology* (page 6)) as follows:

```
<signal-bounds> = {
    "level": <int>,
    "array_name": <string>,
    "phase": [<int>, <int>],
    "lower_bound": <algebraic-expression>,
    "upper_bound": <algebraic-expression>,
}
```

(continues on next page)

(continued from previous page)

```
<algebraic-expression> = [
    {
        "symbol": <string-or-null>,
        "numer": <string>,
        "denom": <string>,
    },
    ...
]
```

Note: The *deserialise_signal_bounds()* (page 43) Python utility function is provided for unpacking this structure.

Each `<algebraic-expression>` defines an algebraic linear expression. As an example, the following expression:

$$\frac{2}{3}a + 5b - 2$$

Would be represented in JSON form as:

```
[
    {"symbol": "a", "numer": "2", "denom": "3"},
    {"symbol": "b", "numer": "5", "denom": "1"},
    {"symbol": null, "numer": "-2", "denom": "1"},
]
```

In the expressions defining the analysis filter signal levels, the following symbols are used:

- `signal_min` – The minimum picture signal value (e.g. -512 for 10 bit signals).
- `signal_max` – The minimum picture signal value (e.g. 511 for 10 bit signals).

In the expressions defining the synthesis filter signal levels, symbols with the form `coeff_<level>_<orient>_min` and `coeff_<level>_<orient>_min` are used. For example `coeff_1_LL_min` would mean the minimum value a level-1 'LL' subband value could have.

Test patterns

The `"analysis_test_patterns` and `"synthesis_test_patterns` lists define test patterns for each analysis and synthesis filter phase like so:

```
<test-pattern-specification> = {
    "level": <int>,
    "array_name": <string>,
    "phase": [<int>, <int>],
    "target": [<int>, <int>],
    "target_transition_multiple": [<int>, <int>],
    "pattern": <test-pattern>,
    "pattern_transition_multiple": [<int>, <int>],
}

<test-pattern> = {
    "dx": <int>,
    "dy": <int>,
    "width": <int>,
    "height": <int>,
    "positive": <string>,
    "mask": <string>,
}
```

Note: The *deserialise_test_pattern_specifications()* (page 44) Python utility function is provided for unpacking this structure.

Test patterns are defined in terms of a collection of pixel polarity values which indicate which pixels should be set to their maximum level and which should be set to their minimum. All other pixel values may be set arbitrarily. For the encoding used by the `<test-pattern>` object to encode the pixel polarities themselves, see *serialise_test_pattern* (page 45).

Test patterns must be carefully aligned within a test picture when used. See *TestPatternSpecification* (page 41) for the meaning of the relevant fields of `<test-pattern-specification>`. .

Missing values

Only intermediate arrays are included which contain novel values. Arrays which are just renamings, interleavings and subsamplings of other arrays are omitted.

4.1.4 Runtime and memory consumption

For typical 'real world' filter configurations, this command should complete within a few seconds and use a trivial amount of memory.

For larger wavelets (e.g. the fidelity filter) and deeper transforms (e.g. three or more levels), the runtime and memory requirements can grow significantly. For example, a 4-level LeGall (5, 3) transform will take on the order of an hour to analyse. As an especially extreme case, a 4-level Fidelity wavelet will require around 16 GB of RAM and several weeks to complete.

The `--verbose` option provides useful progress information as the static analysis process proceeds. As a guide, the vast majority of the runtime will be spent on the synthesis filters due to their far greater number. RAM usage grows rapidly during the processing of the analysis filters and then only grow very slightly during synthesis filter analysis.

4.1.5 Batched/parallel execution

When analysing extremely large filters, it might be useful to split the analysis process into multiple batches to be executed simultaneously on several cores or computers.

To run the analysis process in batched mode, the `--num-batches` argument must be given specifying the total number of batches the execution will be split into. Each batch must also be given a `--batch-num` argument specifying which batch is to be executed. For example, the following commands might be run on four machines to parallelise the analysis of a 3-level Fidelity filter.

```
$ vc2-static-filter-analysis \
    --wavelet-index fidelity \
    --depth 3 \
    --num-batches 4 \
    --batch-num 0 \
    --output static_analysis_batch_0.json

$ vc2-static-filter-analysis \
    --wavelet-index fidelity \
    --depth 3 \
    --num-batches 4 \
    --batch-num 1 \
    --output static_analysis_batch_1.json

$ vc2-static-filter-analysis \
    --wavelet-index fidelity \
```

(continues on next page)

(continued from previous page)

```
    --depth 3 \
    --num-batches 4 \
    --batch-num 2 \
    --output static_analysis_batch_2.json

$ vc2-static-filter-analysis \
    --wavelet-index fidelity \
    --depth 3 \
    --num-batches 4 \
    --batch-num 3 \
    --output static_analysis_batch_3.json
```

Once all four analyses have finished, the results are then combined together using the *vc2-static-filter-analysis-combine* (page 19) utility:

```
$ vc2-static-filter-analysis-combine \
    static_analysis_batch_0.json \
    static_analysis_batch_1.json \
    static_analysis_batch_2.json \
    static_analysis_batch_3.json \
    --output static_analysis.json
```

Warning: Each batch may still require the same amount of RAM as a complete analysis.

Warning: The batching process requires some duplicated processing in each batch. Consequently more total CPU time may be required than non-batched execution.

Warning: Though batches are intended to take similar amounts of time to execute, this is not guaranteed.

4.2 vc2-static-filter-analysis-combine

Combine the results of several batched runs of *vc2-static-filter-analysis* (page 15) into a single set of results.

4.2.1 Example usage

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --num-batches 2 \
    --batch-num 0 \
    --output static_analysis_batch_0.json
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --num-batches 2 \
    --batch-num 1 \
    --output static_analysis_batch_1.json

$ vc2-static-filter-analysis-combine \
    static_analysis_batch_0.json \
```

(continues on next page)

(continued from previous page)

```
    static_analysis_batch_1.json \
    --output static_analysis.json
```

4.2.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-static-filter-analysis-combine [-h] [--output OUTPUT]
                                          inputs [inputs ...]

Combine the outputs of a series of batched calls to vc2-static-filter-
analysis.

positional arguments:
  inputs                A complete set of matching batched filter analysis
                        outputs.

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT, -o OUTPUT
                        The name of the file to write the computed JSON output
                        to. Defaults to stdout.
```

4.3 vc2-bit-widths-table

Compute the signal ranges and bit widths required for each part of a VC-2 analysis and synthesis filter and present the results in a CSV table.

Note: The values printed by this tool are valid only for the wavelet transform, depth, picture bit width and quantisation matrix specified. See *Caveats* (page 5) for further limitations and assumptions made by this software.

4.3.1 Example usage

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json

$ vc2-bit-widths-table \
    static_analysis.json \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3 \
    --output bit_widths.csv

$ column -t -s, bit_widths.csv | head
type       level  array_name  lower_bound  test_pattern_min  test_pattern_max ␣
↪upper_bound  bits
analysis   2      Input       -512         -512              511               511␣
↪          10
analysis   2      DC          -1024        -1024             1022              ␣
↪1022         11
```

(continues on next page)

(continued from previous page)

```
analysis   2      DC'         -2047        -2046          2046          ␣
↪2047         12
analysis   2      DC''        -2047        -2046          2046          ␣
↪2047         12
analysis   2      L           -1537        -1535          1534          ␣
↪1535         12
analysis   2      H           -2047        -2046          2046          ␣
↪2047         12
analysis   2      L'          -3071        -3069          3069          ␣
↪3071         13
analysis   2      H'          -4094        -4092          4092          ␣
↪4094         13
analysis   2      L''         -3071        -3069          3069          ␣
↪3071         13
```

The command can also be used to display the signal ranges of optimised test patterns produced by `vc2-optimise-synthesis-test-patterns`:

```
$ vc2-bit-widths-table \
    static_analysis.json \
    optimised_patterns.json \
    --output bit_widths.csv
```

4.3.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-bit-widths-table [-h] [--picture-bit-width PICTURE_BIT_WIDTH]
                            [--custom-quantisation-matrix CUSTOM_QUANTISATION_
↪MATRIX [CUSTOM_QUANTISATION_MATRIX ...]]
                            [--show-all-filter-phases] [--verbose]
                            [--output OUTPUT]
                            static_filter_analysis
                            [optimised_synthesis_test_patterns]

Compute the signal ranges and bit widths required for each part of a VC-2
analysis and synthesis filter and present the results in a CSV table.

positional arguments:
  static_filter_analysis
                        The static analysis JSON data produced by vc2-static-
                        filter-analysis.
  optimised_synthesis_test_patterns
                        A set of optimised synthesis test patterns produced by
                        vc2-optimise-synthesis-test-patterns.

optional arguments:
  -h, --help            show this help message and exit
  --picture-bit-width PICTURE_BIT_WIDTH, -b PICTURE_BIT_WIDTH
                        The number of bits in the picture signal.
  --custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_
↪MATRIX ...], -q CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]
                        Use a custom quantisation matrix. Optional except for
                        filters without a default quantisation matrix defined.
                        Should be specified as a series 3-argument tuples
                        giving the level, orientation and quantisation matrix
                        value for every entry in the quantisation matrix.
  --show-all-filter-phases, -p
                        Show signal bounds broken down into individual filter
```

(continues on next page)

(continued from previous page)

```
                        phases. Without this option, the bounds are shown for
                        all filter phases.
  --verbose, -v         Show more detailed status information during
                        execution.
  --output OUTPUT, -o OUTPUT
                        The name of the file to write the CSV bit widths table
                        to. Defaults to stdout.
```

4.3.3 CSV Format

The generated CSV have one row per analysis and synthesis filter phase. The columns are defined as follows:

type, level, array_name, x, y Specifies the filter phase represented by the row (see *Terminology* (page 6)). The 'x' and 'y' columns are omitted, and the aggregate worst-case shown for all phases unless `--show-all-filter-phases` is used.

lower_bound, upper_bound The theoretical worst-case lower- and upper-bounds for the signal. These values may be over-estimates but are guaranteed not to be under estimates of the true worst-case.

test_pattern_min, test_pattern_max The minimum and maximum values produced by the test patterns when they're passed through a real encoder and decoder. These values may not represent true worst-case signal levels, though they may often be close.

> **Warning:** It is possible that both minimum and maximum test pattern values could have the same sign. This occurs when quantisation errors are sufficiently strong as to make worst-case encodings/decodings of the test pattern have the same sign. The true signal range would still include zero.

bits The final column summarises the number of bits required for a signed two's complement representation of the values in the ranges indicated. If the test pattern and theory disagree, a range of bits is indicated. The true bit width required lies somewhere in the given range.

4.4 `vc2-bit-width-test-pictures`

Generate test pictures for VC-2 encoders and decoders which produce extreme (large-magnitude) signal values.

4.4.1 Example usage

In the example below we generate a series of test pictures for encoders and decoders which use/expect:

- 2-level 2D LeGall (5, 3) transform
- A particular custom quantisation matrix
- 10-bit pictures
- HD (1920x1080) resolution pictures with 4:2:2 colour subsampling

This tool generates test pictures for a single picture component at a time. Where picture components have different resolutions (as in this example) the `vc2-bit-width-test-pictures` command must be used twice.

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json

$ mkdir luma_test_pictures
```

(continues on next page)

(continued from previous page)

```
$ vc2-bit-width-test-pictures \
    static_analysis.json \
    1920 1080 \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3 \
    --output-directory luma_test_pictures

$ mkdir color_diff_test_pictures
$ vc2-bit-width-test-pictures \
    static_analysis.json \
    1920 540 \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3 \
    --output-directory color_diff_test_pictures

$ ls luma_test_pictures/
analysis_0.json         synthesis_1_qi35.json   synthesis_6_qi48.json
analysis_0.png          synthesis_1_qi35.png    synthesis_6_qi48.png
synthesis_0_qi34.json   synthesis_2_qi44.json   synthesis_7_qi49.json
synthesis_0_qi34.png    synthesis_2_qi44.png    synthesis_7_qi49.png
synthesis_10_qi52.json  synthesis_3_qi45.json   synthesis_8_qi50.json
synthesis_10_qi52.png   synthesis_3_qi45.png    synthesis_8_qi50.png
synthesis_11_qi54.json  synthesis_4_qi46.json   synthesis_9_qi51.json
synthesis_11_qi54.png   synthesis_4_qi46.png    synthesis_9_qi51.png
synthesis_12_qi58.json  synthesis_5_qi47.json
synthesis_12_qi58.png   synthesis_5_qi47.png
```

The two generated directories containing test pictures suitable for testing the luminance and colour-difference picture components.

4.4.2 Test picture format and usage

The test pictures generated by this software are saved as 8-bit monochrome PNG images, regardless of the picture bit width specified. These pictures must be extended to the correct picture bit width before use. The test pictures contain only black (minimum signal level), mid-gray (0) and white (maximum signal level) pixels. An example is shown below:

Analysis test pictures have names of the form `analysis_*.png`. These must be presented to an encoder under test and the output validated against a reference implementation. The results should be comparable with those achieved by the reference implementation.

Synthesis test pictures have names of the form `synthesis_*_qi*.png`. These must first be encoded using

an encoder configured to use the quantisation index given in the filename (after `qi`) for all picture slices. The encoded pictures must be presented to the decoder under test and compared with a reference implementation. The results should be bit-for-bit identical.

4.4.3 JSON metadata format

Each test picture (`*.png`) is accompanied by a JSON metadata file (`*.json`) with a matching name. This JSON file enumerates the test patterns contained within the picture and which values within an encoder or decoder implementation are being targeted.

Each file has the following structure:

```
[
    {
        "level": 1,
        "array_name": "L''",
        "x": 0,
        "y": 0,
        "maximise": true,
        "tx": 2,
        "ty": 4
    },
    ...
]
```

Each object describes a particular test pattern within the picture and its intended target within the encoder or decoder.

The `level`, `array_name`, `x` and `y` values define the specific filter phase targeted (see *Terminology* (page 6)).

The `maximise` value is `true` for patterns intended to maximise a signal value and `false` for those intended to minimise.

The `tx` and `ty` values give the coordinates of the specific value within the targeted filter array which is being targeted by the pattern.

4.4.4 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-bit-width-test-pictures [-h]
                                   [--picture-bit-width PICTURE_BIT_WIDTH]
                                   [--custom-quantisation-matrix CUSTOM_
↪QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]]
                                   [--verbose]
                                   [--output-directory OUTPUT_DIRECTORY]
                                   static_filter_analysis width height
                                   [optimised_synthesis_test_patterns]

Generate test pictures for VC-2 encoders and decoders which produce extreme
(large-magnitude) signal values.

positional arguments:
  static_filter_analysis
                        The static analysis JSON data produced by vc2-static-
                        filter-analysis.
  width                 The width (in samples) of the test pictures to
                        generate.
  height                The height (in samples) of the test pictures to
                        generate.
  optimised_synthesis_test_patterns
```

(continues on next page)

(continued from previous page)

```
                        A set of optimised synthesis test patterns produced by
                        vc2-optimise-synthesis-test-patterns.

optional arguments:
  -h, --help            show this help message and exit
  --picture-bit-width PICTURE_BIT_WIDTH, -b PICTURE_BIT_WIDTH
                        The number of bits in the picture signal.
  --custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_
↪MATRIX ...], -q CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]
                        Use a custom quantisation matrix. Optional except for
                        filters without a default quantisation matrix defined.
                        Should be specified as a series 3-argument tuples
                        giving the level, orientation and quantisation matrix
                        value for every entry in the quantisation matrix.
  --verbose, -v         Show more detailed status information during
                        execution.
  --output-directory OUTPUT_DIRECTORY, --output OUTPUT_DIRECTORY, -o OUTPUT_
↪DIRECTORY
                        The name of the directory to write the generated test
                        patterns and metadata files to. Must already exist.
                        Existing files will be overwritten. (Default: .).
```

4.5 vc2-maximum-quantisation-index

Compute the maximum quantisation index which is useful for a particular VC-2 codec configuration.

Using the same approach as the `vc2-bit-widths-table` command, this command works out the most extreme values which a VC-2 quantiser might encounter (given certain assumptions about the behaviour of the encoder design, see *Caveats* (page 5)). Using this information, it is possible to find the smallest quantisation index sufficient to quantise all transform coefficients to zero.

Though the VC-2 standard does not rule out larger quantisation indices being used, there is no reason for a sensible encoder implementation to use such a any larger quantisation index.

Note: The values printed by this tool are valid only for the wavelet transform, depth, picture bit width and quantisation matrix specified. See *Caveats* (page 5) for further limitations and assumptions made by this software.

4.5.1 Example usage

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json

$ vc2-maximum-quantisation-index \
    static_analysis.json \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3
59
```

4.5.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-maximum-quantisation-index [-h] --picture-bit-width
                                      PICTURE_BIT_WIDTH
                                      [--custom-quantisation-matrix CUSTOM_
↪QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]]
                                      [--verbose]
                                      static_filter_analysis

Compute the maximum quantisation index which is useful for a particular VC-2
codec configuration.

positional arguments:
  static_filter_analysis
                        The static analysis JSON data produced by vc2-static-
                        filter-analysis.

optional arguments:
  -h, --help            show this help message and exit
  --picture-bit-width PICTURE_BIT_WIDTH, -b PICTURE_BIT_WIDTH
                        The number of bits in the picture signal.
  --custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_
↪MATRIX ...], -q CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]
                        Define the custom quantisation matrix used by a codec.
                        Optional except for filters without a default
                        quantisation matrix defined. Should be specified as a
                        series 3-argument tuples giving the level, orientation
                        and quantisation matrix value for every entry in the
                        quantisation matrix.
  --verbose, -v         Show more detailed status information during
                        execution.
```

4.6 vc2-optimise-synthesis-test-patterns

Use a stochastic search to enchance the synthesis test patterns produced by vc2-static-filter-analysis so that they produce even more extreme values.

Note: The enhanced test patterns produced by this tool are valid only for the wavelet transform, depth, picture bit width and quantisation matrix specified. See *Caveats* (page 5) for further limitations and assumptions made by this software.

4.6.1 Example usage

```
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json

$ vc2-maximum-quantisation-index \
    static_analysis.json \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
```

(continues on next page)

(continued from previous page)

```
        2 HL 1   2 LH 3   2 HH 3 \
    --number-of-searches 10 \
    --added-corruption-rate 0.2 \
    --removed-corruption-rate 0.05 \
    --base-iterations 1000 \
    --added-iterations-per-improvement 500 \
    --output optimised_patterns.json
```

4.6.2 Arguments

The complete set of arguments can be listed using `--help`

```
usage: vc2-optimise-synthesis-test-patterns [-h] --picture-bit-width
                                            PICTURE_BIT_WIDTH
                                            [--custom-quantisation-matrix CUSTOM_
↪QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]]
                                            [--seed SEED]
                                            [--number-of-searches NUMBER_OF_
↪SEARCHES]
                                            [--terminate-early TERMINATE_EARLY]
                                            [--added-corruption-rate ADDED_
↪CORRUPTION_RATE]
                                            [--removed-corruption-rate REMOVED_
↪CORRUPTION_RATE]
                                            [--base-iterations BASE_ITERATIONS]
                                            [--added-iterations-per-improvement␣
↪ADDED_ITERATIONS_PER_IMPROVEMENT]
                                            [--output OUTPUT] [--verbose]
                                            static_filter_analysis

Use a stochastic search to enchance the synthesis test patterns produced by
vc2-static-filter-analysis so that they produce even more extreme values.

positional arguments:
  static_filter_analysis
                        The static analysis JSON data produced by vc2-static-
                        filter-analysis.

optional arguments:
  -h, --help            show this help message and exit
  --picture-bit-width PICTURE_BIT_WIDTH, -b PICTURE_BIT_WIDTH
                        The number of bits in the picture signal.
  --custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_
↪MATRIX ...], -q CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]
                        Use a custom quantisation matrix for the search.
                        Optional except for filters without a default
                        quantisation matrix defined. Should be specified as a
                        series 3-argument tuples giving the level, orientation
                        and quantisation matrix value for every entry in the
                        quantisation matrix.
  --seed SEED, -s SEED  The seed for the random number generator used to
                        perform searches.
  --number-of-searches NUMBER_OF_SEARCHES, -N NUMBER_OF_SEARCHES
                        The number of independent search runs to use. The
                        larger this option, the more likely it is that the
                        search will avoid local minima (default: 10).
  --terminate-early TERMINATE_EARLY, -t TERMINATE_EARLY
                        Terminate optimisation early if this many searches
                        fail to find an improvement. (Default: 1).
```

(continues on next page)

(continued from previous page)

```
  --added-corruption-rate ADDED_CORRUPTION_RATE, -a ADDED_CORRUPTION_RATE
                        The proportion of pixel values to corrupt in each
                        search iteration. (Default: 0.05).
  --removed-corruption-rate REMOVED_CORRUPTION_RATE, -r REMOVED_CORRUPTION_RATE
                        The proportion of pixel values to reset to their
                        original state in each search iteration. (Default:
                        0.0).
  --base-iterations BASE_ITERATIONS, -i BASE_ITERATIONS
                        The base number of trials to run the search for. The
                        larger this value, the longer the search will run
                        without finding any improvements before terminating.
                        (Default: 200).
  --added-iterations-per-improvement ADDED_ITERATIONS_PER_IMPROVEMENT, -I ADDED_
↪ITERATIONS_PER_IMPROVEMENT
                        The number of additional search iterations to perform
                        when an improved test pattern is found. (Default:
                        200).
  --output OUTPUT, -o OUTPUT
                        The name of the file to write the optimised test
                        pattern JSON file to. Defaults to stdout.
  --verbose, -v         Show more detailed status information during
                        execution.
```

4.6.3 Choosing parameters

The runtime and output quality of the optimisation algorithm is highly dependent on the paramters supplied. The best performing parameters for one wavelet transform may not be the best for others.

Choosing good parameters requires manual exploration and experimentation and is unfortunately out of the scope of this manual.

See *optimise_synthesis_test_patterns()* (page 38) for additional details.

4.6.4 JSON file format

The output is a JSON file with the following structure:

```
{
    "wavelet_index": <int>,
    "wavelet_index_ho": <int>,
    "dwt_depth": <int>,
    "dwt_depth_ho": <int>,
    "picture_bit_width": <int>,
    "quantisation_matrix": <quantisation-matrix>,
    "optimised_synthesis_test_patterns": [<test-pattern-specification>, ...],
}
```

Quantisation matrix

The `"quantisation_matrix"` field encodes the quantisation matrix used in the same fashion as the VC-2 pseudocode and its format should be self-explanatory. For example, the quantisation matrix passed to the example above is encoded as:

```
{
    0: {"LL": 1},
    1: {"HL": 2, "LH": 0, "HH": 4},
    2: {"HL": 1, "LH": 3, "HH": 3},
}
```

Note: The *deserialise_quantisation_matrix()* (page 45) Python utility function is provided for unpacking this structure.

Test patterns

The `<test-pattern-specification>` values follow the same format used by `vc2-static-filter-analysis` (see *Test patterns* (page 17)) with some additional fields:

```
<test-pattern-specification> = {
    ...,
    "quantisation_index": <int>,
    "decoded_value": <int>,
    "num_search_iterations": <int>,
}
```

These fields give the quantisation index found to produce the most extreme value for the test pattern, the value it managed to produce and the number of search iterations taken to reach that test pattern. This information is provided for informational purposes only.

Note: The *deserialise_test_pattern_specifications()* (page 44) Python utility function is provided for unpacking this structure.

Missing values

Only intermediate arrays are included which contain novel values. Arrays which are just renamings, interleavings and subsamplings of other arrays are omitted.

4.7 vc2-bundle

Create, or query a compressed bundle containing a set of static filter analyses (from *vc2-static-filter-analysis* (page 15)) and optimised synthesis test patterns (from *vc2-optimise-synthesis-test-patterns* (page 26)).

4.7.1 Example usage

Creating a bundle

```
$ # Create some files
$ vc2-static-filter-analysis \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2 \
    --output static_analysis.json
$ vc2-maximum-quantisation-index \
    static_analysis.json \
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3 \
    --output optimised_patterns.json

$ # Bundle the files together
$ vc2-bundle \
    create \
    bundle.zip \
    static_analysis.json \
    optimised_patterns.json \
    # ...
```

Listing bundle contents

```
$ vc2-bundle list bundle.zip
Static filter analyses
======================

0.
    * wavelet_index: le_gall_5_3
    * wavelet_index_ho: le_gall_5_3
    * dwt_depth: 2
    * dwt_depth_ho: 0

Optimised synthesis test patterns
=================================

0.
    * wavelet_index: le_gall_5_3
    * wavelet_index_ho: le_gall_5_3
    * dwt_depth: 2
    * dwt_depth_ho: 0
    * picture_bit_width: 10
    * custom_quantisation_matrix:
      {
          0: {'LL': 1},
          1: {'HL': 2, 'LH': 0, 'HH' 4},
          2: {'HL': 1, 'LH': 3, 'HH' 3},
      }
```

Extracting files from a bundle

```
$ # Extract a static analysis file from the bundle
$ vc2-bundle \
    extract-static-filter-analysis \
    bundle.zip \
    extracted.json \
    --wavelet-index le_gall_5_3 \
```

(continues on next page)

(continued from previous page)

```
    --dwt-depth 2

$ # Extract a set of optimised synthesis test patterns from the bundle
$ vc2-bundle \
    extract-optimised-synthesis-test-pattern \
    bundle.zip \
    extracted.json \
    --wavelet-index le_gall_5_3 \
    --dwt-depth 2
    --picture-bit-width 10 \
    --custom-quantisation-matrix \
        0 LL 1 \
        1 HL 2   1 LH 0   1 HH 4 \
        2 HL 1   2 LH 3   2 HH 3 \
```

4.7.2 Arguments

The complete set of arguments can be listed using `--help` for each subcommand.

```
usage: vc2-bundle [-h] [--verbose]
                  {create,list,extract-static-filter-analysis,extract-optimised-
↪synthesis-test-patterns}
                  ...

Create, or query a compressed bundle containing a set of static filter
analyses (from vc2-static-filter-analysis) and optimised synthesis test
patterns (from vc2-optimise-synthesis-test-patterns).

positional arguments:
  {create,list,extract-static-filter-analysis,extract-optimised-synthesis-test-
↪patterns}
    create              Create a bundle file.
    list                List the contents of a bundle file.
    extract-static-filter-analysis
                        Extract a static filter analysis from the bundle.
    extract-optimised-synthesis-test-patterns
                        Extract a set of optimised synthesis test patterns
                        from the bundle.

optional arguments:
  -h, --help            show this help message and exit
  --verbose, -v         Show more detailed status information during
                        execution.
```

```
usage: vc2-bundle create [-h]
                         [--static-filter-analysis JSON_FILE [JSON_FILE ...]]
                         [--optimised-synthesis-test-patterns JSON_FILE [JSON_FILE␣
↪...]]
                         bundle_file

positional arguments:
  bundle_file           Filename for the bundle file to create/overwrite.

optional arguments:
  -h, --help            show this help message and exit
  --static-filter-analysis JSON_FILE [JSON_FILE ...], -s JSON_FILE [JSON_FILE ...]
                        Filenames of the JSON files containing static filter
                        analyses produced by vc2-static-filter-analysis.
  --optimised-synthesis-test-patterns JSON_FILE [JSON_FILE ...], -o JSON_FILE␣
↪[JSON_FILE ...]
```

(continues on next page)

(continued from previous page)

```
                        Filenames of the JSON files containing optimised
                        synthesis test patterns produced by vc2-optimise-
                        synthesis-test-patterns.
```

```
usage: vc2-bundle list [-h] bundle_file

positional arguments:
  bundle_file  Filename for the bundle file to list.

optional arguments:
  -h, --help   show this help message and exit
```

```
usage: vc2-bundle extract-static-filter-analysis [-h] [--output OUTPUT]
                                                 --wavelet-index WAVELET_INDEX
                                                 [--wavelet-index-ho WAVELET_INDEX_
↪HO]
                                                 [--dwt-depth DWT_DEPTH]
                                                 [--dwt-depth-ho DWT_DEPTH_HO]
                                                 bundle_file

positional arguments:
  bundle_file           Filename for the bundle file to query.

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT, -o OUTPUT
                        Filename for the extracted JSON file. Defaults to
                        stdout.
  --wavelet-index WAVELET_INDEX, -w WAVELET_INDEX
                        The VC-2 wavelet index for the wavelet transform. One
                        of: 0 or deslauriers_dubuc_9_7, 1 or le_gall_5_3, 2 or
                        deslauriers_dubuc_13_7, 3 or haar_no_shift, 4 or
                        haar_with_shift, 5 or fidelity, 6 or daubechies_9_7.
  --wavelet-index-ho WAVELET_INDEX_HO, -W WAVELET_INDEX_HO
                        The VC-2 wavelet index for the horizontal parts of the
                        wavelet transform. If not specified, assumed to be the
                        same as --wavelet-index/-w.
  --dwt-depth DWT_DEPTH, -d DWT_DEPTH
                        The VC-2 transform depth. Defaults to 0 if not
                        specified.
  --dwt-depth-ho DWT_DEPTH_HO, -D DWT_DEPTH_HO
                        The VC-2 horizontal-only transform depth. Defaults to
                        0 if not specified.
```

```
usage: vc2-bundle extract-optimised-synthesis-test-patterns
       [-h] [--output OUTPUT] --wavelet-index WAVELET_INDEX
       [--wavelet-index-ho WAVELET_INDEX_HO] [--dwt-depth DWT_DEPTH]
       [--dwt-depth-ho DWT_DEPTH_HO] --picture-bit-width PICTURE_BIT_WIDTH
       [--custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_
↪QUANTISATION_MATRIX ...]]
       bundle_file

positional arguments:
  bundle_file           Filename for the bundle file to query.

optional arguments:
  -h, --help            show this help message and exit
  --output OUTPUT, -o OUTPUT
                        Filename for the extracted JSON file. Defaults to
                        stdout.
```

(continues on next page)

(continued from previous page)

```
  --wavelet-index WAVELET_INDEX, -w WAVELET_INDEX
                        The VC-2 wavelet index for the wavelet transform. One
                        of: 0 or deslauriers_dubuc_9_7, 1 or le_gall_5_3, 2 or
                        deslauriers_dubuc_13_7, 3 or haar_no_shift, 4 or
                        haar_with_shift, 5 or fidelity, 6 or daubechies_9_7.
  --wavelet-index-ho WAVELET_INDEX_HO, -W WAVELET_INDEX_HO
                        The VC-2 wavelet index for the horizontal parts of the
                        wavelet transform. If not specified, assumed to be the
                        same as --wavelet-index/-w.
  --dwt-depth DWT_DEPTH, -d DWT_DEPTH
                        The VC-2 transform depth. Defaults to 0 if not
                        specified.
  --dwt-depth-ho DWT_DEPTH_HO, -D DWT_DEPTH_HO
                        The VC-2 horizontal-only transform depth. Defaults to
                        0 if not specified.
  --picture-bit-width PICTURE_BIT_WIDTH, -b PICTURE_BIT_WIDTH
                        The number of bits in the picture signal.
  --custom-quantisation-matrix CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_
↪MATRIX ...], -q CUSTOM_QUANTISATION_MATRIX [CUSTOM_QUANTISATION_MATRIX ...]
                        Use a custom quantisation matrix for the search.
                        Optional except for filters without a default
                        quantisation matrix defined. Should be specified as a
                        series 3-argument tuples giving the level, orientation
                        and quantisation matrix value for every entry in the
                        quantisation matrix.
```

CHAPTER

FIVE

HIGH-LEVEL PYTHON API REFERENCE

The *vc2_bit_widths* (page 33) Python module exposes the underlying functionality provided by the command-line utilities (see *Command-line utilities reference* (page 15)) via a Python interface. In addition, Python functions are provided for assisting in the serialisation and deserialisation of the file formats used by the command-line utilities.

5.1 vc2_bit_widths.helpers: Helper Functions

The *vc2_bit_widths.helpers* (page 35) module provides a set of high-level utility functions implementing the key processes involved in bit-width analysis and test picture production for VC-2 codecs. These functions are used in much the same way as the command line interfaces, so refer to *Tutorial* (page 9) for a more general introduction.

Many of the functions in this module can take a significant amount of time to execute for very large filters. Status information is reported using Python's built-in `logging`[6] library and may be made visible using:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

5.1.1 Static filter analysis

static_filter_analysis(*wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*, *num_batches=1*, *batch_num=0*)

Performs a complete static analysis of a VC-2 filter configuration, computing theoretical upper- and lower-bounds for signal values (see *Computing signal bounds with Affine Arithmetic* (page 55)) and heuristic test patterns (see *Test pattern generation* (page 63)) for all intermediate and final analysis and synthesis filter values.

Parameters

wavelet_index [vc2_data_tables.WaveletFilters ([vc2_data_tables], page 7) or int]

wavelet_index_ho [vc2_data_tables.WaveletFilters ([vc2_data_tables], page 7) or int]

dwt_depth [int]

dwt_depth_ho [int] The filter parameters.

num_batches [int]

batch_num [int] Though for most filters this function runs either instantaneously or at worst in the space of a couple of hours, unusually large filters can take an extremely long time to run. For example, a 4-level Fidelity transform may take around a month to evaluate.

[6] https://docs.python.org/3/library/logging.html#module-logging

These arguments may be used to split this job into separate batches which may be computed separately (and in parallel) and later combined. For example, setting `num_batches` to 3 results in only analysing every third filter phase. The `batch_num` parameter should then be set to either 0, 1 or 2 to specify which third.

The skipped phases are simply omitted from the returned dictionaries. The dictionaries returned for each batch should be unified to produce the complete analysis.

Returns

analysis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}]

synthesis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}] Expressions defining the upper and lower bounds for all intermediate and final analysis and synthesis filter values.

The keys of the returned dictionaries give the level, array name and filter phase for which each pair of bounds corresponds (see *Terminology* (page 6)). The naming conventions used are those defined by *vc2_bit_widths.vc2_filters.analysis_transform()* (page 108) and *vc2_bit_widths.vc2_filters.synthesis_transform()* (page 108). Arrays which are just interleavings, subsamplings or renamings of other arrays are omitted.

The lower and upper bounds are given algebraically as *LinExp* (page 114)s.

For the analysis filter bounds, the expressions are defined in terms of the variables `LinExp("signal_min")` and `LinExp("signal_max")`. These should be substituted for the minimum and maximum picture signal level to find the upper and lower bounds for a particular picture bit width.

For the synthesis filter bounds, the expressions are defined in terms of variables of the form `LinExp("coeff_LEVEL_ORIENT_min")` and `LinExp("coeff_LEVEL_ORIENT_max")` which give lower and upper bounds for the transform coefficients with the named level and orientation.

The *evaluate_filter_bounds()* (page 36) function may be used to substitute concrete values into these expressions for a particular picture bit width.

analysis_test_patterns: {(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}

synthesis_test_patterns: {(level, array_name, x, y): *TestPatternSpecification* (page 41), ...} Heuristic test patterns which are designed to maximise a particular intermediate or final filter value. For a minimising test pattern, invert the polarities of the pixels.

The keys of the returned dictionaries give the level, array name and filter phase for which each set of bounds corresponds (see *Terminology* (page 6)). Arrays which are just interleavings, subsamplings or renamings of other arrays are omitted.

5.1.2 Calculating bit-width requirements

evaluate_filter_bounds(*wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*, *analysis_signal_bounds*, *synthesis_signal_bounds*, *picture_bit_width*)

Evaluate all analysis and synthesis filter signal bounds expressions for a given picture bit width, giving concrete signal ranges.

Parameters

wavelet_index [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

wavelet_index_ho [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

dwt_depth [int]

dwt_depth_ho [int] The filter parameters.

analysis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}]

synthesis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}] The outputs of *static_filter_analysis()* (page 35).

picture_bit_width [int] The number of bits in the input pictures.

Returns

concrete_analysis_signal_bounds [{(level, array_name, x, y): (lower_bound, upper_bound), ...}]

concrete_synthesis_signal_bounds [{(level, array_name, x, y): (lower_bound, upper_bound), ...}] The concrete, integer signal bounds for all analysis and signal filters given a `picture_bit_width`-bit input picture.

Includes values for *all* arrays and phases, even if array interleavings/subsamplings/renamings are omitted in the input arguments.

evaluate_test_pattern_outputs(*wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*, *picture_bit_width*, *quantisation_matrix*, *max_quantisation_index*, *analysis_test_patterns*, *synthesis_test_patterns*)

Given a set of test patterns, compute the signal levels actually produced by them when passed through a real encoder/decoder.

Parameters

wavelet_index [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

wavelet_index_ho [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

dwt_depth [int]

dwt_depth_ho [int] The filter parameters.

picture_bit_width [int] The number of bits in the input pictures.

quantisation_matrix [{level: {orient: value, ...}, ...}] The quantisation matrix.

max_quantisation_index [int] The maximum quantisation index to try (e.g. as computed by *quantisation_index_bound()* (page 38)). Each synthesis test pattern will be quantised with every quantisation index up to (and inclusing) this limit and the worst-case value for any quantisation index will be reported.

analysis_test_patterns [{(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}]

synthesis_test_patterns [{(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}] The test patterns to assess, e.g. from *static_filter_analysis()* (page 35) or *optimise_synthesis_test_patterns()* (page 38).

Returns

analysis_test_pattern_outputs [{(level, array_name, x, y): (lower_bound, upper_bound), ...}]

synthesis_test_pattern_outputs [{(level, array_name, x, y): ((lower_bound, qi), (upper_bound, qi)), ...}] The worst-case signal levels achieved for each of the provided test signals when using minimising and maximising versions of the test pattern respectively.

For the syntehsis test patterns, the quantisation index used to achieve the worst-case values is also reported.

Includes values for *all* arrays and phases, even if array interleavings/subsamplings/renamings are omitted in the input arguments.

5.1.3 Bounding quantisation indices

quantisation_index_bound(*concrete_analysis_signal_bounds*, *quantisation_matrix*)

Find the largest quantisation index which could usefully be used given the supplied analysis signal bounds.

Parameters

concrete_analysis_signal_bounds [{(level, orient, x, y): (lower_bound, upper_bound), ...}] Concrete analysis filter bounds as produced by, e.g., *evaluate_filter_bounds()* (page 36).

quantisation_matrix [{level: {orient: value, ...}, ...}] The quantisation matrix in use.

Returns

max_quantisation_index [int] The upper bound for the quantisation indices sensibly used by an encoder. This value will be the smallest quantisation index which will quantise all possible transform coefficients to zero.

5.1.4 Optimising synthesis test patterns

optimise_synthesis_test_patterns(*wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*, *quantisation_matrix*, *picture_bit_width*, *synthesis_test_patterns*, *max_quantisation_index*, *random_state*, *number_of_searches*, *terminate_early*, *added_corruption_rate*, *removed_corruption_rate*, *base_iterations*, *added_iterations_per_improvement*)

Perform a greedy search based optimisation of a complete set of synthesis test patterns.

See *Search algorithm & parameters* (page 93) for details of the optimisation process and parameters.

Parameters

wavelet_index [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

wavelet_index_ho [`vc2_data_tables.WaveletFilters` ([vc2_data_tables], page 7) or int]

dwt_depth [int]

dwt_depth_ho [int] The filter parameters.

quantisation_matrix [{level: {orient: value, ...}, ...}] The quantisation matrix in use.

picture_bit_width [int] The number of bits in the input pictures.

synthesis_test_patterns [{(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}] Synthesis test patterns to use as the starting point for optimisation, as produced by e.g. *static_filter_analysis()* (page 35).

max_quantisation_index [int] The maximum quantisation index to use, e.g. computed using *quantisation_index_bound()* (page 38).

random_state [`numpy.random.RandomState`[7]] The random number generator to use for the search.

number_of_searches [int] Repeat the greedy stochastic search process this many times for each test pattern. Since searches will tend to converge on local minima, increasing this parameter will tend to produce improved results.

terminate_early [None or int] If an integer, stop searching if the first `terminate_early` searches fail to find an improvement. If None, always performs all searches.

added_corruption_rate [float] The proportion of pixels to assign with a random value during each search attempt (0.0-1.0).

removed_corruption_rate [float] The proportion of pixels to reset to their starting value during each search attempt (0.0-1.0).

base_iterations [int] The initial number of search iterations to perform in each attempt.

added_iterations_per_improvement [int] The number of additional search iterations to perform whenever an improved picture is found.

Returns

optimised_test_patterns [{(level, array_name, x, y): *OptimisedTestPatternSpecification* (page 41), ...}] The optimised test patterns.

Note that arrays are omitted for arrays which are just interleavings of other arrays.

5.1.5 Generating test pictures

generate_test_pictures(*picture_width*, *picture_height*, *picture_bit_width*, *analysis_test_patterns*, *synthesis_test_patterns*, *synthesis_test_pattern_outputs*)

Generate a series of test pictures containing the supplied selection of test patterns.

Parameters

picture_width [int]

picture_height [int] The dimensions of the pictures to generate.

picture_bit_width [int] The number of bits in the input pictures.

analysis_test_patterns [{(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}]

synthesis_test_patterns [{(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}] The individual analysis and synthesis test patterns to be combined. A maximising and minimising variant of each pattern will be included in the output. See *static_filter_analysis()* (page 35) and *optimise_synthesis_test_patterns()* (page 38).

synthesis_test_pattern_outputs [{(level, array_name, x, y): ((lower_bound, qi), (upper_bound, qi)), ...}] The worst-case quantisation indicies for each synthesis test pattern, as computed by *evaluate_test_pattern_outputs()* (page 37).

Returns

analysis_pictures [[*AnalysisPicture* (page 40), ...]]

synthesis_pictures [[*SynthesisPicture* (page 40), ...]] A series of test pictures containing correctly aligned instances of each supplied test pattern.

Pictures are returned with values in the range 0 to (2**picture_bit_width)-1, as expected by a VC-2 encoder.

Each analysis picture includes a subset of the test patterns supplied (see *AnalysisPicture* (page 40)). The analysis test pictures are intended to be passed to an analysis filter as-is.

Each synthesis test picture includes a subset of the test patterns supplied, grouped according to the quantisation index to be used. The synthesis test pictures should first

[7] https://numpy.org/doc/stable/reference/random/legacy.html#numpy.random.RandomState

be passed through a synthesis filter and then the transform coefficients quantised using the quantisation index specified (see *SynthesisPicture* (page 40)). The quantised transform coefficients should then be passed through the synthesis filter.

class AnalysisPicture(*picture*, *test_points*)
namedtuple()[8]. A test picture which is to be used to assess an analysis filter.

Parameters

picture [numpy.array] The test picture.

test_points [[*TestPoint* (page 40), ...]] A list of locations within the analysis filter being tested by this picture.

class SynthesisPicture(*picture*, *quantisation_index*, *test_points*)
namedtuple()[9]. A test picture which is to be used to assess an synthesis filter.

Parameters

picture [numpy.array] The test picture.

quantisation_index [int] The quantisation index to use for all picture slices when encoding the test picture.

test_points [[*TestPoint* (page 40), ...]] A list of locations within the synthesis filter being tested by this picture.

class TestPoint(*level*, *array_name*, *x*, *y*, *maximise*, *tx*, *ty*)
namedtuple()[10]. Definition of the location of a point in an encoder/decoder tested by a test picture.

Parameters

level [int]

array_name [str]

x [int]

y [int] The encoder/decoder intermediate value array and phase tested by this test point.

maximise [bool] If True, at this point the signal level is being maximised, if False it will be minimised.

tx [int]

ty [int] The coordinates (in the intermediate value array (level, array_name)) which are being maximised or minimised.

5.2 vc2_bit_widths.patterns: Containers for test pattern data

The following datastructures are used to define test patterns.

[8] https://docs.python.org/3/library/collections.html#collections.namedtuple
[9] https://docs.python.org/3/library/collections.html#collections.namedtuple
[10] https://docs.python.org/3/library/collections.html#collections.namedtuple

5.2.1 Test pattern specification

These types define a test pattern along with information about required spacing or quantisation levels.

class TestPatternSpecification(*target*, *pattern*, *pattern_translation_multiple*, *target_translation_multiple*)

A definition of a test pattern for a VC-2 filter. This test pattern is intended to maximise the value of a particular intermediate or output value of a VC-2 filter.

Test patterns for both for analysis and synthesis filters are defined in terms of picture test patterns. For analysis filters, the picture should be fed directly to the encoder under test. For synthesis filters, the pattern must first be fed to an encoder and the transform coefficients quantised before being fed to a decoder.

Test patterns tend to be quite small (tens to low hundreds of pixels square) and so it is usually sensible to collect together many test patterns into a single picture (see *vc2_bit_widths.picture_packing* (page 83)). To retain their functionality, test patterns must remain correctly aligned with their target filter. When relocating a test pattern, the pattern must be moved only by multiples of the values in `pattern_translation_multiple`. For each multiple moved, the target value effected by the pattern moves by the same multiple of `target_translation_multiple`.

Parameters

target [(tx, ty)] The target coordinate which is maximised by this test pattern.

pattern [*TestPattern* (page 42)] The input pattern to be fed into a VC-2 encoder. Only those pixels defined in this pattern need be set – all other pixels may be set to arbitrary values and have no effect.

The test pattern is specified such that the pattern is as close to the top-left corner as possible given `pattern_translation_multiple`, without any negative pixel coordinates. That is, `0 <= min(x) < mx` and `0 <= min(y) < my`.

pattern_translation_multiple [(mx, my)]

target_translation_multiple [(tmx, tmy)] The multiples by which pattern pixel coordinates and target array coordinates may be translated when relocating the test pattern. Both the pattern and target must be translated by the same multiple of these two factors.

For example, if the pattern is translated by (2*mx, 3*my), the target must be translated by (2*tmx, 3*tmy).

class OptimisedTestPatternSpecification(*target*, *pattern*, *pattern_translation_multiple*, *target_translation_multiple*, *quantisation_index*, *decoded_value*, *num_search_iterations*)

A test pattern specification which has been optimised to produce more extreme signal values for a particular codec configuration.

Parameters

target [(tx, ty)]

pattern [*TestPattern* (page 42)]

pattern_translation_multiple [(mx, my)]

target_translation_multiple [(tmx, tmy)] Same as *TestPatternSpecification* (page 41)

quantisation_index [int] The quantisation index which, when used for all coded picture slices, produces the largest values when this pattern is decoded.

decoded_value [int] For informational purposes. The value which will be produced in the target decoder array for this input pattern.

num_search_iterations [int] For informational purposes. The number of search iterations performed to find this value.

invert_test_pattern_specification(*test_pattern*)
Given a *TestPatternSpecification* (page 41) or *OptimisedTestPatternSpecification* (page 41), return a copy with the signal polarity inverted.

5.2.2 Test pattern

A description of a test pattern in terms of pixel polarities.

class TestPattern(**args*)
A test pattern.

A test pattern is described by a set of pixels with a defined polarity (either -1 or +1) with all other pixels being undefined (represented as 0). Where a pixel has +ve polarity, the maximum signal value should be used in test pictures, for -ve polarities, the minimum signal value should be used.

This object stores test patterns as a `numpy.array` defining a rectangular region where some pixels are not undefined. This region lies at the offset defined by *origin* (page 42) and the values within the region are defined by *polarities* (page 42).

TestPattern (page 42) objects can be constructed from either a dictionary of the form `{(x, y): polarity, ...}` or from a pair of arguments, `origin` and `polarities` giving a `(dy, dx)` tuple and 2D `np.array` respectively.

property origin
The origin of the rectangular region definining the test pattern given as (dy, dx).

> **Warning:** Numpy-style index ordering used!

property polarities
A `numpy.array` definining the polarities of the pixels within a rectangular region definde by *origin* (page 42).

as_dict()
Return a dictionary representation of the test pattern of the form `{(x, y): polarity, ...}`.

as_picture_and_slice(*signal_min=-1*, *signal_max=1*, *dtype=<class 'numpy.int64'>*)
Convert this test pattern into a picture array with its origin at (0, 0).

Not supported for test patterns with pixels at negative coordinates.

Parameters

signal_min [int]

signal_max [int] The values to use for pixels with negative and positive polarities, respecitvely.

dtype The `numpy.array` datatype for the returned array.

Returns

picture [`numpy.array`] A test picture with its origin at (0, 0) containing the test pattern (with zeros in all undefined pixels).

picture_slice [(`slice`[11], `slice`[12])] A `numpy.array` slice identifying the region of `picture` which contains the test pattern.

__len__()
The number of defined pixels in the test pattern.

__neg__()
Return a *TestPattern* (page 42) with inverted polarities.

[11] https://docs.python.org/3/library/functions.html#slice
[12] https://docs.python.org/3/library/functions.html#slice

5.3 vc2_bit_widths.json_serialisations: JSON Data File Serialisation/Deserialisation

The *vc2_bit_widths.json_serialisations* (page 42) module provides functions for serialising and deserialising parts of the JSON files produced by the various command-line interfaces to *vc2_bit_widths* (page 33) (see *Command-line utilities reference* (page 15)).

Note: These functions do not serialise/deserialise JSON directly, instead they produce/accept standard Python data structures compatible with the Python `json`[13] module.

5.3.1 Signal bounds

serialise_signal_bounds(*signal_bounds*)

Convert a dictionary of analysis or synthesis signal bounds expressions into a JSON-serialisable form.

See *serialise_linexp()* (page 43) for details of the `"lower_bound"` and `"upper_bound"` fields.

For example::

```
>>> before = {
...     (1, "LH", 2, 3): (
...         LinExp("foo")/2 + 1,
...         LinExp("bar")/4 + 2,
...     ),
...     ...
... }
>>> serialise_signal_bounds(before)
[
    {
        "level": 1,
        "array_name": "LH",
        "phase": [2, 3],
        "lower_bound": [
            {"symbol": "foo", "numer": "1", "denom": "2"},
            {"symbol": None, "numer": "1", "denom": "1"},
        ],
        "upper_bound": [
            {"symbol": "bar", "numer": "1", "denom": "4"},
            {"symbol": None, "numer": "2", "denom": "1"},
        ],
    },
    ...
]
```

deserialise_signal_bounds(*signal_bounds*)

Inverse of *serialise_signal_bounds()* (page 43).

serialise_linexp(*exp*)

Serialise a restricted subset of *LinExp* (page 114) s into JSON form.

Restrictions:

- Only supports expressions made up of rational values (i.e. no floating point coefficients/constants.
- Symbols must all be strings (i.e. no tuples or *AAError* (page 114) sybols)

Example::

[13] https://docs.python.org/3/library/json.html#module-json

```
>>> before = LinExp({
...     "a": Fraction(1, 5),
...     "b": Fraction(10, 1),
...     None: Fraction(-2, 3),
... })
>>> serialise_linexp(before)
[
    {"symbol": "a", "numer": "1", "denom": "5"},
    {"symbol": "b", "numer": "10", "denom": "1"},
    {"symbol": None, "numer": "-2", "denom": "3"},
]
```

Note: Numbers are encoded as strings to avoid floating point precision limitations.

deserialise_linexp(*json*)
Inverse of *serialise_linexp()* (page 43).

5.3.2 Test patterns

serialise_test_pattern_specifications(*spec_namedtuple_type*, *test_patterns*)
Convert a dictionary of analysis or synthesis test pattern specifications into a JSON-serialisable form.

See *serialise_test_pattern()* (page 45) for the serialisation used for the pattern data.

For example:

```
>>> before = {
...     (1, "LH", 2, 3): TestPatternSpecification(
...         target=(4, 5),
...         pattern=TestPattern({(x, y): polarity, ...}),
...         pattern_translation_multiple=(6, 7),
...         target_translation_multiple=(8, 9),
...     ),
...     ...
... }
>>> serialise_test_pattern_specifications(TestPatternSpecification, before)
[
    {
        "level": 1,
        "array_name": "LH",
        "phase": [2, 3],
        "target": [4, 5],
        "pattern": {...},
        "pattern_translation_multiple": [6, 7],
        "target_translation_multiple": [8, 9],
    },
    ...
]
```

Parameters

spec_namedtuple_type [namedtuple()[14] class] The namedtuple used to hold the test pattern specification. One of *TestPatternSpecification* (page 41) or *OptimisedTestPatternSpecification* (page 41).

test_patterns [{(level, array_name, x, y): (...), ...}]

[14] https://docs.python.org/3/library/collections.html#collections.namedtuple

deserialise_test_pattern_specifications(*spec_namedtuple_type*, *test_patterns*)
Inverse of *serialise_test_pattern_specifications()* (page 44).

serialise_test_pattern(*test_pattern*)
Convert a `TestPattern` into a compact JSON-serialisable form.

For example::

```
>>> before = TestPattern({
...     (4, 10): +1, (5, 10): -1, (6, 10): +1, (7, 10): -1,
...     (4, 11): -1, (5, 11): +1, (6, 11): -1, (7, 11): +1,
...     (4, 12): +1, (5, 12): -1, (6, 12): +1, (7, 12): -1,
...     (4, 13): -1, (5, 13): +1, (6, 13): -1, (7, 13): +1,
... })
>>> serialise_test_pattern(before)
{
    'dx': 4,
    'dy': 10,
    'width': 4,
    'height': 4,
    'positive': 'paU=',
    'mask': '//8=',
}
```

The serialised format is based on a Base64 (RFC 3548) encoded bit-packed positive/mask representation of the picture.

In a positive/mask representation, the input picture is represented as two binary arrays: 'positive' and 'mask'. When an entry in the mask is 1, a 1 in the corresponding positive array means '+1' and a 0 means '-1'. When the mask entry is 0, the corresponding entry in the positive array is ignored and a value of '0' is assumed.

The positive and mask arrays are defined as having their bottom left corner at (dx, dy) and having the width and height defined. All values outside of this region are zero.

The positive and mask arrays are serialised in raster-scan order into a bit array. This bit array is packed into bytes with the first bit in each byte being the most significant.

The byte-packed bit arrays are finally Base64 (RFC 3548) encoded into the 'positive' and 'mask' fields.

deserialise_test_pattern(*dictionary*)
Inverse of `serialise_namedtuple()`.

5.3.3 Quantisation matrices

serialise_quantisation_matrix(*quantisation_matrix*)
Convert a quantisation matrix into JSON-serialisable form.

For example::

```
>>> before = {0: {"L": 2}, 1: {"H": 0}}
>>> serialise_quantisation_matrix(before)
{"0": {"L": 2}, "1": {"H": 0}}
```

deserialise_quantisation_matrix(*quantisation_matrix*)
Inverse of *serialise_quantisation_matrix()* (page 45).

5.4 `vc2_bit_widths.bundle`: Analysis file bundling

Pre-computed filter analyses (e.g. from multiple runs of *vc2-static-filter-analysis* (page 15)) and optimised test patterns (e.g. from runs of *vc2-optimise-synthesis-test-patterns* (page 26)) may be packed into a 'bundle' file along with an index of available patterns.

The bundle file is a compressed zip file and offers significant space savings over storing the analyses uncompressed.

The provision of an index file also enables more efficient lookup of files by codec parameters than reading every file.

bundle_create_from_serialised_dicts(*file_or_filename*, *serialised_static_filter_analyses=()*, *serialised_optimised_synthesis_test_patterns=()*)

Create a bundle file containing the supplied pre-serialised filter analyses and test patterns.

Parameters

file_or_filename [file-like or str] The bundle to write to.

serialised_static_filter_analyses [iterable] An iterable of dictionaries containing filter analyses with all entries serialised as produced by *vc2-static-filter-analysis* (page 15). Must be free of duplicates.

serialised_optimised_synthesis_test_patterns [iterable] An iterable of dictionaries containing filter analyses with all entries serialised as expected by *vc2-optimise-synthesis-test-patterns* (page 26). Must be free of duplicates.

bundle_index(*file_or_filename*)

Get the index from a bundle file.

Parameters

file_or_filename [file-like or str] The bundle to read.

Returns

index [dict] Returns a dictionary containing two lists under the keys "static_filter_analyses" and "optimised_synthesis_test_patterns". Both lists contain dictionaries containing the following fields:

- "wavelet_index" (int)
- "wavelet_index_ho" (int)
- "dwt_depth" (int)
- "dwt_depth_ho" (int)
- "filename": The file in the bundle containing the corresponding analysis or test patterns.

The "optimised_synthesis_test_patterns" list dicts also contain the following additional entries:

- "quantisation_matrix" (`{level: {orient: value, ...}, ...}`)
- "picture_bit_width" (int)

bundle_get_static_filter_analysis(*file_or_filename*, *wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*)

Read a static filter analysis for a particular filter configuration from a bundle. Raises `KeyError`[15] if no matching analysis is present in the bundle.

Parameters

file_or_filename [file-like or str] The bundle to read.

wavelet_index [int]

wavelet_index_ho [int]

dwt_depth [int]

dwt_depth_ho [int]

Returns

analysis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}]

synthesis_signal_bounds [{(level, array_name, x, y): (lower_bound_exp, upper_bound_exp), ...}]

analysis_test_patterns: {(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}

synthesis_test_patterns: {(level, array_name, x, y): *TestPatternSpecification* (page 41), ...}
See *vc2_bit_widths.helpers.static_filter_analysis()* (page 35).

bundle_get_optimised_synthesis_test_patterns(*file_or_filename*, *wavelet_index*, *wavelet_index_ho*, *dwt_depth*, *dwt_depth_ho*, *quantisation_matrix*, *picture_bit_width*)

Read a static filter analysis for a particular filter configuration from a bundle. Raises KeyError[16] if no matching analysis is present in the bundle.

Parameters

file_or_filename [file-like or str] The bundle to read.

wavelet_index [int]

wavelet_index_ho [int]

dwt_depth [int]

dwt_depth_ho [int]

quantisation_matrix [{level: {orient: value, ...}, ...}]

picture_bit_width [int]

Returns

optimised_test_patterns [{(level, array_name, x, y): *OptimisedTestPatternSpecification* (page 41), ...}] See *vc2_bit_widths.helpers.optimise_synthesis_test_patterns()* (page 38).

[15] https://docs.python.org/3/library/exceptions.html#KeyError
[16] https://docs.python.org/3/library/exceptions.html#KeyError

Part II

Theory and Design

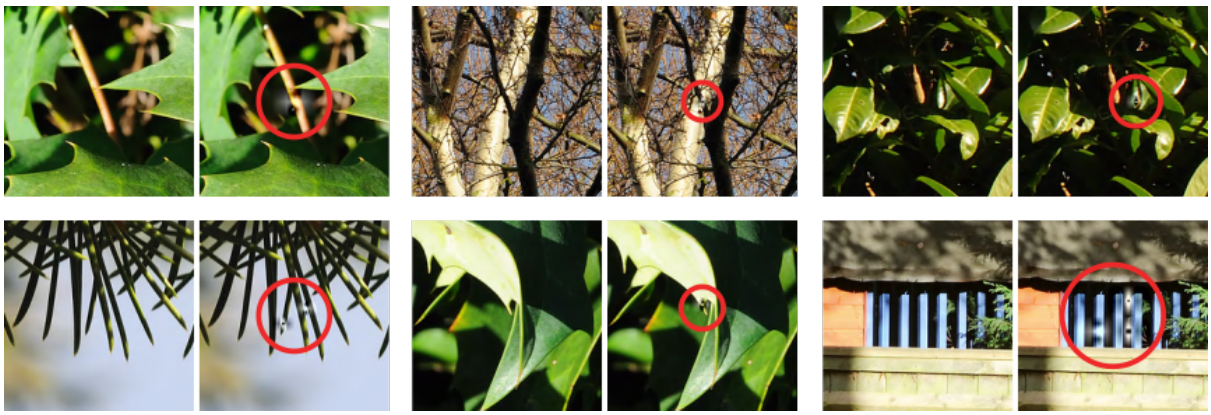
CHAPTER

SIX

THEORY OVERVIEW

This part of the documentation describes the underlying theory and techniques used by this software, some of which are original.

The objective of this software is to ensure implementations of VC-2 use a sufficient number of bits in their arithmetic to avoid integer wrap-around or saturation errors. The examples below show the effects of insufficient bit depth in a real VC-2 codec:



Though the Discrete Wavelet Transform (DWT) underlying VC-2 is linear, the integer arithmetic and quantisation make VC-2 a non-linear filter. Computing the worst-case signal ranges for non-linear filters is not practical in general. Instead, as discussed in *Related work* (page 53), implementers typically resort to either wasteful over-allocation of bits or error-prone empirical testing to determine the number of bits required.

This software uses Affine Arithmetic (AA), as described in *Computing signal bounds with Affine Arithmetic* (page 55), to find hard upper-bounds on signal ranges. While AA-based bounds are guaranteed not to under-estimate the true signal range, they typically over-estimate the true signal range.

The magnitude of the over-estimates produced by AA is strongly related to the magnitude of the non-linearity in the filter. In an analysis filter, for example, the only non-linearities arise from rounding errors which tend to be small, making the AA over-estimate small as well. Quantisation, however, can introduce substantial errors and so, post-quantisation, AA tends to produce fairly large over-estimates. This effect is quantified in *Quantisation and affine arithmetic* (page 57).

In *Refining worst-case quantisation error bounds* (page 57), a refinement to the way the worst-case bounds of VC-2's quantiser are calculated is presented, reducing the over-estimates produced by AA.

Since AA produces large over-estimates, this software also attempts to generate synthetic test patterns which elicit realistic, near-worst-case signal values. These test signals are generated by an original heuristic described in *Test pattern generation* (page 63). These test patterns have been found to produce significantly larger signal values than natural images or random noise in practice in experiments described in *Experimental results* (page 65).

CHAPTER

SEVEN

RELATED WORK

There are numerous examples of implementations of integer wavelet transforms in the literature, however published attempts at selecting bit-widths for arithmetic operations have been quite limited.

In [BaFT11] and [FaGa08], the authors take the potentially wasteful approach of adding an extra bit after every addition and summing the bits after a multiplication. In both cases, however, the authors fail to account for accumulated signal growth in multi-level transforms.

The other common approach used by, for example, [SZAA02], [BRGT06] and [HTLS08], is to pass a series of test pictures through a codec and observe the largest signal values encountered. This approach is error prone since the difference between 'worst case' signals and typical natural images can be great, as shown later.

In general, the problem of determining how many bits are required for a given integer arithmetic problem is NP-complete. For small problems involving a few tens of variables, SAT-Modulo-Theory (SMT) solvers may be used to find exact answers. For example, SMT solvers have been used to automatically identify integer-overflow bugs in C and C++ code [MoBj10].

Video filters unfortunately represent considerably larger problems than SMT solvers can tackle effectively, requiring hundreds or thousands of variables.

Kinsman and Nicolici [KiNi10] demonstrated an approach to applying SMT solvers to larger problems by terminating the SMT solver's search process after a fixed timeout. The incomplete state of the SMT solver is then used to give an (guaranteed over-estimated) upper-bound on signal values. Unfortunately, the performance of this approach is dependent on the timeouts chosen and the behaviour of the SMT solver during a given run. Further, the approach was only demonstrated for small numbers of variables (low tens) and it is unclear that it would scale well for considerably larger problems.

In the wider fields of digital signal processing and numerical methods, Interval Arithmetic (IA) and Affine Arithmetic (AA) are widely used techniques for bounding errors in linear functions in the presence of rounding and quantisation. López and Carreras give a good introduction and discussion of both techniques in [LoCN07].

Like Kinsman and Nicolici's truncated SMT approach, both IA and AA give hard upper bounds on signal levels. Unfortunately both are prone to over-estimating these bounds, with AA giving the tighter bounds. Nevertheless, these approaches are deterministic and easy to compute.

CHAPTER

EIGHT

COMPUTING SIGNAL BOUNDS WITH AFFINE ARITHMETIC

Though VC-2 implements the discrete wavelet transform, a linear filter, integer rounding and quantisation make VC-2 a non-linear filter. In this section we describe the process by which Affine Arithmetic (AA) may be used to find upper-bounds for signal ranges in VC-2.

8.1 Analysing linear filters

Given an algebraic description of a linear filter, it is straight-forward to determine the inputs which produce the most extreme output values.

For example, consider the following algebraic description which could describe how a linear filter might compute the value of a particular output, given two input pixel values a and b:

$$\frac{a}{2} - \frac{b}{8} + 1$$

In this expression a is weighted with a positive coefficient ($\frac{1}{2}$) while b is weighted with a negative coefficient ($-\frac{1}{8}$). As a consequence, to produce an output with the highest possible value we should set a to a large positive value and b to a large negative value. Conversely, the opposite is true if we wish to produce the lowest possible value.

For example, if we define the input signal range as being $[-100, 100]$, it is maximum result, 63.5, is produced when $a = 100$ and $b = -100$ and the minimum result, -61.5, when $a = -100$ and $b = 100$.

In this way, given an algebraic description of a linear filter, we can compute a set of worst-case input values (i.e. a test pattern) and also the output signal range.

8.2 Affine arithmetic

Affine arithmetic[17] provides a way to bound the effects of non-linearities due to rounding errors.

In affine arithmetic, non-linear operations are modelled as linear operations with error terms.

For example, when a value is divided by two using truncating integer arithmetic ($//$), this is modelled with affine arithmetic as:

$$x//2 = \frac{x}{2} + \frac{e_1 - 1}{2}$$

Where e_1 is an error term representing some value in the interval $[-1, +1]$.

As a result of the use of error terms, affine arithmetic expressions effectively specify *ranges* of possible values. In the example above, that range would be $\left[\frac{x}{2} - 1, \frac{x}{2}\right]$.

Every time a non-linear operation is modelled using affine arithmetic a new error term must be introduced, thereby (pessimistically) modelling all errors as being independent. In practice, rounding errors are not independent and so affine arithmetic will tend to indicate an overly broad range of values.

[17] https://en.wikipedia.org/wiki/Affine_arithmetic

For example, if we substitute $x = 11$ into $x//2$, affine arithmetic tells us that the answer lies in the range $[4.5, 5.5]$ which is true ($11//2 = 5$), but imprecise.

Nevertheless, affine arithmetic's pessimism guarantees that the true result is *always* contained in the range indicated.

As a rule of thumb, so long as the rounding errors in an expression are small, so too is the range indicated by affine arithmetic.

8.3 Worked example

The example below demonstrates the procedure used to find the theoretical bounds of a filter.

Consider again the following filter on an input in the range $[-100, 100]$:

$$\frac{a}{2} - \frac{b}{8} + 1$$

As before, we can use simple linear filter analysis to determine that the filter value is maximised when $a = 100$ and $b = -100$ and minimised when $a = -100$ and $b = 100$.

Lets assume that the filter is approximated using truncating integer arithmetic as:

$$(a + 1)//2 - (b + 4)//8 + 1$$

Represented using affine arithmetic we have:

$$\frac{a+1}{2} + \frac{e_1 - 1}{2} - \left(\frac{b+4}{8} + \frac{e_2 - 1}{2}\right) + 1$$

Substituting the minimising and maximising values for a and b we find that the filter's minimum and maximum values lies in the following ranges:

$$\begin{aligned}
\text{maximum value} &= \frac{100+1}{2} + \frac{e_1 - 1}{2} - \left(\frac{-100+4}{8} + \frac{e_2 - 1}{2}\right) + 1 \\
&= 50.5 + \frac{e_1 - 1}{2} - \left(-12 + \frac{e_2 - 1}{2}\right) + 1 \\
&= 63.5 + \frac{e_1 - 1}{2} - \frac{e_2 - 1}{2} \\
&= [62.5, 64.5]
\end{aligned}$$

$$\begin{aligned}
\text{minimum value} &= \frac{-100+1}{2} + \frac{e_1 - 1}{2} - \left(\frac{100+4}{8} + \frac{e_2 - 1}{2}\right) + 1 \\
&= -49.5 + \frac{e_1 - 1}{2} - \left(13 + \frac{e_2 - 1}{2}\right) + 1 \\
&= -61.5 + \frac{e_1 - 1}{2} - \frac{e_2 - 1}{2} \\
&= [-62.5, -60.5]
\end{aligned}$$

From this we can therefore say that the output of our integer approximation of the filter is bounded by the range $[-62.5, 64.5]$.

8.4 Quantisation and affine arithmetic

Affine arithmetic may also be used to model the effects of quantisation since VC-2's dead-zone quantiser is essentially just truncating integer division.

Consider the following (simplified) definition of VC-2's quantiser and dequantiser:

$$\text{quantise}(x, qf) = x//qf$$

$$\text{dequantise}(X, qf) = X \times qf + \frac{qf}{2}$$

In affine arithmetic this becomes:

$$\text{quantise}(x, qf) = \frac{x}{qf} + \frac{e_1 - 1}{2}$$

$$\text{dequantise}(X, qf) = X \times qf + \frac{qf}{2} + \frac{e_2 - 1}{2}$$

So the effect of quantising and dequantising a value, as modelled by affine arithmetic is:

$$\begin{aligned}\text{dequantise}(\text{quantise}(x, qf), qf) &= \left(\frac{x}{qf} + \frac{e_1 - 1}{2}\right) \times qf + \frac{qf}{2} + \frac{e_2 - 1}{2} \\ &= x + qf\frac{e_1 - 1}{2} + \frac{qf}{2} + \frac{e_2 - 1}{2} \\ &= \left[x - \frac{qf}{2} - 1, x + \frac{qf}{2}\right]\end{aligned}$$

This tells us that for large quantisation factors (where $qf \approx x$), quantisation produces a range:

$$\text{dequantise}(\text{quantise}(x, x), x) = \left[\frac{x}{2} - 1, \frac{3x}{2}\right]$$

When negative numbers are taken into account, the affine range becomes:

$$\text{dequantise}(\text{quantise}(x, x), x) = \left[-\frac{3}{2}x, \frac{3}{2}x\right]$$

That is, worst-case quantisation has the effect of replacing the quantised value with an affine error variable with a magnitude $\frac{3}{2}\times$ the quantised value. In the next section we'll attempt to bound this range.

8.5 Refining worst-case quantisation error bounds

Though in some cases quantisation can result in the quantised value growing by a factor of $\frac{3}{2}$, as predicted by affine arithmetic, for many values the worst-case gain is lower.

For any value in the range $[-x_{\text{max}}, x_{\text{max}}]$, the largest value possible after quantisation and dequantisation is found by quantising and dequantising x_{max} by the largest quantisation index which doesn't quantise the value to zero. Once known, this figure may be used to define a slightly smaller affine range to represent the target value.

A formal proof of this statement is provided in the remainder of this section.

Quantisation using VC-2's quantisation/dequantisation process can fairly simply be shown to produce outputs up to $\frac{3}{2}\times$ larger than the original value. This proof sets out to show the exact upper bound of the output of VC-2's quantiser given a defined input range.

8.5.1 Definitions

VC-2 defines a dead-zone quantiser based on an integer approximation of the following:

$$\text{quantise}(x,\, qf) = \begin{cases} \left\lfloor \frac{x}{qf} \right\rfloor & \text{if } x \geq 0 \\ -\left\lfloor \frac{-x}{qf} \right\rfloor & \text{if } x < 0 \end{cases}$$

$$\text{dequantise}(x,\, qf) = \begin{cases} x\,qf + \frac{qf}{2} & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ x\,qf - \frac{qf}{2} & \text{if } x < 0 \end{cases}$$

Where $x \in \mathbb{Z}$ and qf is the quantisation factor, defined in terms of the quantisation index, qi, as follows:

$$qf = 2^{qi/4} \quad \text{for} \quad qi \in \mathbb{N}$$

8.5.2 Formal problem statement

Given some value, x_{max}, what is the largest-magnitude value which may be produced by $\text{dequantise}(\text{quantise}(x,\, qf),\, qf)$ for $-x_{\text{max}} \leq x \leq x_{\text{max}}$ and any qf?

8.5.3 Lemmas

In these lemmas the following shorthand notation will be used:

$$x'_{qf} = \text{dequantise}(\text{quantise}(x,\, qf),\, qf)$$

Without loss of generality, only non-negative values are considered below. Analogous lemmas may be found trivially for the negative cases due to the symmetry of the quantisation and dequantisation processes.

Lemma 1: When $qf > x$, $x'_{qf} = 0$.

Proof: When $qf > x$, $\text{quantise}(x,\, qf) = \left\lfloor \frac{x}{qf} \right\rfloor$ is trivially equal to zero. Since $\text{dequantise}(0,\, qf) = 0$ by definition, $x'_{qf} = 0$ for all cases where $qf > x$. QED.

Lemma 2: If $x_1 \leq x_2$ then ${x_1}'_{qf} \leq {x_2}'_{qf}$.

That is, for a fixed qf, VC-2's quantiser is monotonic: changing the input value in one direction will never produce a change the output value in the opposite direction.

Proof: For the case where $x > 0$, the complete quantisation and dequantisation process combine to form:

$$x'_{qf} = \left\lfloor \frac{x}{qf} \right\rfloor qf + \frac{qf}{2}$$

This expression is composed of only linear parts with the exception of the floor operator which, nevertheless, is monotonic. As a consequence, the operation as a whole is monotonic with a lower bound of 0.

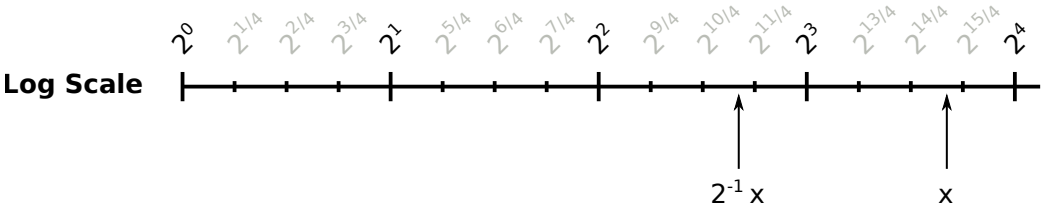
In the case where $x = 0$, $x'_{qf} = 0$.

QED.

Lemma 3: For $x > 0$, there exists at least one valid quantisation factor in the range $\frac{x}{2} < qf \leq x$.

Proof: The quantisation factor is restricted to quarter powers of two ($qf = 2^{qi/4}$ for $qi \in \mathbb{N}$). Allowable quantisation factors are therefore spaced a factor of $2^{-1/4}$ apart. Since the range $\frac{x}{2} < qf \leq x$ covers a range a factor of 2^1 wide, several quantisation factors will fall within this region. QED.

The above proof can be visualised using a log-scaled number line. The following figure shows a log-scale with valid quantisation factors marked, along with an example x and $\frac{x}{2}$ value. In this illustration it can be seen that four valid quantisation factors in $\frac{x}{2} < qf \leq x$ will always be available in this range.



Lemma 4: For $\frac{x}{2} < qf \leq x$, $x'_{qf} = \frac{3}{2}qf$.

That is, for the very largest quantisation factors which may be applied to x, without quantising it to zero (see lemma 1), the quantised value may be computed by the linear expression $x'_{qf} = \frac{3}{2}qf$.

Proof: When $\frac{x}{2} < qf \leq x$, quantise$(x,\ qf) = \left\lfloor \frac{x}{qf} \right\rfloor = 1$. As a consequence:

$$\begin{aligned} x'_{qf} &= \left\lfloor \frac{x}{qf} \right\rfloor qf + \frac{qf}{2} \\ &= 1\, qf + \frac{qf}{2} \\ &= \frac{3}{2}qf \end{aligned}$$

Lemma 5: The largest quantisation factor in $\frac{x}{2} < qf \leq x$, which we will call $qf_{\max,x}$, produces the largest dequantised value for any quantisation factor in this range.

Proof: In the range $\frac{x}{2} < qf \leq x$, lemma 4 shows that $x'_{qf} = \frac{3}{2}qf$. This expression is linear and therefore monotonic. Therefore, the largest qf allowed also produces the largest x'_{qf} possible in this range. QED.

Lemma 6: $2^{-1/4}x < qf_{\max,x} \leq x$

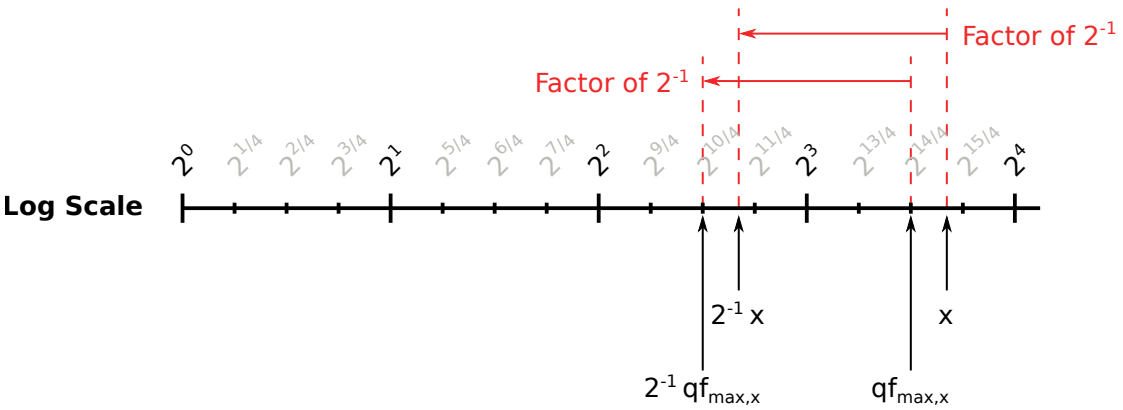
Proof: By definition (lemma 5) the upper bound of $qf_{\max,x}$ is $qf_{\max,x} \leq x$.

Since quantisation factors are spaced at intervals of $2^{1/4}$, the lower bound is therefore $2^{-1/4}x$.

QED.

Lemma 7: If $qf_{\max,x}$ is the largest $qf \leq x$, $\frac{qf_{\max,x}}{2}$ is the largest $qf \leq \frac{x}{2}$.

Proof: The following visual illustration shows a log-scaled number line on which the above values are plotted.



On a log scale, scaling x to $\frac{x}{2}$ moves by a factor of 2^{-1} to the left, or exactly four quantisation factors. As a consequence, the nearest quantisation factor below $\frac{x}{2}$ is also four quantisation factors to the left of $qf_{\max,x}$, that is $\frac{qf_{\max,x}}{2}$.

QED.

Lemma 8: $x'_{qf} \leq x + \frac{qf}{2}$ and therefore we have an upper bound on x'_{qf} which is monotonic with qf.

Proof: For $x > 0$:

$$x'_{qf} = \left\lfloor \frac{x}{qf} \right\rfloor qf + \frac{qf}{2}$$

The effect of the floor operation can be replaced with an error term, $0 \leq e < 1$:

$$\begin{aligned} x'_{qf} &= \left(\frac{x}{qf} - e\right) qf + \frac{qf}{2} \\ &= x - e\,qf + \frac{qf}{2} \end{aligned}$$

Therefore we get the upper bound:

$$x'_{qf} \leq x + \frac{qf}{2}$$

Which is linear and, consequently, monotonic with qf.

QED.

Lemma 9: $x'_{qf} < x'_{qf_{\text{max},x}}$ for all qf in the region $1 \leq qf \leq \frac{x}{2}$.

Proof: By lemma 7, the largest quantisation factor in the range $1 \leq qf \leq \frac{x}{2}$ is $\frac{qf_{\text{max},x}}{2}$. Lemma 8 tells us that this quantisation factor also gives an upper-bound on x'_{qf} for $1 \leq qf \leq \frac{x}{2}$:

$$x'_{\frac{qf_{\text{max},x}}{2}} \leq x + \frac{qf_{\text{max},x}/2}{2}$$

Since $\frac{qf_{\text{max},x}}{2} \leq \frac{x}{2}$ (lemmas 6 and 7), we can substitute the former for the latter in the inequality to get:

$$\begin{aligned} x'_{\frac{qf_{\text{max},x}}{2}} &\leq x + \frac{x/2}{2} \\ &\leq x + \frac{x}{4} \\ &\leq \frac{5}{4}x \\ &\leq 1.25\,x \end{aligned}$$

Lemma 4 states that:

$$x'_{qf_{\text{max},x}} = \frac{3}{2} qf_{\text{max},x}$$

Lemma 6 gives a lower-bound for $qf_{\text{max},x}$ in terms of x, leading to the inequality:

$$\begin{aligned} x'_{qf_{\text{max},x}} &> \frac{3}{2} 2^{-1/4} x \\ &> 1.261\ldots\,x \end{aligned}$$

From this we can conclude that:

$$x'_{\frac{qf_{\text{max},x}}{2}} < x'_{qf_{\text{max},x}}$$

And since we considered the upper-bound for $x'_{\frac{qf_{\text{max},x}}{2}}$, which is monotonic with qf (lemma 8), we can therefore state that:

$$x'_{qf} < x'_{qf_{\text{max},x}} \quad \text{for} \quad 1 \leq qf \leq \frac{x}{2}$$

QED.

Lemma 10: The largest x'_{qf} for any qf is produced for the largest non-zero-producing quantisation factor, $qf_{\text{max},x}$.

Proof: From the lemmas above:

- There exists at least one quantisation factor in the range $\frac{x}{2} < qf_{\text{max},x} \leq x$ (lemma 3).
- Within this range, the largest quantisation factor, $qf_{\text{max},x}$, also produces the largest dequantised value, $x'_{qf_{\text{max},x}}$ (lemma 5).
- For $qf > qf_{\text{max},x}$ we get $x'_{qf} = 0$ (lemma 1).
- For $qf \leq \frac{x}{2}$ have shown that $x'_{qf} < x'_{qf_{\text{max},x}}$ (lemma 9).

Therefore, $x'_{qf} \leq x'_{qf_{\text{max},x}}$ for *all* qf.

QED.

8.5.4 Problem solution

Using the lemmas above we are able to define a solution to our original problem statement, repeated here for convenience:

> Given some value, x_{max}, what is the largest-magnitude value which may be produced by dequantise(quantise(x, qf), qf) for $-x_{\text{max}} \leq x \leq x_{\text{max}}$ and any qf?

Lemma 10 tells us that the largest dequantised value for x_{max} will be $qf_{\text{max},x_{\text{max}}}$, that is, the largest quantisation factor that doesn't quantise x_{max} to zero. Any other quantisation factor will never quantise x_{max} to a larger value. Lemma 2 tells us that replacing x_{max} with any $x < x_{\text{max}}$ will also never produce a larger dequantised value.

The solution to the problem, therefore, is:

$$\text{largest dequantised value} = \text{dequantise}(\text{quantise}(x_{\text{max}},\ qf_{\text{max},x_{\text{max}}}),\ qf_{\text{max},x_{\text{max}}})$$

QED.

8.5.5 Validity under integer arithmetic

Under VC-2's integer arithmetic, all fractional values are truncated towards zero, that is, results are monotonically adjusted downward in magnitude. As a consequence, the monotonicity-related results for the lemmas above hold.

The function $2^{qi/4}$ is approximated in fixed-point arithmetic by the function `quant_factor` in the VC-2 specification. This approximation is accurate to the full precision of the arithmetic used up to quantisation index 134, corresponding with a quantisation factor of $2^{33.5}$. In real applications (which use substantially smaller quantisation factors), the approximation is accurate.

Finally, to give additional confidence, this solution has been verified empirically for all 20 bit integers.

CHAPTER

NINE

TEST PATTERN GENERATION

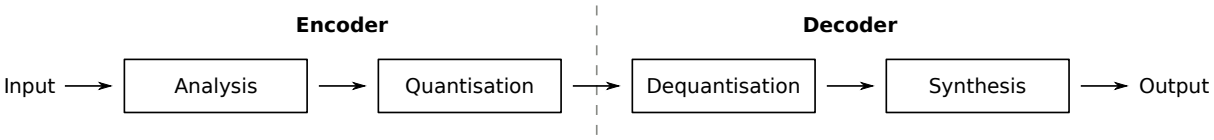
The signal bounds calculated using affine arithmetic provide hard upper bounds on signal levels in both analysis and synthesis filters. These theoretical bounds are complemented with a set of test patterns which are designed to produce near worst-case signal levels in actual codec implementations. While the theoretical bounds computed using affine arithmetic can be over-estimates, the signal levels reached by these test patterns can be considered a lower-bound on the true worst-case signal levels. Between these two approaches a range of plausible signal levels are identified.

9.1 Analysis filter test patterns

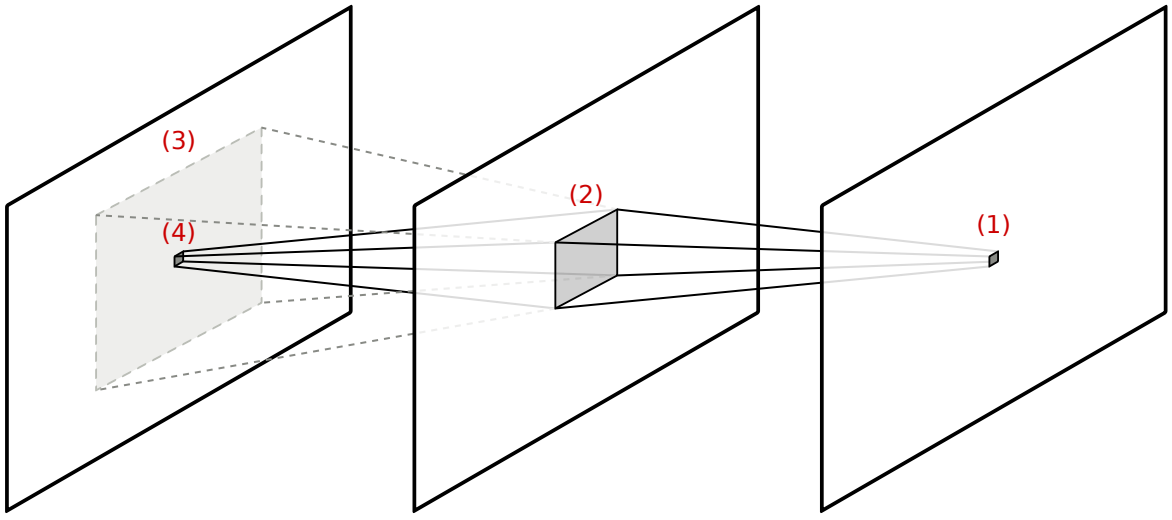
VC-2's analysis filters are *nearly* linear, modulo rounding errors. Since these rounding errors are small, we can produce an acceptable test pattern by treating the analysis filter as a true linear filter.

9.2 Synthesis filter test patterns

The input to VC-2's synthesis filter is a set of transform domain coefficients produced by analysing, quantising and then dequantising a picture:



In the figure below lets work backward through the pipeline starting with an output pixel, (1).



The output pixel is computed using a particular set of transform coefficients, (2), generated by the analysis transform (and mutated by the quantisation/dequantisation step).

Each of the transform coefficients in (2) are ultimately the result of filtering a particular region of the input picture (3).

In the absence of quantisation, only the value of the input pixel, (4), has any effect on the output pixel, (1).

When quantisation perturbs transform coefficients, the contributions of values of other pixels in (3) cease to exactly cancel out and begin to effect the output pixel, (1).

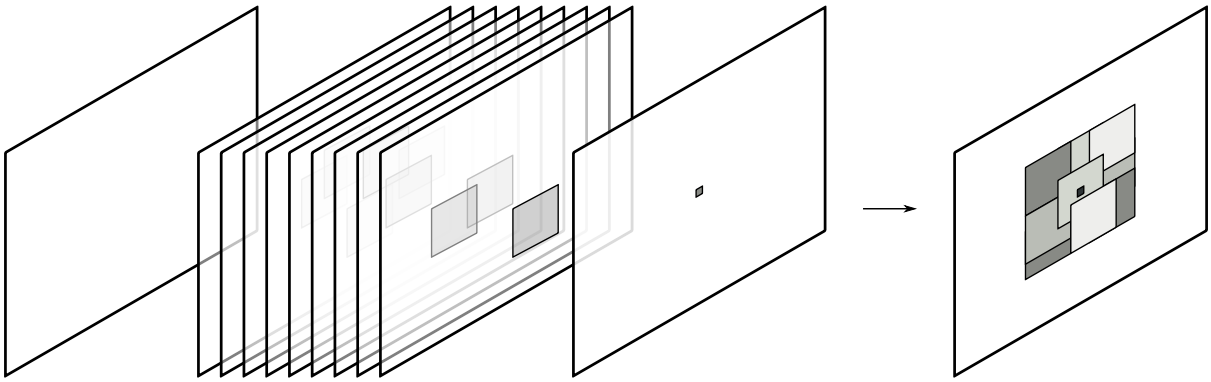
9.2.1 Heuristic synthesis test pattern

To devise a test pattern which maximises a synthesis decoder value, a simple heuristic is employed: the larger the magnitude of the transform coefficients, the larger the energy that can leak into the target value.

A test pattern which simultaneously attempts to maximise the synthesis target value and the transform coefficients is constructed from a 'collage' of test patterns like so:

1. Start with an empty test pattern.
2. Ignoring all non-linearities, enumerate the set of transform coefficients (and their weights) that contribute to the target output of the synthesis filter (labelled '(2)' in the previous illustration). For each transform coefficient in turn, in ascending order of weight magnitude:
    a. Compute the test pattern which maximises this transform coefficient in isolation (i.e. the same way the analysis filter test patterns were produced).
    b. If this transform coefficient has a negative weight, invert the test pattern found in 'a'.
    c. Copy the transform coefficient maximising test pattern into our test pattern, overwriting any previously set pixel values.
3. Ignoring all non-linearities, enumerate the input pixels which directly contribute to the target synthesis filter output (e.g. pixel '(4)' in the earlier illustration). For each pixel with a positive weight, set the corresponding pixel in our test pattern to its maximum value. For pixels with a negative weight, set the corresponding pixel to the minimum value.

The resulting stack-up of test patterns is illustrated in the figure below:



Initial Picture (Step 1)

Transform Coefficient Maximising Test Patterns (Step 2)

Directly Contributing Input Pixels (Step 3)

Combined Test Pattern

This test pattern first-and-foremost prioritises features which directly maximise the target value. Next priority is given to features which maximise the transform coefficients with the largest weight (and therefore largest influence on the target value).

CHAPTER

TEN

EXPERIMENTAL RESULTS

To verify the effectiveness of the test patterns generated by this software a number of experiments have been carried out, the results of which are shown here. Broadly, these experiments are divided into two groups which demonstrate that:

- The heuristic synthesis test pattern performs better than a naive test pattern which assumes filter linearity.
- The generated test patterns produce more extreme signal levels than real pictures or noise signals in practice.

The software used to run these experiments can be found on GitHub[18]. The source of this documentation page includes comments showing the exact command used to produce each of the plots shown.

10.1 Heuristic vs naive synthesis test patterns

The heuristic developed in *Test pattern generation* (page 63) is designed to produce more extreme signal values after quantisation than test patterns which ignore non-linearities in VC-2. To verify this, the following experiment compares the result of passing both types of synthesis test pattern through a VC-2 encoder and decoder.

10.1.1 Method

The following procedure is carried out for each synthesis filter phase (i.e. for each test pattern):

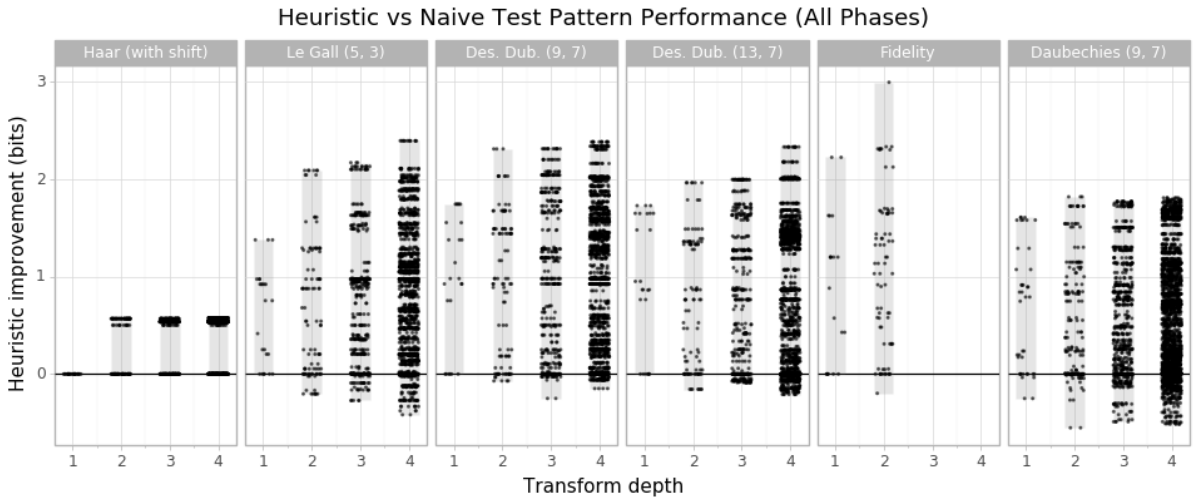
1. Encode the pattern-under-test.
2. Quantise the transform coefficients produced in step 1.
3. Decode the quantised transform coefficients and record the signal value in the targeted part of the synthesis transform.
4. Repeat steps 2-3 for all quantisation indices which don't quantise all transform coefficients to zero.

Quantisation is carried out using a single quantisation index for all picture slices and using the default quantisation matrix for the filter configuration used.

10.1.2 Results

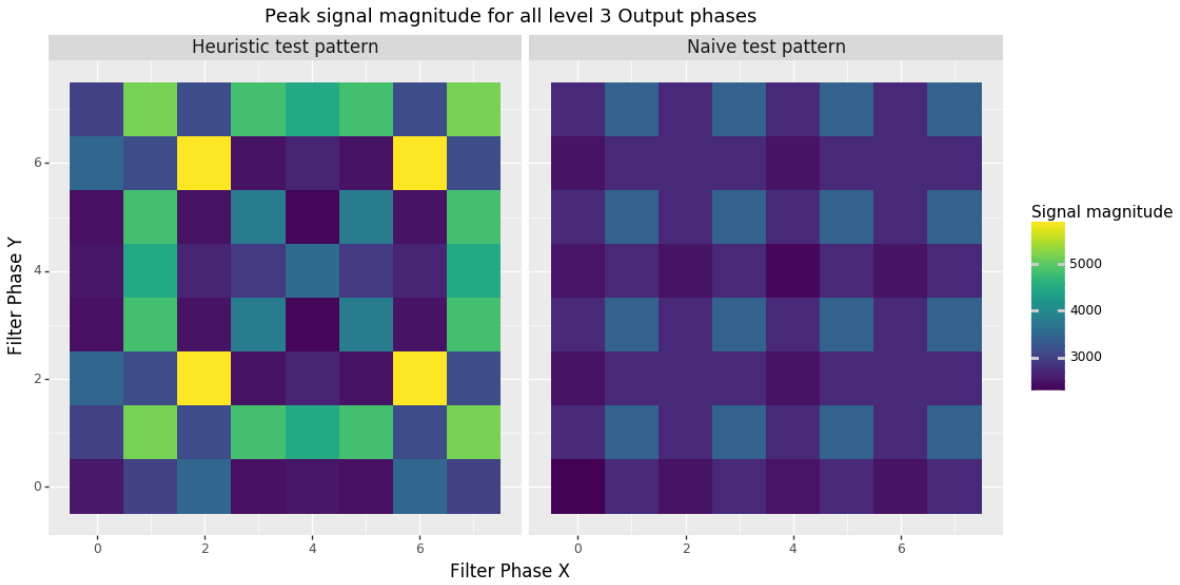
The plot below shows the relative performance of the naive and heuristic test patterns across all arrays and filter phases, broken down by transform depth and wavelet. Each dot shows the relative performance of the test pattern for a particular filter phase.

[18] https://github.com/bbc/vc2-signal-width-experiments/

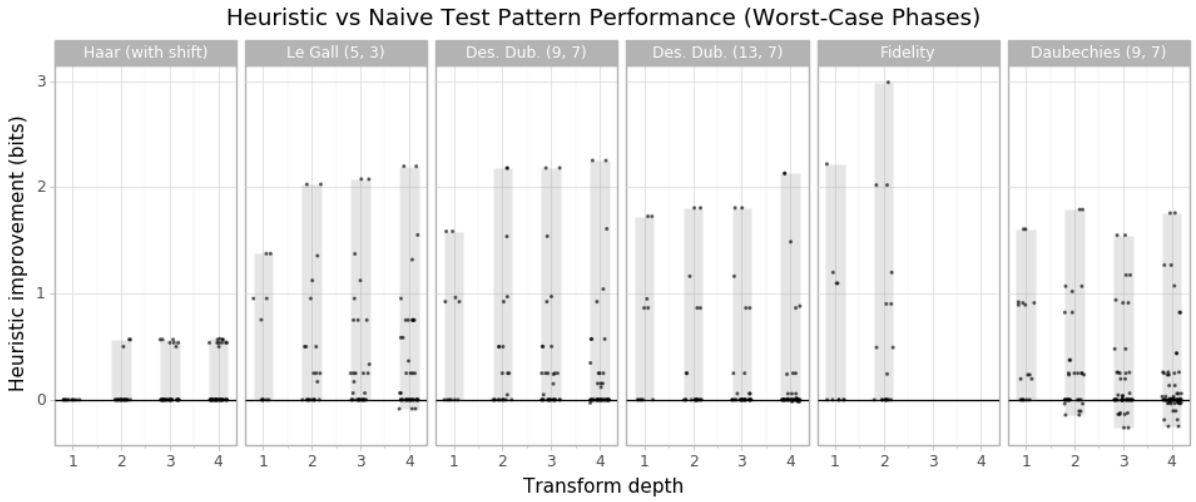


In general, the heuristic test patterns produce signal levels at least one bit larger than the naive test patterns, only rarely producing lower signal levels. Occasional under-performance is to be expected since the heuristic is, well, a heuristic.

The figure above treats each filter phase independently. In practice, most VC-2 implementations will choose a number of bits for each array as a whole rather than using different numbers of bits for different phases. As the figure below illustrates, the signal ranges for a given array can vary substantially between filter phases:



Because of this, it may make more sense to present the effect the heuristic test patterns have on the worst-case signal levels within each array. The plot is redrawn below but this time each dot represents the relative improvement of going from the most extreme value produced by the naive filter to the most extreme value for the heuristic filter for any phase.



Overall, this tells us that for some filter arrays the test patterns produce signal values one or more bits larger than the naive test patterns. Again, the heuristic occasionally produces lower signal levels than the naive test pattern though the effect is lessened when only worst-case phases are considered.

In summary we can conclude that the heuristic test patterns produce larger signal levels than the naive test patterns in most cases.

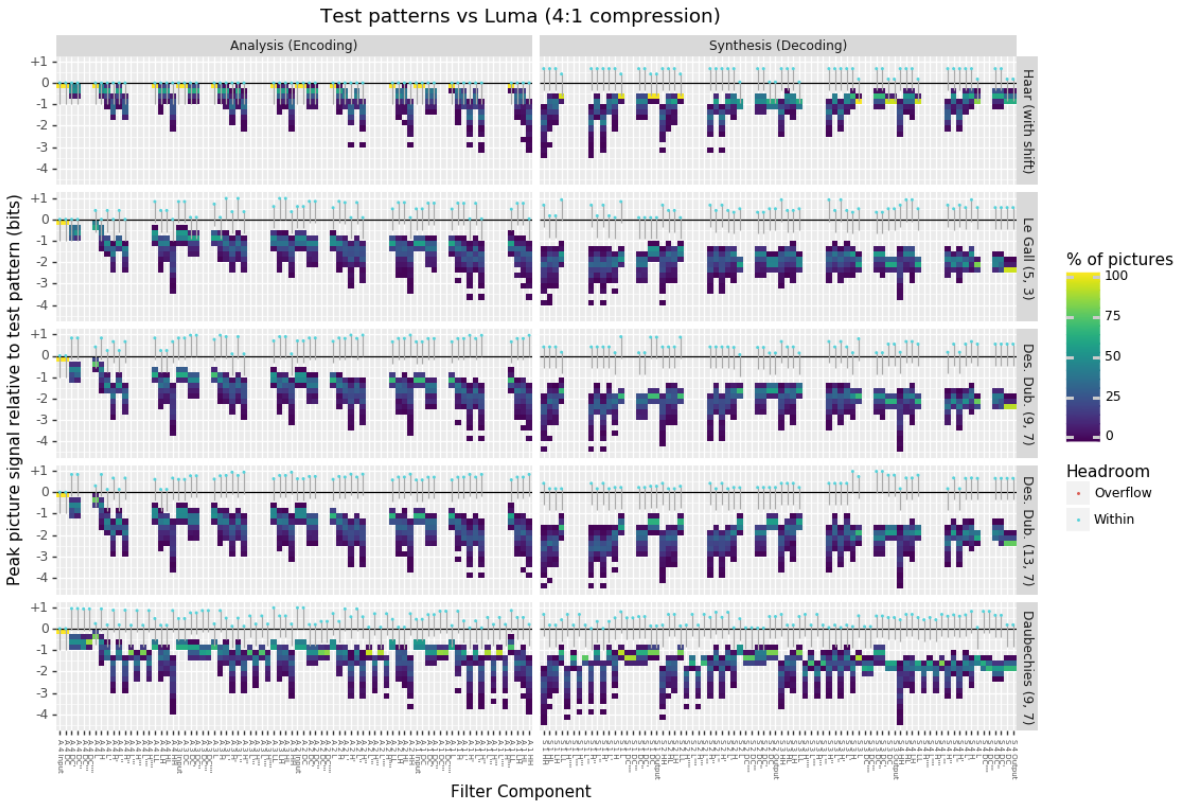
10.2 Comparison with natural images and noise

As noted in *Related work* (page 53), a common approach to picking signal widths in wavelet transforms is to pass images or noise through the filter and observe signal levels in practice. To measure the effectiveness of this approach, an instrumented VC-2 encoder/decoder was used to log the signal levels produced by noise signals, real pictures and the test patterns generated by this software.

The plots below result from 4-level symmetric transforms using the VC-2 default quantisation matrices.

10.2.1 Real pictures, 4:1 compression ratio

This first plot shows the peak signal levels produced by a collection of 48 natrual, 10-bit, luma-only, HD images, compressed at a ratio of 4:1, compared with those produced by the heuristic test patterns.



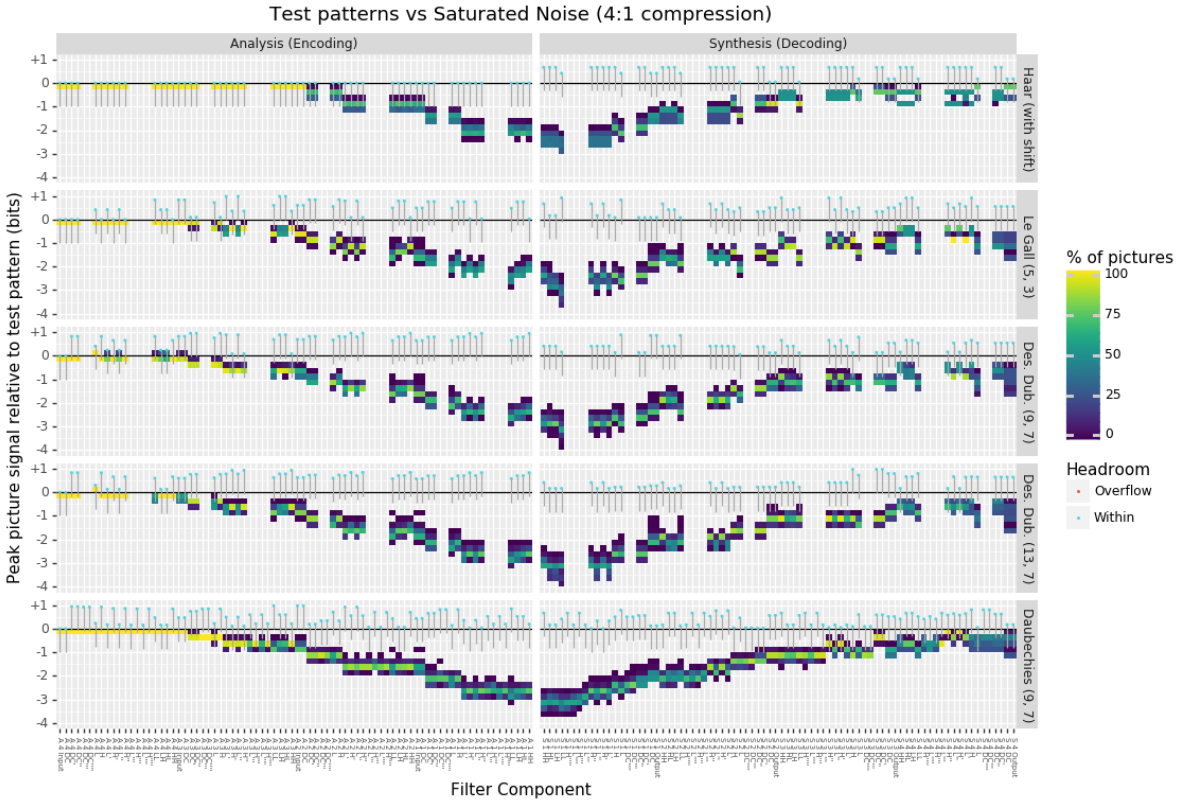
The plot above is broken up into several subplots which show the results for different wavelet transform types.

Within a subplot, each column shows a histogram of the distribution of relative signal values. From this plot we can immediately see that no picture ever produces a signal level larger than the heuristic test patterns. In fact, in the majority of pictures, real pictures produce peak signal levels one or more bits lower than those produced by the test patterns. Some pictures produce peaks over four bits lower than the test patterns for certain filter arrays.

From this plot we can conclude that bit-width requirements are likely to be under-estimated by at least 1 bit for most wavelets if only real pictures are used to determine bit-widths.

10.2.2 Saturated noise, 4:1 compression ratio

The other common test source for chosing bit widths is noise. The plot below shows the peak signal levels achieved by 300 10-bit, HD, saturated, uniform random noise pictures, compressed at a 4:1 ratio.



This time, the peak signal levels of the noise signals more closely matches that of the test patterns for early synthesis filter arrays. Deeper within the transform, however, the test patterns begin to produce larger signal levels by as many as two bits or more at the deepest parts of the transforms.

Under the two Deslauriers Dubuc wavelets, however, some noise signals peak slightly above the test patterns in a small number of filter arrays. In practice, however, the bit widths required to support the test patterns are also sufficient for these peaks in this case. This is visualised by the 'drawing pins' in the plot.

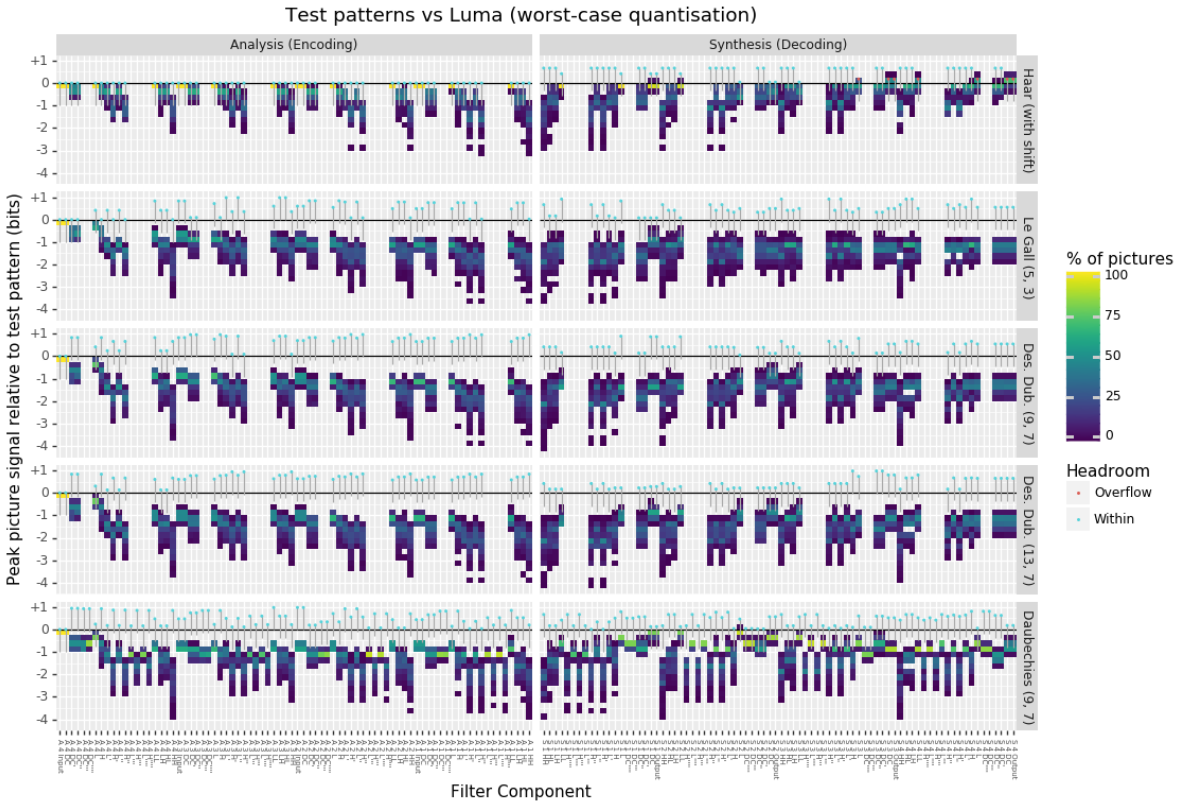
The body of each 'drawing pin' shows the range of values which would be rounded up to the same number of bits as the test pattern. In this case, we can see that most of the noise overshoots are comfortably within this range. In the more ambiguous cases, the colour of the pin head indicates whether any picture actually exceeded the number of bits used by the test pattern. In this case all of the pins are cyan indicating the test patterns indicated the correct number of bits, even if the absolute signal range was a slight underestimate.

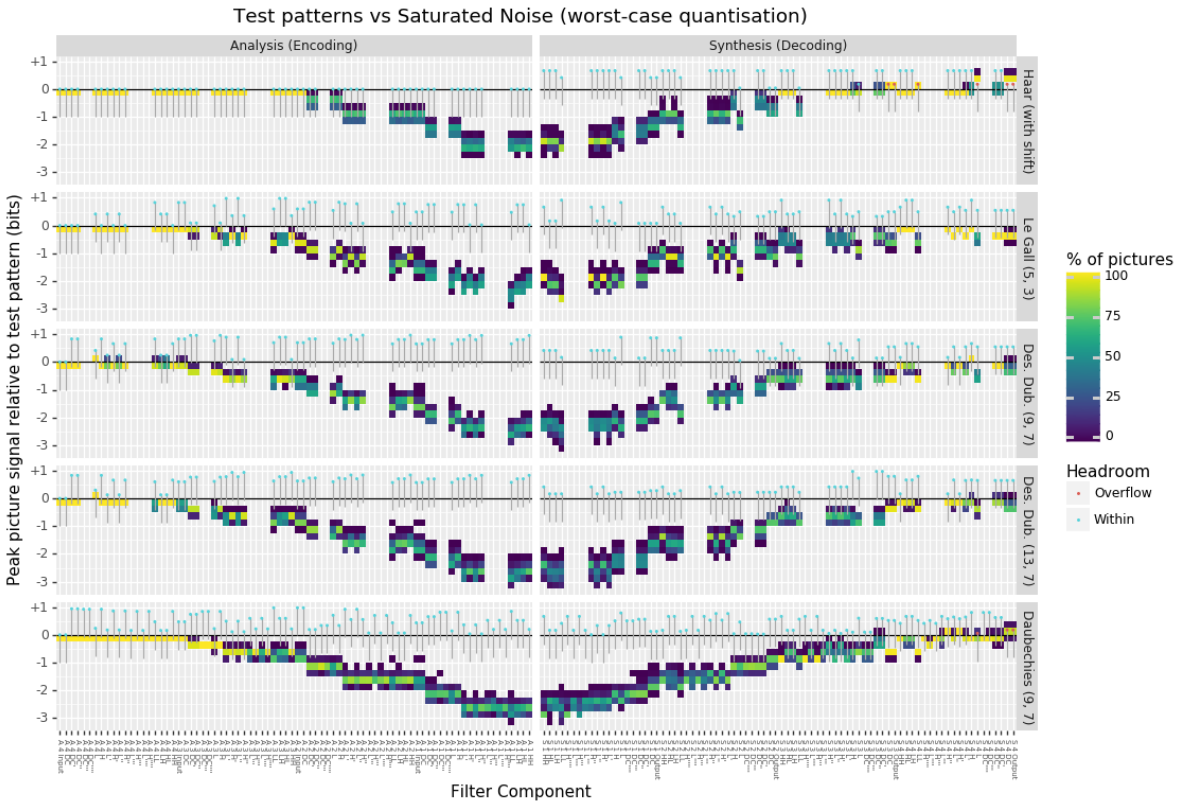
The main conclusions to be drawn from this graph is that the heuristic test patterns also produce significantly higher signal levels than noise signals. In particular, in deep parts of most transforms the signal levels are under-estimated by over two bits – a worse under-estimate than produced by real pictures.

10.2.3 Worst-case quantisation

Both of the plots above show pictures compressed by a typical 4:1 compression ratio. In practice, the most extreme signal levels are produced under larger quantisation levels – and indeed the test patterns use much larger quantisation levels.

The two plots below instead show the picture and noise signal levels at whatever quantisation index makes them largest – i.e. worst-case quantisation.





In general the trends are essentially the same with both real pictures and noise being likely to under estimate signal levels, particularly deeper in the transform.

The most significant difference is that the Haar transform test patterns are out-performed by both real pictures and noise in the final stages of the synthesis filters. This shortcoming presents a possible motivation for using *vc2-optimise-synthesis-test-patterns* (page 26) to optimise test patterns for the Haar transform.

10.2.4 Method

Experiments used the following input pictures:

- **Real pictures (48 natural images)**
  - YCbCr and RGB formats
  - 8, 10, 12 and 16 bit depth
  - UHD and HD resolution scalings

For a total of 2304 picture components (48 pictures $\times$ 6 picture components (Y, Cb and Cr; R, G and B) $\times$ 4 bit depths $\times$ 2 resolutions (UHD and HD).

- **Noise signals (300 noise pictures)**
  - Saturated and non-saturated uniform random (white) noise
  - 8, 10, 12 and 16 bit depth
  - HD resolution only

For a total of 2400 (monochrome) noise pictures (300 noise samples $\times$ 2 noise types $\times$ 4 bit depths).

The following codec configurations were tested:

- **Wavelet:**
  - Haar (with shift)
  - Le Gall (5, 3)

  - Deslauriers-Dubuc (9, 7)
  - Deslauriers-Dubuc (13, 7)
  - Daubechies (9, 7)
  - Fidelity
- **Transform depth**
  - 2 levels (symmetric)
  - 4 levels (symmetric)
- **Quantisation matrix**
  - Default quantisation matrix used for the wavelet/depth chosen
- **Quantisation index (applied globally to all picture slices)**
  - All possible quantisation indices used

For a total of 768 codec configurations (6 wavelets $\times$ 2 transform depths $\times$ (approx) 64 quantisation indices). This total is approximate as the number of quantisation indices tested varies depending on the picture being encoded.

Each component (channel) of every test picture and noise plate is individually encoded (analysed), quantised and decoded (synthesised) using each configuration of the codec enumerated above. In each of these runs, the peak signal levels in each array in the encoder and decoder (see *Terminology* (page 6)) are recorded.

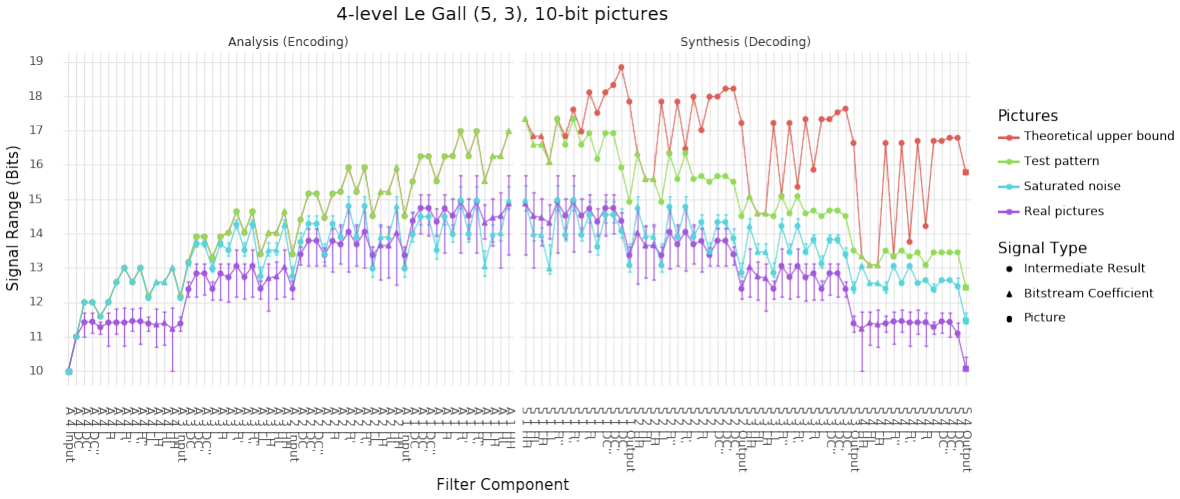
In total approximately 3,612,672 picture component and codec configuration combinations were tested (768 configurations $\times$ 2304 + 2400 picture components).

10.2.5 Detailed Results

Due to VC-2's flexibility, the experiments carried out include a fairly large number of variables. Many of these variables have little impact on the general trends in the results. As a consequence, we begin by looking at specific examples which demonstrate general trends before exploring the effects of different codec configurations.

General trends

The plot below shows the worst-case signal levels in each array of a 4-level Le Gall (5, 3) transform acting on 10 bit pictures.



The 'Theoretical upper bound' line gives the upper-bound computed according to affine arithmetic (see *Computing signal bounds with Affine Arithmetic* (page 55)). The 'Test pattern' line shows the signal levels reached by the test patterns generated by this software (see *Test pattern generation* (page 63)).

In the analysis filter, the test patterns almost exactly reach the theoretical upper bound. In the synthesis filter, however, quantisation causes the theoretical upper bounds to grow well beyond the level of the test patterns. In all codec configurations, the synthesis filter's theoretical upper bounds appear to be significantly over-estimated (by several bits) compared with signal levels observed in practice.

The 'Saturated noise' and 'Real pictures' lines show the signal levels reached by saturated noise signals and real HD luma (Y) picture components respectively. The lines show the mean peak signal across all pictures while the error bars show the range. The results are shown for the quantisation index which achieves an overall 4:1 compression ratio typical of VC-2 applications.

Note: In these experiments, the encoder encodes the entire picture as a single picture slice using the lowest quantisation index which fits the required picture data. This is a simplification of real encoder behaviour necessary to keep the parameter space for these experiments under control.

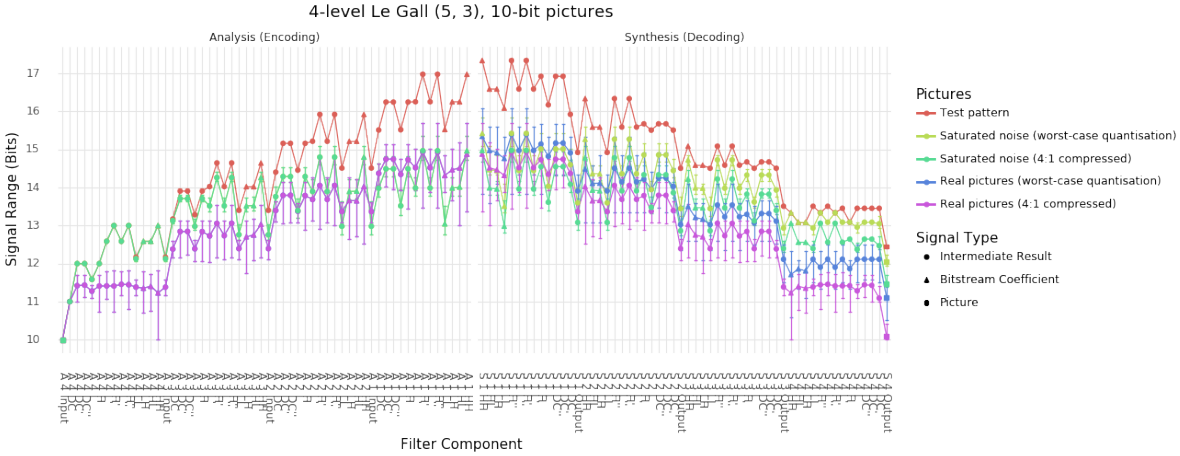
A key result shown in this plot is that at every part of the analysis and synthesis filters, the test patterns produce signal levels at least as large as the pictures or noise. In many cases, the test patterns produce peak signal levels over 1 bit larger than the pictures and noise. This means that had these real pictures or noise been used to pick bit widths for a VC-2 implementation, these would have under-estimated the required number of bits.

A secondary observation is that the ability of random noise signals to find extreme signal levels reduces at deeper levels of the transform and also following quantisation. In fact, this effect is so pronounced that at the deepest part of the transform, real pictures actually produce more extreme signal levels than the noise. This may be explained by these parts of the transform being dominated by low-spatial-frequency content which natural pictures are heavily skewed towards.

Effects of quantisation

The previous plot showed the signal levels reached when real pictures and noise are passed through a VC-2 codec using quantisation indices consistent with a typical 4:1 compression ratio. By design, the quantisation levels required to achieve this level of compression produce only small errors. At higher quantisation levels, larger errors are produced which can lead to more extreme signal values being produced from real picture and noise signals.

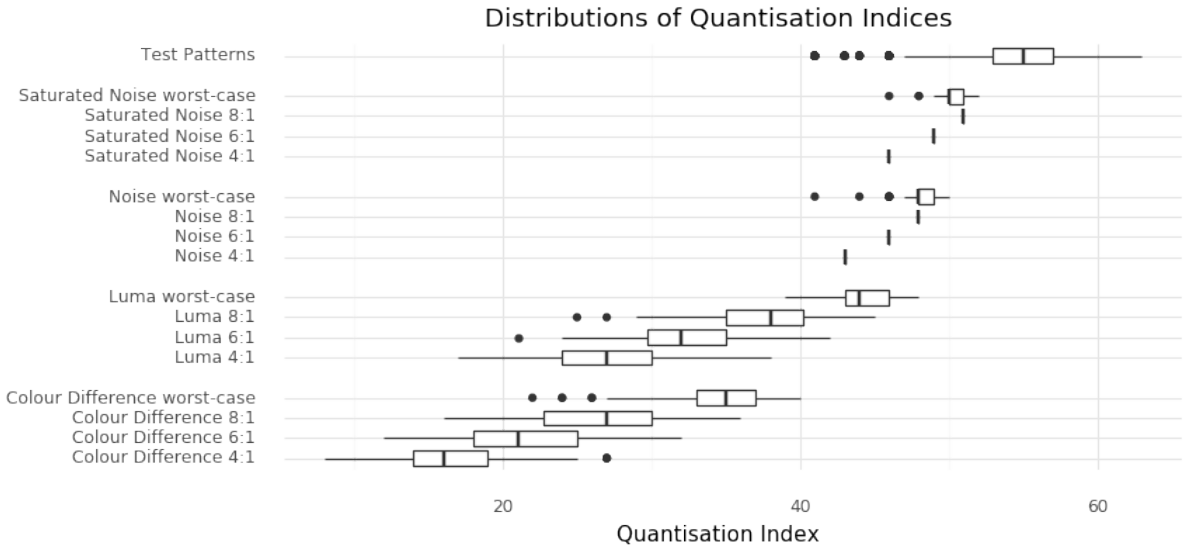
To illustrate the relative effects of atypical-quantisation levels, the plot below compares the signal levels produced for 4:1 compressed pictures and noise and 'worst-case' quantisations of those same pictures. The 'worst-case' values report the most extreme signal value produced at *any* quantisation index.



As the plots show, worst-case quantisation produces consistently higher signal levels than those found under 4:1 compression. Nevertheless, these signal levels remain below the signal levels produced by the test patterns.

Once again, a 4-level Le Gall (5, 3) transform acting on 10 bit pictures is shown above but the pattern is found to be consistent across other configurations as is discussed in greater detail later.

Note: The 'worst-case' quantisation levels used in the plot above are much higher than those used under typical compression ratios. The boxplots below illustrate the distributions of quantisation indices which are used in practice.



The 'Test patterns' boxplot shows the distribution of quantisation indices used by the heuristic synthesis test patterns. The other box plots show the actual distributions of quantisation indices used to encode the test pictures and noise samples at various compression ratios.

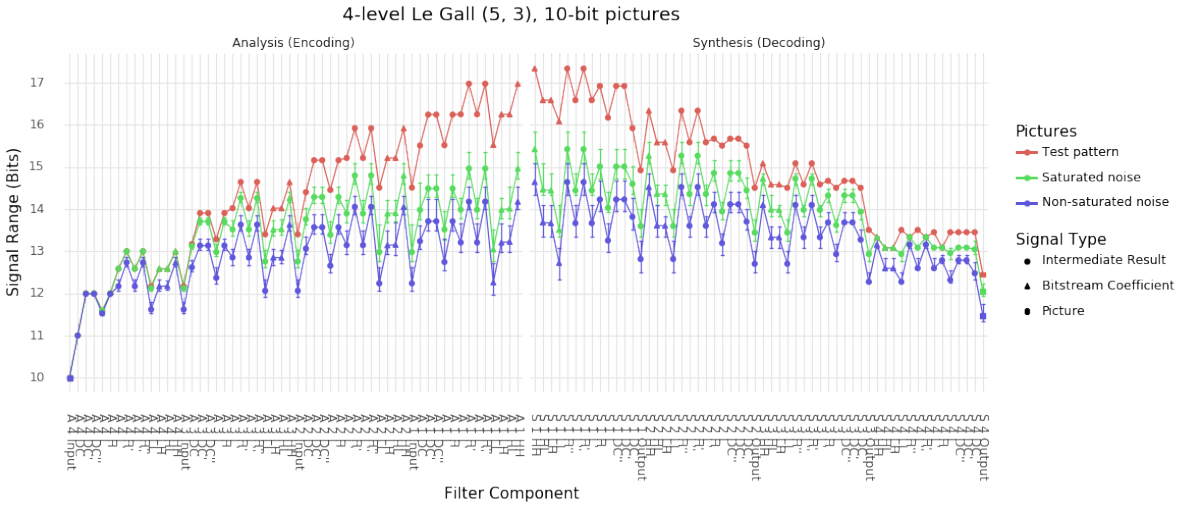
The plots show that the quantisation levels used for real pictures at typical (i.e. 4:1) or larger (6:1 and 8:1) compression ratios are significantly lower than worst-case quantisation levels, and those levels used by the test patterns. Even in the case of noise signals, the quantisation levels used still fall short of these worst-case levels.

The plot specifically shows the quantisation indices used by a 4-level Le Gall (5, 3) transform for 10 bit input signals however the general trend is consistent accross configurations.

This result implies that even if encoders are deliberately configured to use very large quantisation levels, real pictures and noise signals still do not produce the signal levels produced by the heuristic test patterns.

Noise types

The plot below compares the signal levels achieved by non-saturated and saturated noise signals, shown for worst-case quantisation indices. Again, a 4-level Le Gall (5, 3) transform is shown acting on 10 bit inputs.

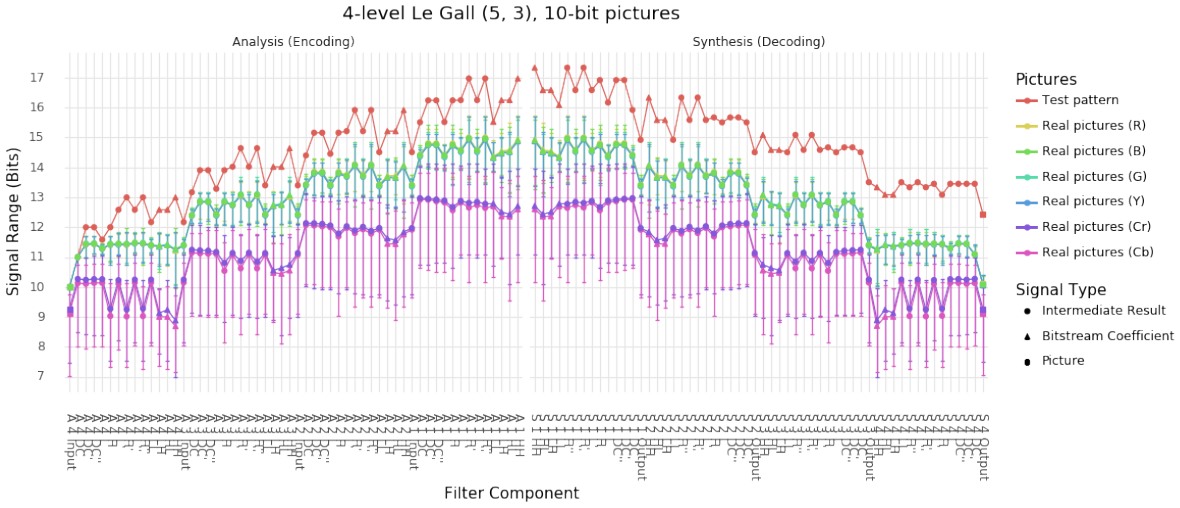


As might be expected, saturated noise results in higher worst-case signal levels, a pattern found to be consistent across all arrays and all noise samples and codec configurations tested. This confirms that saturated noise makes a better test signal for finding extreme signal values than unsaturated noise.

Note: A singular exception to the rule that saturated noise produces larger signals than non-saturated noise was found in the experimental data. Specifically the final output stage of the 2-level Haar (with shift) synthesis transform, non-saturated noise produced worse-case signals around 0.2 bits larger than saturated noise. Due to the isolated incidence of this rule, It is assumed that this outcome is the product of random chance and that over a larger number of noise pictures, the rule would hold.

Picture components

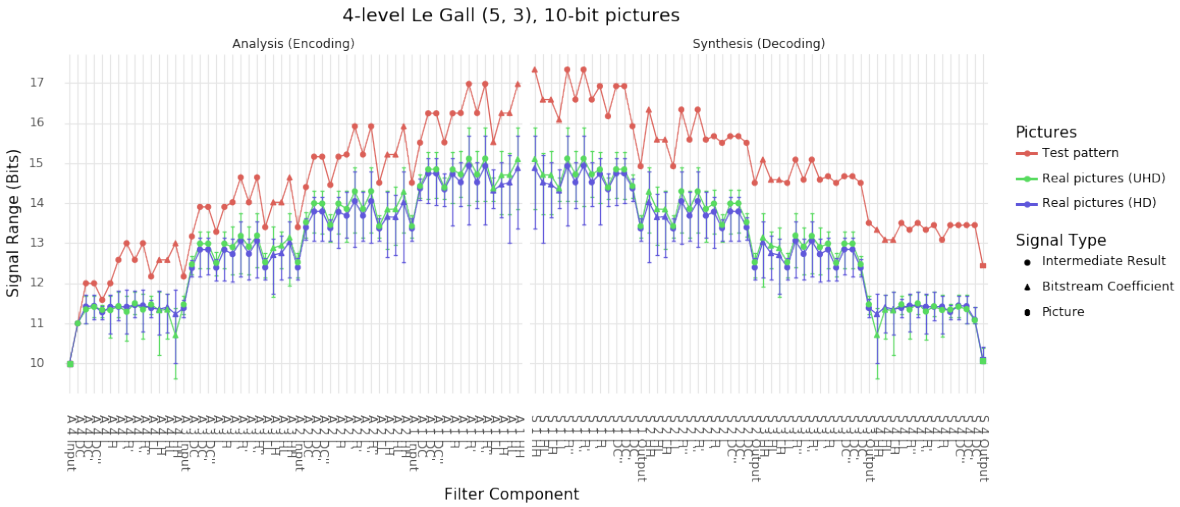
The plot below compares the signal levels produced by different colour components of the real picture signals. Again, a 4-level Le Gall (5, 3) transform is shown acting on 10 bit inputs and 4:1 compression.



As might be expected, the signal levels from the luma component (Y) in a YCbCr picture and the components of an RGB picture are extremely similar. Likewise, the colour difference signals (Cb and Cr) from a YCbCr picture show much lower signal ranges due to relatively low signal levels encountered in typical pictures. As a consequence, we only consider the luma component of real picture signals in these experiments.

Picture size

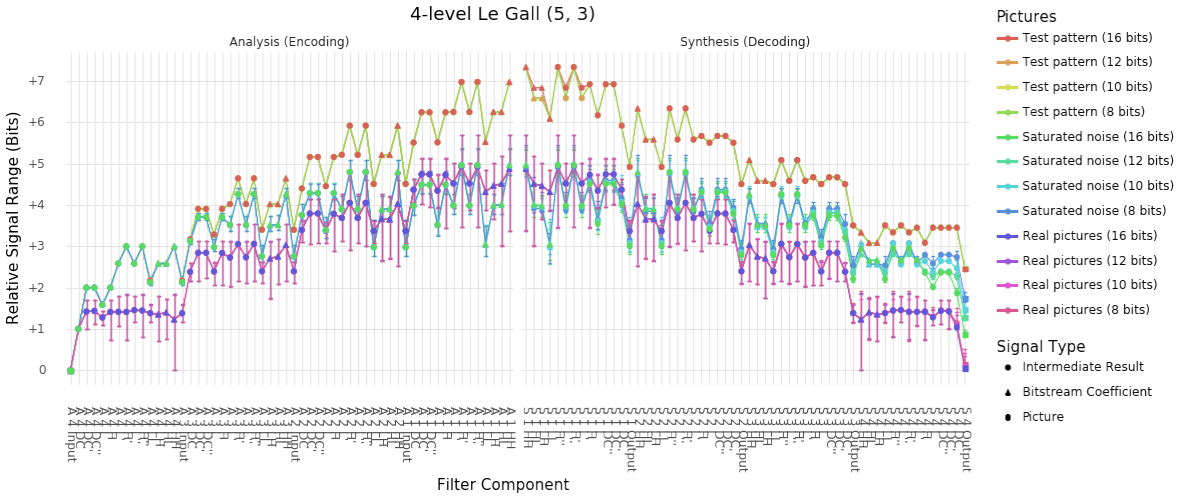
The plot below compares the signal levels produced by HD and UHD real picture signals. Again, a 4-level Le Gall (5, 3) transform is shown acting on 10 bit inputs and 4:1 compression.



The signal levels encountered at the different resolutions are broadly similar with UHD signals producing slightly larger signal levels deeper in the transform and slightly smaller signals nearer the start/end. Due to the similarity of the signals produced by the two picture sizes, only HD signals are shown in the other plots.

Bit Depth

The plot below compares the signal levels produced by different picture bit depths (on a relative scale). Again, a 4-level Le Gall (5, 3) transform is shown with 4:1 compression.



As shown, the results are essentially indistinguishable at every bit width, though there are some occasional (small) differences which are assumed to be due to differences in where quantisation boundaries fall. Since the impact on bit widths on the overall trends in the results is extremely small, only 10 bit examples are shown.

Part III

Internals and Low-Level API

CHAPTER

ELEVEN

LOW-LEVEL API OVERVIEW

The remaining documentation defines the low-level APIs which are used internally to implement the high-level APIs (see *High-level Python API Reference* (page 35)). This documentation is primarily intended for developers of `vc2_bit_widths` (page 33) but also potentially also researchers requiring unusual or low-level functionality.

The sections below are arranged in approximately reverse-dependency order with higher-level utilities listed first.

11.1 Evaluation and deployment of test patterns

The `pattern_evaluation` (page 80) module provide functions for evaluating test patterns (by passing them through a VC-2 encoder/decoder). Once evaluated, these test patterns may then be grouped together (according to e.g. required quantisation index) and packed into full-sized pictures using `picture_packing` (page 83).

11.2 Calculating signal bounds and test patterns

The `signal_bounds` (page 86) module performs static analysis on VC-2 filters to calculate theoretical worst-case signal bounds. These analyses may then be converted into test patterns using the components in the `pattern_generation` (page 89) module and optionally further optimised using algorithms in the `pattern_optimisation` (page 92) module.

11.3 Specialised partial evaluation functions for VC-2 filters

The `fast_partial_analysis_transform` (page 96) and `fast_partial_analyse_quantise_synthesise` (page 98) modules provide specialised implementations of VC-2's analysis and synthesis transforms (respectively).

These implementations compute 'partial' analyses and syntheses meaning they only calculate one analysis or synthesis output value. Since this is often all that is required by this software, ths can save a significant amount of computation.

These implementations are also able to directly compute intermediate results which would ordinarily be accessible in ordinary analysis or synthesis implementations.

Note: These implementations are still Python based (albeit with some `numpy`[19] acceleration) and so are so the 'fast' monicer only truly applies when compared with the VC-2 pseudocode implemented in Python.

---

[19] https://numpy.org/doc/stable/reference/index.html#module-numpy

11.4 Implementations of VC-2's quantiser and wavelet filters

The *quantisation* (page 101) module provides an implementation of VC-2's quantisation and dequantisation routines, along with various utilities for bounding the quantiser's outputs or inputs.

The *vc2_filters* (page 103) implements VC-2's wavelet filters in terms of *InfiniteArray* (page 125)s. These are used to construct the algebraic descriptions of VC-2's filters used to construct test patterns and signal bounds. They are also used to derive efficient partial implementations of these filters in *fast_partial_analyse_quantise_synthesise* (page 98).

11.5 Utilities for representing and implementing filter behaviour

Finally *vc2_bit_widths* (page 33) contains a number of low-level mathematical modules which provide the foundations upon which the others are built.

The *linexp* (page 110) module implements a specialised Computer Algebra System[20] optimised for building and manipulating algebraic descriptions of VC-2's filters.

The *pyexp* (page 115) module implements a scheme for extracting Python functions which compute a single value from the output of a function acting on whole arrays. This functionality is used within *fast_partial_analyse_quantise_synthesise* (page 98) to build efficient partial implementations of VC-2's synthesis filters.

Finally, *infinite_arrays* (page 120) provide an abstraction for describing the operation of VC-2's filters in terms of array operations. As well as being suited to building the algebraic descriptions used in this module, this form of description itself directly provides several useful metrics about a filter and its outputs.

[20] https://en.wikipedia.org/wiki/Computer_algebra_system

CHAPTER

TWELVE

VC2_BIT_WIDTHS.PATTERN_EVALUATION: MEASURE THE OUTPUTS ACHIEVED BY TEST PATTERNS

This module provides routines for passing test patterns through a VC encoder and decoder and determining the values produced.

12.1 Evaluation functions

evaluate_analysis_test_pattern_output(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *level*, *array_name*, *test_pattern_specification*, *input_min*, *input_max*)

Given an analysis test pattern (e.g. created using `make_analysis_maximising_pattern()`), return the actual intermediate encoder value when the signal is processed.

Parameters

h_filter_params [`vc2_data_tables.LiftingFilterParameters` ([vc2_data_tables], page 7)]

v_filter_params [`vc2_data_tables.LiftingFilterParameters` ([vc2_data_tables], page 7)] Horizontal and vertical filter *synthesis* (not analysis!) filter parameters (e.g. from `vc2_data_tables.LIFTING_FILTERS` ([vc2_data_tables], page 7)) defining the wavelet transform used.

dwt_depth [int]

dwt_depth_ho [int] The transform depth used by the filters.

level [int]

array_name [str] The intermediate value in the encoder the test pattern targets.

test_pattern_specification [*TestPatternSpecification* (page 41)] The test pattern to evaluate.

input_min [int]

input_max [int] The minimum and maximum value which may be used in the test pattern.

Returns

encoded_minimum [int]

encoded_maximum [int] The target encoder value when the test pattern encoded with minimising and maximising signal levels respectively.

evaluate_synthesis_test_pattern_output(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *quantisation_matrix*, *synthesis_pyexp*, *test_pattern_specification*, *input_min*, *input_max*, *max_quantisation_index*)

Given a synthesis test pattern (e.g. created using `make_synthesis_maximising_pattern()` or

optimise_synthesis_maximising_test_pattern()), return the actual decoder value, and worst-case quantisation index when the signal is processed.

Parameters

h_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)] Horizontal and vertical filter *synthesis* (not analysis!) filter parameters (e.g. from vc2_data_tables.LIFTING_FILTERS ([vc2_data_tables], page 7)) defining the wavelet transform used.

dwt_depth [int]

dwt_depth_ho [int] The transform depth used by the filters.

quantisation_matrix [{level: {orient: int, ...}, ...}] The VC-2 quantisation matrix to use.

synthesis_pyexp [PyExp] A *PyExp* (page 119) expression which defines the synthesis process for the decoder value the test pattern is maximising/minimising. Such an expression is usually obtained from the use of *synthesis_transform()* (page 108) and *make_variable_coeff_arrays()* (page 109).

test_pattern_specification [*TestPatternSpecification* (page 41)] The test pattern to evaluate.

input_min [int]

input_max [int] The minimum and maximum value which may be used in the test pattern.

max_quantisation_index [int] The maximum quantisation index to use. This should be set high enough that at the highest quantisation level all transform coefficients are quantised to zero.

Returns

decoded_minimum [(value, quantisation_index)]

decoded_maximum [(value, quantisation_index)] The target decoded value (and worst-case quantisation index) when the test pattern is encoded using minimising and maximising values respectively.

12.2 Utility functions

convert_test_pattern_to_padded_picture_and_slice(*test_pattern*, *input_min*, *input_max*, *dwt_depth*, *dwt_depth_ho*)

Convert a description of a test pattern (in terms of polarities) into a numpy.array, padded ready for processing with a filter with the specified transform depths.

Parameters

test_pattern [*TestPattern* (page 42)]

input_min [int]

input_max [int] The full signal range to expand the test pattern to.

dwt_depth [int]

dwt_depth_ho [int] The transform depth used by the filters.

Returns

picture [numpy.array] A 2D array containing the test picture (with all undefined pixels set to 0.

picture_slice [(slice[21], slice[22])] A 2D slice out of `test_pattern` which contains the active pixels in `test_pattern` (i.e. excluding any padding).

[21] https://docs.python.org/3/library/functions.html#slice
[22] https://docs.python.org/3/library/functions.html#slice

CHAPTER

THIRTEEN

VC2_BIT_WIDTHS.PICTURE_PACKING: PACK TEST PATTERNS INTO PICTURES

This module contains a simple packing algorithm which attempts to efficiently pack a collection of test patterns onto a small number of test pictures.

13.1 API

pack_test_patterns(*width*, *height*, *test_pattern_specifications*)

Given a picture size and a series of test patterns, pack those patterns onto as few pictures as possible.

Parameters

width, height [int] The size of the pictures on to which the test patterns should be allocated.

test_pattern_specifications [{key: *TestPatternSpecification* (page 41), ...}] The test patterns to be packed.

Returns

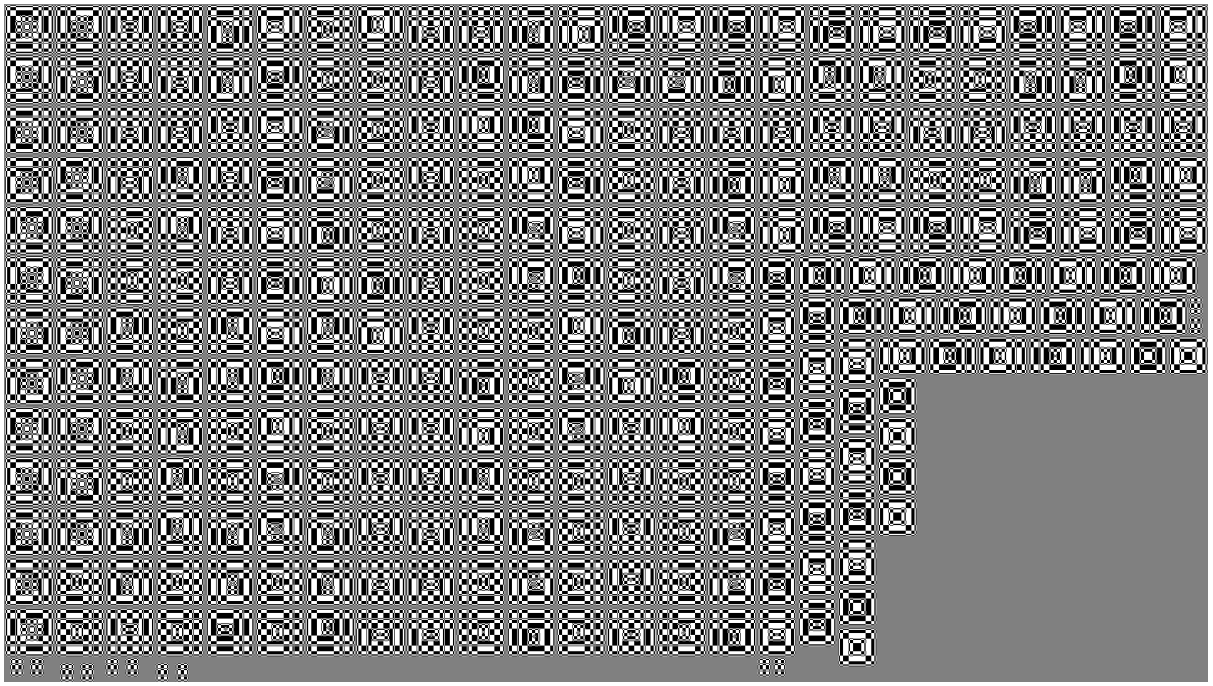
pictures [[numpy.array, ...]] A series of pictures containing test patterns.

locations [{key: (picture_number, tx, ty), ...}] For each of the supplied test patterns, gives the index of the picture it was assigned to and the coordinates (within the targeted filter array) of the array value which will be maximised/minimised.

If any of the supplied test patterns are too large to fit in the specified width and height, those patterns will be omitted (and corresponding entries in the 'locations' will also be omitted).

13.2 Example output

The *pack_test_patterns()* (page 85) function takes a dictionary of *TestPatternSpecification* (page 41) or *OptimisedTestPatternSpecification* (page 41) objects and arranges these over as few pictures as possible. An example output picture is shown below:



In typical use, several independent sets of packed test pictures should be created. One set can be created for all analysis transform test patterns. For synthesis transform test patterns, the patterns should be divided up into groups based on the quantisation index which should be used.

CHAPTER

FOURTEEN

VC2_BIT_WIDTHS.SIGNAL_BOUNDS: FINDING BOUNDS FOR FILTER OUTPUT VALUES

The *vc2_bit_widths.signal_bounds* (page 86) module contains functions for computing theoretical lower- and upper-bounds for VC-2 codec intermediate and final values.

14.1 Finding signal bounds

The following functions may be used to convert algebraic descriptions of a VC-2 filters into worst-case signal ranges (according to an affine arithmetic based model of rounding and quantisation).

The process of finding signal bounds is split into two stages:

1. Finding a generic algebraic expression for worst-case signal bounds.
2. Evaluating those expressions for a particular picture bit width.

The two steps are split this way to allow step 2 to be inexpensively repeated for different picture bit widths.

14.1.1 Analysis filter

analysis_filter_bounds(*expression*)

Find the lower- and upper-bound reachable in a *LinExp* (page 114) describing an analysis filter.

The filter expression must consist of only affine error symbols (*AAError* (page 114)) and symbols of the form `(_, x, y)` representing pixel values in an input picture.

Parameters

expression [*LinExp* (page 114)]

Returns

lower_bound [*LinExp* (page 114)]

upper_bound [*LinExp* (page 114)] Algebraic expressions for the lower and upper bounds for the signal level. These expressions are given in terms of the symbols `LinExp("signal_min")` and `LinExp("signal_max")`, which represent the minimum and maximum picture signal levels respectively.

evaluate_analysis_filter_bounds(*lower_bound_exp*, *upper_bound_exp*, *num_bits*)

Convert a pair of symbolic analysis filter bounds into concrete, numerical lower/upper bounds for an input picture signal with the specified number of bits.

Parameters

lower_bound_exp [*LinExp* (page 114)]

upper_bound_exp [*LinExp* (page 114)] Analysis filter bounds expressions from *analysis_filter_bounds()* (page 87).

num_bits [int] The number of bits in the signal values being analysed.

Returns

lower_bound [int]

upper_bound [int] The concrete (numerical) bounds for the analysis filter given `num_bits` input pictures.

14.1.2 Synthesis filter

`synthesis_filter_bounds`(*expression*)
Find the lower- and upper-bound reachable in a *`LinExp`* (page 114) describing a synthesis filter.

The filter expression must contain only affine error symbols (*`AAError`* (page 114)) and symbols of the form `((_, level, orient), x, y)` representing transform coefficients.

Parameters

expression [*`LinExp`* (page 114)]

Returns

(lower_bound, upper_bound) [*`LinExp`* (page 114)] Lower and upper bounds for the signal level in terms of the symbols of the form `LinExp("signal_LEVEL_ORIENT_min")` and `LinExp("signal_LEVEL_ORIENT_max")`, representing the minimum and maximum signal levels for transform coefficients in level `LEVEL` and orientation `ORIENT` respectively.

`evaluate_synthesis_filter_bounds`(*lower_bound_exp*, *upper_bound_exp*, *coeff_bounds*)
Convert a pair of symbolic synthesis filter bounds into concrete, numerical lower/upper bounds given the bounds of the input transform coefficients.

Parameters

lower_bound_exp [*`LinExp`* (page 114)]

upper_bound_exp [*`LinExp`* (page 114)] Synthesis filter bounds expressions from *`synthesis_filter_bounds()`* (page 88).

coeff_bounds [{(level, orient): (lower_bound, upper_bound), ...}] For each transform coefficient, the concrete lower and upper bounds (e.g. as computed using *`evaluate_analysis_filter_bounds()`* (page 87) and *`vc2_bit_widths.quantisation.maximum_dequantised_magnitude()`* (page 103) for each transform subband).

Returns

lower_bound [int]

upper_bound [int] The concrete (numerical) bounds for the synthesis filter.

14.2 Integer representation utilities

The following utility functions compute the relationship between bit-width and numerical range.

`twos_compliment_bits`(*value*)
How many bits does the provided integer require for a two's compliment (signed) integer representation?

`signed_integer_range`(*num_bits*)
Return the lower- and upper-bound values for a signed integer with the specified number of bits.

`unsigned_integer_range`(*num_bits*)
Return the lower- and upper-bound values for an unsigned integer with the specified number of bits.

The following function may be used to pessimistically round values to integers:

round_away_from_zero(*value*)
Round a number away from zero.

CHAPTER

FIFTEEN

VC2_BIT_WIDTHS.PATTERN_GENERATION: HEURISTIC TEST PATTERN GENERATION

The routines in this module implement heuristcs for producing test patterns for VC-2 filters which produce near-maximum magnitude values in a target intermediate or final value.

Due to non-linearities in VC-2's filters (i.e. rounding and quantisation), the test patterns generated are not true worst-case signals but rather a 'best effort' to get close to the worst case. Analysis test patterns will tend to be very close to worst-case signals while synthesis signals are likely to be modest under-estimates. Nevertheless, these signals often reach values well above real pictures and noise.

15.1 Test pattern generators

15.1.1 Analysis

make_analysis_maximising_pattern(*input_array*, *target_array*, *tx*, *ty*)

Create a test pattern which maximises a value within an intermediate/final output of an analysis filter.

Note: In lossless coding modes, test patterns which maximise a given value in the encoder also maximise the corresponding value in the decoder. Consequently this function may also be used to (indirectly) produce lossless decoder test patterns.

Warning: The returned test pattern is designed to maximise a real-valued implementation of the target filter. Though it is likely that this signal also maximises integer-based implementations (such as those used by VC-2) it is not guaranteed.

Parameters

input_array [*SymbolArray* (page 126)] The input array to the analysis filter.

target_array [*InfiniteArray* (page 125)] An intermediate or final output array produced by the analysis filter within which a value should be maximised.

tx, ty [int] The coordinates of the target value within target_array which is to be maximised.

Returns

test_pattern_specification [TestPatternSpecification]

15.1.2 Synthesis

make_synthesis_maximising_pattern(*analysis_input_array*, *analysis_transform_coeff_arrays*, *synthesis_target_array*, *synthesis_output_array*, *tx*, *ty*)

Create a test pattern which, after lossy encoding, is likely to maximise an intermediate/final value of the synthesis filter.

> **Warning:** Because (heavy) lossy VC-2 encoding is a non-linear process, finding encoder inputs which maximise the decoder output is not feasible in general. This function uses a simple heuristic (see *Test pattern generation* (page 63)) to attempt to achieve this goal but cannot provide any guarantees about the extent to which it succeeds.

Parameters

analysis_input_array [*SymbolArray* (page 126)] The input array to a compatible analysis filter for the synthesis filter whose values will be maximised.

analysis_transform_coeff_arrays [{level: {orient: *InfiniteArray* (page 125), ...}}] The final output arrays from a compatible analysis filter for the synthesis filter whose values will be maximised. These should be provided as a nested dictionary of the form produced by *analysis_transform()* (page 108).

synthesis_target_array [*InfiniteArray* (page 125)] The intermediate or final output array produced by the synthesis filter within which a value should be maximised.

synthesis_output_array [*InfiniteArray* (page 125)] The output array for the synthesis filter.

tx, ty [int] The index of the value within the synthesis_target_array which is to be maximised.

Returns

test_pattern_specification [TestPatternSpecification]

CHAPTER

SIXTEEN

VC2_BIT_WIDTHS.PATTERN_OPTIMISATION: (SYNTHESIS) TEST PATTERN OPTIMISATION

This module contains routines for optimising heuristic synthesis filter test patterns for a particular codec configuration (e.g. bit width and quantisation matrix).

16.1 Search algorithm & parameters

At a high-level the greedy search proceeds to optimise a test pattern in the following steps:

1. Make some random changes to the input pattern.
2. See if these changes made the decoded output value larger. If they did, keep the changes, if they didn't, discard them.
3. Go to step 1, unless no progress has been made for some time.

As an added detail, the whole search process may be repeated several times to reduce the impact of local minima encountered during a search.

The search is controlled by a number of parameters, explained below. These must be chosen appropriately for each codec configuration and good parameters for one transform may perform poorly with others. Only general advice is offered about the effect of each parameter, experimentation is essential.

16.1.1 Step 1: Random perturbations

Test patterns are perturbed as follows:

1. `removed_corruptions_per_iteration` pixels within the test pattern are selected at random (with replacement – i.e. fewer than `removed_corruptions_per_iteration` pixels may be chosen in some cases).
2. The chosen pixels are set back to their original state, before optimisation began.
3. `added_corruptions_per_iteration` pixels within the test pattern are selected at random (in the same way as above).
4. The chosen pixels are overwritten with a positive or negative polarity at random.

The `added_corruptions_per_iteration` parameter controls the exploratory tendency of the search. Setting this to a low value will make the search move more slowly but tend not less readily discard good features of a pattern discovered so far. Setting this to a high value will allow the search to more easily escape local minima but may require increased runtime to complete.

The `removed_corruptions_per_iteration` parameter can allow a search to 'undo' bad decisions, working on the basis that the starting pattern is more effective than random noise.

The effect of these two parameters varies widely between wavelets. Experimentation is recommended.

16.1.2 Step 2: Evaluate the perturbed test pattern

This step in the algorithm uses *FastPartialAnalyseQuantiseSynthesise* (page 100) to evaluate the test pattern's performance over a range of quantisation indices. This process exactly matches the integer arithmetic specified in the VC-2 standard.

This step is parameterised by the obvious wavelet, transform depth, quantisation matrix and picture signal range parameters.

The `max_quantisation_index` specifies the maximum quantisation index which will be used to evaluate a test pattern. This should be sufficiently high that at any higher level, all transform coefficients are quantised to zero. Setting this parameter too low may cause especially effective test patterns to be missed (the most effective patterns often work best at high quantisation indices). Setting this parameter too high simply wastes time. The *quantisation_index_bound()* (page 38) function may be used to determine the ideal value for this parameter.

16.1.3 Step 3: Repeat or terminate

If the perturbed test pattern was found to produce a larger signal value than both the unmodified test pattern and the best perturbed pattern found previously, the new test pattern will be accepted as the new starting point for subsequent searches. If no improvement was found, the perturbed pattern is discarded and the previous best pattern is used for the next search iteration.

To trigger the eventual termination of the search, a counter is used. This counter is initialised `base_iterations` parameter and is decremented by one after each search iteration. Whenever an improved test pattern is discovered, the counter is incremented by `added_iterations_per_improvement`. When the counter reaches zero, the search is terminated.

These parameters should be set experimentally according to the time available and wavelet used. Longer searches produce larger, though diminishing, improvements. The `added_iterations_per_improvement` parameter allows fruitful searches to continue for longer than searches which do not appear to be producing results.

16.1.4 Search repitition

The whole search process may be repeated, re-starting from the unmodified test pattern each time, to escape local minima encountered during a search. The best result found in any attempt will be returned as the final result.

The `number_of_searches` parameter may be used to set the number of times the whole search process should be repeated. It can sometimes be more productive to choose a larger `number_of_searches` with lower `base_iterations` and `added_iterations_per_improvement` over fewer but longer searchs.

For certain test patterns, random search may never manage to make any improvements. In these cases, the `terminate_early` parameter may be used to stop repeating the search process before `number_of_searches` if no search manages to find any improvement. When `base_iterations` is sufficiently large, this early termination behaviour should be unlikely to result in false negatives.

16.2 API

optimise_synthesis_maximising_test_pattern(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *quantisation_matrix*, *synthesis_pyexp*, *test_pattern_specification*, *input_min*, *input_max*, *max_quantisation_index*, *random_state*, *number_of_searches*, *terminate_early*, *added_corruptions_per_iteration*, *removed_corruptions_per_iteration*, *base_iterations*, *added_iterations_per_improvement*)

Optimise a test pattern generated by *make_synthesis_maximising_pattern()* (page 92) using repeated greedy stochastic search, attempting to produce larger signal values for the specified codec configuration.

The basic *make_synthesis_maximising_pattern()* (page 92) function simply uses a heuristic to produce a patterns likely to produce extreme values in the general case. By contrast this function attempts to more directly optimise the test pattern for particular input signal ranges and quantiser configurations.

Parameters

h_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)] Horizontal and vertical filter synthesis filter parameters (e.g. from vc2_data_tables.LIFTING_FILTERS ([vc2_data_tables], page 7)) defining the wavelet transform used.

dwt_depth [int]

dwt_depth_ho [int] The transform depth used by the filters.

quantisation_matrix [{level: {orient: int, ...}, ...}] The VC-2 quantisation matrix to use.

synthesis_pyexp [PyExp] A *PyExp* (page 119) expression which defines the synthesis process for the decoder value the test pattern is maximising/minimising. Such an expression is usually obtained from the use of *synthesis_transform()* (page 108) and *make_variable_coeff_arrays()* (page 109).

test_pattern_specification [TestPatternSpecification] The test pattern to optimise. This test pattern must be translated such that no analysis or synthesis step depends on VC-2's edge extension behaviour. This will be the case for test patterns produced by *make_synthesis_maximising_pattern()* (page 92).

input_min [int]

input_max [int] The minimum and maximum value which may be used in the test pattern.

max_quantisation_index [int] The maximum quantisation index to use. This should be set high enough that at the highest quantisation level all transform coefficients are quantised to zero.

random_state [numpy.random.RandomState[23]] The random number generator to use during the search.

number_of_searches [int] Repeat the greedy stochastic search process this many times. Since searches will tend to converge on local minima, increasing this parameter will tend to produce improved results.

terminate_early [None or int] If an integer, stop searching if the first terminate_early searches fail to find an improvement. If None, always performs all searches.

added_corruptions_per_iteration [int] The number of pixels to assign with a random value during each search attempt.

removed_corruptions_per_iteration [int] The number of pixels entries to reset to their original value during each search attempt.

base_iterations [int] The initial number of search iterations to perform.

added_iterations_per_improvement [int] The number of additional search iterations to perform whenever an improved pattern is found.

Returns

optimised_test_pattern [`OptimisedTestPatternSpecification`] The optimised test pattern found during the search.

[23] https://numpy.org/doc/stable/reference/random/legacy.html#numpy.random.RandomState

CHAPTER

SEVENTEEN

VC2_BIT_WIDTHS.FAST_PARTIAL_ANALYSIS_TRANSFORM: WAVELET ANALYSIS TRANSFORM

This module contains a numpy[24]-based implementation of a VC-2-like integer lifting wavelet analysis (encoding) transform. This implementation is approximately 100x faster than executing the equivalent VC-2 pseudocode under Python. It also optionally may be used to extract intermediate results from the codec.

This module is not indended as a general-purpose encoder implementation but rather for rapid evaluation of test patterns. Since test patterns are edge-effect free, this implementation does not implement VC-2's edge effect handling behaviour – hence the 'partial' part of the module name. Output values which would have been effected by edge effects will contain nonsense values.

Note: This Python+numpy filter is still extremely slow compared to any reasonable native software implementation. However, being pure Python, it is more portable and therefore useful in this application.

17.1 Example usage

The example below demonstrates how the *fast_partial_analysis_transform()* (page 98) function may be used to perform an analysis transform on an example picture:

```
>>> import numpy as np
>>> from vc2_data_tables import LIFTING_FILTERS, WaveletFilters
>>> from vc2_bit_widths.fast_partial_analysis_transform import (
...     fast_partial_analysis_transform,
... )

>>> # Codec parameters
>>> wavelet_index = WaveletFilters.le_gall_5_3
>>> wavelet_index_ho = WaveletFilters.le_gall_5_3
>>> dwt_depth = 2
>>> dwt_depth_ho = 0

>>> # A test picture
>>> width = 1024  # NB: Must be appropriate multiple for
>>> height = 512  # filter depth chosen!
>>> picture = np.random.randint(-512, 511, (height, width))

>>> h_filter_params = LIFTING_FILTERS[wavelet_index_ho]
>>> v_filter_params = LIFTING_FILTERS[wavelet_index]

>>> # Perform the analysis transform (in place)
>>> transform_coeffs = fast_partial_analysis_transform(
...     h_filter_params,
...     v_filter_params,
```

(continues on next page)

[24] https://numpy.org/doc/stable/reference/index.html#module-numpy

(continued from previous page)

```
...     dwt_depth,
...     dwt_depth_ho,
...     picture,
... )
```

17.2 API

fast_partial_analysis_transform(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *signal*, *target=None*)

Perform a multi-level 2D analysis transform, ignoring edge effects.

Parameters

h_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)] Horizontal and vertical filter *synthesis* (not analysis!) filter parameters (e.g. from vc2_data_tables.LIFTING_FILTERS ([vc2_data_tables], page 7)).

dwt_depth, dwt_depth_ho: int Transform depths for 2D and horizontal-only transforms.

signal [numpy.array] The picture to be transformed. Will be transformed in-place (in interleaved form) but de-interleaved views will also be returned.

Width must be a multiple of `2**(dwt_depth+dwt_depth_ho)` pixels and height a multiple of `2**dwt_depth` pixels.

target [(level, array_name, tx, ty) or None] If None, the complete analysis transform will be performed. If a tuple is given, the transform will run until the specified value has been computed and this value alone will be returned.

Returns

transform_coeffs [{level: {orient: numpy.array, ...}, ...} or intermediate_value] Subsampled views of the `signal` array (which will have been modified in-place).

Alternatively, if the `target` argument is not None, the single intermediate value requested will be returned.

CHAPTER

EIGHTEEN

VC2_BIT_WIDTHS.FAST_PARTIAL_ANALYSE_QUANTISE_SYNTHESISE: FAST SINGLE VALUE ENCODE, QUANTISE AND DECODE

This module implements a relatively efficient (for Python) implementation of an encode, quantise and decode cycle where only a single decoder output (or intermediate) value is computed.

18.1 Example usage

The example below demonstrates how the *FastPartialAnalyseQuantiseSynthesise()* (page 100) function may be used to compute the output of a VC-2 decoder intermediate value for a given picture:

```
>>> import numpy as np
>>> from vc2_data_tables import (
...     LIFTING_FILTERS,
...     WaveletFilters,
...     QUANTISATION_MATRICES,
... )
>>> from vc2_bit_widths.vc2_filters import (
...     synthesis_transform,
...     make_variable_coeff_arrays,
... )
>>> from vc2_bit_widths.fast_partial_analyse_quantise_synthesise import (
...     FastPartialAnalyseQuantiseSynthesise,
... )

>>> # Codec parameters
>>> wavelet_index = WaveletFilters.le_gall_5_3
>>> wavelet_index_ho = WaveletFilters.le_gall_5_3
>>> dwt_depth = 2
>>> dwt_depth_ho = 0

>>> # Quantisation settings
>>> quantisation_indices = list(range(64))
>>> quantisation_matrix = QUANTISATION_MATRICES[(
...     wavelet_index,
...     wavelet_index_ho,
...     dwt_depth,
...     dwt_depth_ho,
... )]

>>> # A test picture
>>> width = 1024  # NB: Must be appropriate multiple for
>>> height = 512  # filter depth chosen!
>>> picture = np.random.randint(-512, 511, (height, width))

>>> h_filter_params = LIFTING_FILTERS[wavelet_index_ho]
```

(continues on next page)

(continued from previous page)

```
>>> v_filter_params = LIFTING_FILTERS[wavelet_index]

>>> # Construct synthesis expression for target synthesis value
>>> level = 1
>>> array_name = "L"
>>> tx, ty = 200, 100
>>> _, intermediate_synthesis_arrays = synthesis_transform(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     make_variable_coeff_arrays(dwt_depth, dwt_depth_ho)
... )
>>> synthesis_exp = intermediate_synthesis_arrays[(level, array_name)][tx, ty]

>>> # Setup codec
>>> codec = FastPartialAnalyseQuantiseSynthesise(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     quantisation_matrix,
...     quantisation_indices,
...     synthesis_exp,
... )

>>> # Perform the analysis/quantise/synthesis transforms
>>> codec.analyse_quantise_synthesise(picture.copy())  # NB: corrupts array!
```

18.2 API

class FastPartialAnalyseQuantiseSynthesise(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *quantisation_matrix*, *quantisation_indices*, *synthesis_exp*)

An object which encodes a picture, quantises the transform coefficients at a range of different quantisation indices, and then decodes a single decoder output or intermediate value at each quantisation level.

For valid results to be produced by this class, both the analysis and synthesis processes must be completely edge-effect free. If this condition is not met, behaviour is undefined and may crash or produce invalid results.

Parameters

h_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)] Horizontal and vertical filter *synthesis* (not analysis!) filter parameters (e.g. from vc2_data_tables.LIFTING_FILTERS ([vc2_data_tables], page 7)) defining the wavelet transform types to use.

dwt_depth [int]

dwt_depth_ho [int] Transform depths for 2D and horizontal-only transforms.

quantisation_matrix [{level: {orient: int, ...}, ...}] The VC-2 quantisation matrix to use (e.g. from vc2_data_tables.QUANTISATION_MATRICES ([vc2_data_tables], page 8)).

quantisation_indices [[qi, ...]] A list of quantisation indices to use during the quantisation step.

synthesis_exp [*PyExp* (page 119)] A *PyExp* (page 119) object which describes the partial synthesis (decoding) process to be performed.

This expression will typically be an output of a *vc2_bit_widths.vc2_filters.synthesis_transform()* (page 108) which has been fed coefficient *Argument* (page 119)s produced by *vc2_bit_widths.vc2_filters.make_variable_coeff_arrays()* (page 109).

This expression must only rely on transform coefficients known to be edge-effect free in the encoder's output.

property quantisation_indices

The quantisation indices used during decoding.

analyse_quantise_synthesise(*picture*)

Encode, quantise (at multiple levels) and then decode the supplied picture. The decoded result at each quantisation level will be returned.

Parameters

picture [numpy.array] The picture to be encoded. Will be corrupted as a side effect of calling this method.

Width must be a multiple of `2**(dwt_depth+dwt_depth_ho)` pixels and height a multiple of `2**dwt_depth` pixels.

Dimensions must also be sufficient that all transform coefficients used by the `synthesis_exp` will be edge-effect free.

Returns

decoded_values [[value, ...]] The decoded value at for each quantisation index in `quantisation_indices`.

CHAPTER

NINETEEN

VC2_BIT_WIDTHS.QUANTISATION: VC-2 QUANTISATION

The *vc2_bit_widths.quantisation* (page 101) module contains an implementation of VC-2's quantisation scheme along with functions for analysing its properties.

19.1 Quantisation & dequantisation

The VC-2 quantisation related pseudocode functions are implemented as follows:

forward_quant(*coeff*, *quant_index*)
Quantise a value according to the informative description in (13.3.1)

inverse_quant(*quantized_coeff*, *quant_index*)
Dequantise a value using the normative method in (13.3.1)

quant_factor(*index*)
Compute the quantisation factor for a given quantisation index. (13.3.2)

quant_offset(*index*)
Compute the quantisation offset for a given quantisation index. (13.3.2)

19.2 Analysis

The following functions compute incidental information about the behaviour of the VC-2 quantisation scheme.

maximum_useful_quantisation_index(*value*)
Compute the smallest quantisation index which quantizes the supplied value to zero. This is considered to be the largest useful quantisation index with respect to this value.

maximum_dequantised_magnitude(*value*)
Find the value with the largest magnitude that the supplied value may be dequantised to for any quantisation index.

See *Quantisation and affine arithmetic* (page 57) for a proof of this method.

CHAPTER

TWENTY

VC2_BIT_WIDTHS.VC2_FILTERS: VC-2 FILTERS IMPLEMENTED AS INFINITEARRAYS

This module provides an implementation of the complete VC-2 wavelet filtering process in terms of *InfiniteArray* (page 125)s.

By using *SymbolArray* (page 126)s as inputs, algebraic descriptions (using *LinExp* (page 114)) of the VC-2 filters may be assembled. From these analyses may be performed to determine, for example, signal ranges and rounding error bounds.

Using *VariableArray* (page 126)s as inputs, Python functions may be generated (using *pyexp* (page 115)) which efficiently compute individual filter outputs or intermediate values in isolation.

20.1 Usage

To create an InfiniteArrays-based description of a VC-2 filter we must first define the filter to be implemented. In particular, we need a set of vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7) describing the wavelets to use. In practice these are easily obtained from vc2_data_tables.LIFTING_FILTERS ([vc2_data_tables], page 7) like so:

```
>>> from vc2_data_tables import WaveletFilters, LIFTING_FILTERS

>>> wavelet_index = WaveletFilters.haar_with_shift
>>> wavelet_index_ho = WaveletFilters.le_gall_5_3
>>> dwt_depth = 1
>>> dwt_depth_ho = 3

>>> h_filter_params = LIFTING_FILTERS[wavelet_index_ho]
>>> v_filter_params = LIFTING_FILTERS[wavelet_index]
```

Given this description we can construct a set of symbolic analysis filters using *analysis_transform()* (page 108):

```
>>> from vc2_bit_widths.infinite_arrays import SymbolArray
>>> from vc2_bit_widths.vc2_filters import analysis_transform

>>> input_picture = SymbolArray(2)
>>> output_coeff_arrays, intermediate_analysis_arrays = analysis_transform(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     input_picture,
... )
```

Two dictionaries are returned. The first dictionary, output_coeff_arrays, provides a nested dictionary of the form {level: {orient: array, ...}, ...} containing the InfiniteArrays representing the generated transform coefficients.

The second dictionary, `intermediate_analysis_arrays`, is of the form `{(level, array_name): array, ...}` and exposes every intermediate `InfiniteArray` from the filtering process (see *Terminology* (page 6) for a guide to the naming convention used). This dictionary contains a superset of the arrays contained in the first.

Similarly we can use *`synthesis_transform()`* (page 108) to construct an algebraic description of the synthesis filters. This function takes an array for each transform component as input. The *`make_symbol_coeff_arrays()`* (page 109) utility function provides a convenient way to produce the necessary *`SymbolArray`* (page 126)s:

```
>>> from vc2_bit_widths.vc2_filters import (
...     make_symbol_coeff_arrays,
...     synthesis_transform,
... )

>>> input_coeff_arrays = make_symbol_coeff_arrays(dwt_depth, dwt_depth_ho)
>>> output_picture, intermediate_synthesis_arrays = synthesis_transform(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     input_coeff_arrays,
... )
```

As before, two values are returned. The first, `output_picture`, is a `InfiniteArray` representing the final decoded picture. The second, `intermediate_synthesis_arrays`, again contains all of the intermediate `InfiniteArrays` (and the output picture).

Warning: The *`analysis_transform()`* (page 108) and *`synthesis_transform()`* (page 108) functions always return almost immediately since *`InfiniteArray`* (page 125)s only compute their values on-demand. For very large transforms, accessing values within these arrays (and triggering their evaluation) can take a non-trivial amount of time and memory.

20.2 Omitting arrays

Some of the intermediate arrays returned by *`analysis_transform()`* (page 108) and *`synthesis_transform()`* (page 108) are simple interleavings/subsamplings/renamings of other intermediate arrays. These arrays may be identified using their *`nop`* (page 125) property and skipped to avoid duplicating work when performing filter analysis.

When arrays have been skipped during processing it can still be helpful to show the duplicate entries when results are presented. The *`add_missing_analysis_values()`* (page 109) and *`add_missing_synthesis_values()`* (page 109) functions are provided to perform exactly this task.

For example, lets count up the number of symbols in each filter phase in the example wavelet transforms, skipping duplicate arrays:

```
>>> def count_symbols(expression):
...     return len(list(expression.symbols()))

>>> analysis_symbol_counts = {
...     (level, array_name, x, y): count_symbols(array[x, y])
...     for (level, array_name), array in intermediate_analysis_arrays.items()
...     for x in range(array.period[0])
...     for y in range(array.period[1])
...     if not array.nop
... }
>>> synthesis_symbol_counts = {
```

(continues on next page)

(continued from previous page)

```
...     (level, array_name, x, y): count_symbols(array[x, y])
...     for (level, array_name), array in intermediate_synthesis_arrays.items()
...     for x in range(array.period[0])
...     for y in range(array.period[1])
...     if not array.nop
... }
```

We can then fill in all of the missing entries and present the results to the user:

```
>>> from vc2_bit_widths.vc2_filters import (
...     add_missing_analysis_values,
...     add_missing_synthesis_values,
... )

>>> full_analysis_symbol_counts = add_missing_analysis_values(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     analysis_symbol_counts,
... )
>>> full_synthesis_symbol_counts = add_missing_synthesis_values(
...     h_filter_params,
...     v_filter_params,
...     dwt_depth,
...     dwt_depth_ho,
...     synthesis_symbol_counts,
... )

>>> for (level, array_name, x, y), symbol_count in full_analysis_symbol_counts.
↪items():
...     print("{} {} {} {}: {} symbols".format(
...         level, array_name, x, y, symbol_count
...     ))
4 Input 0 0: 1 symbols
4 DC 0 0: 1 symbols
4 DC' 0 0: 1 symbols
4 DC' 1 0: 4 symbols
<...snip...>
1 DC'' 0 0: 340 symbols
1 DC'' 1 0: 245 symbols
1 L 0 0: 340 symbols
1 H 0 0: 245 symbols

>>> for (level, array_name, x, y), symbol_count in full_synthesis_symbol_counts.
↪items():
...     print("{} {} {} {}: {} symbols".format(
...         level, array_name, x, y, symbol_count
...     ))
1 L 0 0: 1 symbols
1 H 0 0: 1 symbols
1 DC'' 0 0: 1 symbols
1 DC'' 1 0: 1 symbols
<...snip...>
4 Output 14 0: 34 symbols
4 Output 14 1: 38 symbols
4 Output 15 0: 42 symbols
4 Output 15 1: 48 symbols
```

20.3 API

20.3.1 Transforms

analysis_transform(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *array*)

Perform a multi-level VC-2 analysis Discrete Wavelet Transform (DWT) on a `InfiniteArray` in a manner which is the complement of the 'idwt' pseudocode function described in (15.4.1) in the VC-2 standard.

Parameters

h_filter_params, v_filter_params [`vc2_data_tables.LiftingFilterParameters` ([vc2_data_tables], page 7)] Horizontal and vertical filter parameters for the corresponding *synthesis* trhansform (e.g. from `vc2_data_tables.LIFTING_FILTERS` ([vc2_data_tables], page 7)). These filter parameters will be transformed into analysis lifting stages internally.

dwt_depth, dwt_depth_ho: int Transform depths for 2D and horizontal-only transforms.

array [`InfiniteArray`] The array representing the picture to be analysed.

Returns

coeff_arrays [{level: {orientation: `InfiniteArray`, ...}, ...}] The output transform coefficient values. These nested dictionaries are indexed the same way as 'coeff_data' in the idwt pseudocode function in (15.4.1) in the VC-2 specification.

intermediate_arrays [{(level, array_name): `InfiniteArray`, ...}] All intermediate (and output) value arrays, named according to the convention described in *Terminology* (page 6).

This value is returned as an `OrderedDict`[25] giving the arrays in their order of creation; a sensible order for display purposes.

synthesis_transform(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *coeff_arrays*)

Perform a multi-level VC-2 synthesis Inverse Discrete Wavelet Transform (IDWT) on a `InfiniteArray` in a manner equivalent to the 'idwt' pseudocode function in (15.4.1) of the VC-2 standard.

Parameters

h_filter_params, v_filter_params [`vc2_data_tables.LiftingFilterParameters` ([vc2_data_tables], page 7)] Horizontal and vertical filter synthesis filter parameters (e.g. from `vc2_data_tables.LIFTING_FILTERS` ([vc2_data_tables], page 7)).

dwt_depth, dwt_depth_ho: int Transform depths for 2D and horizontal-only transforms.

coeff_arrays [{level: {orientation: `InfiniteArray`, ...}, ...}] The transform coefficient arrays to be used for synthesis. These nested dictionaries are indexed the same way as 'coeff_data' in the idwt pseudocode function in (15.4.1) in the VC-2 specification. See *`make_symbol_coeff_arrays()`* (page 109) and *`make_variable_coeff_arrays()`* (page 109).

Returns

array [`InfiniteArray`] The final output array (i.e. decoded picture).

intermediate_arrays [{(level, array_name): `InfiniteArray`, ...}] All intermediate (and output) value arrays, named according to the convention described in *Terminology* (page 6).

This value is returned as an `OrderedDict`[26] giving the arrays in their order of creation; a sensible order for display purposes.

[25] https://docs.python.org/3/library/collections.html#collections.OrderedDict
[26] https://docs.python.org/3/library/collections.html#collections.OrderedDict

20.3.2 Coefficient array creation utilities

make_symbol_coeff_arrays(*dwt_depth*, *dwt_depth_ho*, *prefix='coeff'*)
Create a set of *SymbolArray* (page 126)s representing transform coefficient values, as expected by *synthesis_transform()* (page 108).

Returns

coeff_arrays [{level: {orientation: *SymbolArray* (page 126), ...}, ...}] The transform coefficient values. These dictionaries are indexed the same way as 'coeff_data' in the idwt pseudocode function in (15.4.1) in the VC-2 specification.

The symbols will have the naming convention `((prefix, level, orient), x, y)` where:

- prefix is given by the 'prefix' argument
- level is an integer giving the level number
- orient is the transform orientation (one of "L", "H", "LL", "LH", "HL" or "HH").
- x and y are the coordinate of the coefficient within that subband.

make_variable_coeff_arrays(*dwt_depth*, *dwt_depth_ho*, *exp=Argument('coeffs')*)
Create a set of *SymbolArray* (page 126)s representing transform coefficient values, as expected by *synthesis_transform()* (page 108).

Returns

coeff_arrays [{level: {orientation: *VariableArray* (page 126), ...}, ...}] The transform coefficient values. These dictionaries are indexed the same way as 'coeff_data' in the idwt pseudocode function in (15.4.1) in the VC-2 specification.

The expressions within the *VariableArray* (page 126)s will be indexed as follows:

```
>>> from vc2_bit_widths.pyexp import Argument

>>> coeffs_arg = Argument("coeffs_arg")
>>> coeff_arrays = make_variable_coeff_arrays(3, 0, coeffs_arg)
>>> coeff_arrays[2]["LH"][3, 4] == coeffs_arg[2]["LH"][3, 4]
True
```

20.3.3 Omitted value insertion

add_missing_analysis_values(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *analysis_values*)
Fill in results for omitted (duplicate) filter arrays and phases.

Parameters

h_filter_params, v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

dwt_depth, dwt_depth_ho: int The filter parameters.

analysis_values [{(level, array_name, x, y): value, ...}] A dictionary of values associated with individual intermediate analysis filter phases with entries omitted where arrays are just interleavings/subsamplings/renamings.

Returns

full_analysis_values [{(level, array_name, x, y): value, ...}] A new dictionary of values with missing filters and phases filled in.

add_missing_synthesis_values(*h_filter_params*, *v_filter_params*, *dwt_depth*, *dwt_depth_ho*, *synthesis_values*, *fill_in_equivalent_phases=True*)
Fill in results for omitted (duplicate) filter arrays and phases.

Parameters

h_filter_params, v_filter_params [vc2_data_tables.LiftingFilterParameters ([vc2_data_tables], page 7)]

dwt_depth, dwt_depth_ho: int The filter parameters.

synthesis_values [{(level, array_name, x, y): value, ...}] A dictionary of values associated with individual intermediate synthesis filter phases with entries omitted where arrays are just interleavings/subsamplings/renamings.

fill_in_equivalent_phases [bool] When two *InfiniteArray* (page 125)s with different periods are interleaved, the interleaved signal's period will include repetitions of some phases of one of the input arrays. For example, consider the following arrays:

```
 a = ... a0 a1 a0 a1 ...     (Period = 2)
 b = ... b0 b0 b0 b0 ...     (Period = 1)

ab = ... a0 b0 a1 b0 ...     (Period = 4)
```

In this example, the interleaved array has a period of 4 with two phases coming from a and two from b. Since b has a period of one, however, one of the phases of ab contains a repeat of one of the phases of 'b'.

Since in a de-duplicated set of filters and phases, duplicate phases appearing in interleaved arrays are not present, some other value must be used when filling in these phases. If the 'fill_in_equivalent_phases' argument is True (the default), the value from an equivalent phase will be copied in. If False, None will be used instead.

Where the dictionary being filled in contains results generic to the phase being used (and not the specific filter coordinates), the default value of 'True' will give the desired results.

Returns

full_synthesis_values [{(level, array_name, x, y): value, ...}] A new dictionary of values with missing filters and phases filled in.

CHAPTER

TWENTYONE

VC2_BIT_WIDTHS.LINEXP: A SIMPLE COMPUTER ALGEBRA SYSTEM WITH AFFINE ARITHMETIC

This module implements a Computer Algebra System (CAS) which is able to perform simple linear algebraic manipulations on linear expressions. In addition, a limited set of non-linear operations (such as truncating division) are supported and modelled using affine arithmetic.

Note: Compared with the mature and powerful `sympy` CAS, *linexp* (page 110) is extremely limited. However, within its restricted domain, *linexp* (page 110) is significantly more computationally efficient.

21.1 Linear expressions

Linear expressions are defined as being of the form:

$$k_1 v_1 + k_2 v_2 + \cdots + k_j$$

Where k_n are constants and v_n are free symbols (i.e. variables). For example, the following represents a linear expression:

$$a + 2b - 3c + 100$$

Where a, b and c are the free symbols and 1, 2, -3 and 100 are the relevant constants.

21.2 Usage

The expression above may be constructed using this library as follows:

```
>>> from vc2_bit_widths.linexp import LinExp

>>> # Create expressions containing just the symbols named 'a', 'b' and 'c'.
>>> a = LinExp("a")
>>> b = LinExp("b")
>>> c = LinExp("c")

>>> expr = a + 2*b - 3*c + 100
>>> print(expr)
a + 2*b + -3*c + 100
```

The *LinExp.subs()* (page 114) method may be used to substitute symbols with numbers:

```
>>> result = expr.subs({"a": 1000, "b": 10000, "c": 100000})
>>> result
LinExp(-278900)
```

When a *LinExp* (page 114) contains just a constant value, the constant may be extracted as a conventional Python number type using `constant`:

```
>>> result.is_constant
True
>>> result.constant
-278900
```

In a similar way, for *LinExp* (page 114) instances consisting of a single symbol with weight 1, the symbol name may be extracted like so:

```
>>> a.is_symbol
True
>>> a.symbol
'a'

>>> # Weighted symbols are considered different
>>> a3 = a * 3
>>> a3.is_symbol
False
```

Instances of *LinExp* (page 114) support all Python operators when the resulting expression would be strictly linear. For example:

```
>>> expr_times_ten = expr * 10
>>> print(expr_times_ten)
10*a + 20*b + -30*c + 1000

>>> three = (expr * 3) / expr
>>> print(three)
3

>>> # Not allowed since the result would not be linear
>>> expr / a
TypeError: unsupported operand type(s) for /: 'LinExp' and 'LinExp'
```

Expressions may use `Fraction`[27]s to represent coefficients exactly. Accordingly *LinExp* (page 114) implements division using `Fraction`[28]:

```
>>> expr_over_three = expr / 3
>>> print(expr_over_three)
(1/3)*a + (2/3)*b + -1*c + 100/3
```

Internally, linear expressions are represented by a dictionary of the form `{symbol: coefficient, ...}`. For example:

```
>>> repr(expr)
"LinExp({'a': 1, 'b': 2, 'c': -3, None: 100})"
```

Note that the constant term is stored in the special entry `None`.

The *LinExp* (page 114) class provides a number of methods for inspecting the value within an expression. For example, iterating over a *LinExp* (page 114) yields the `(symbol, coefficient)` pairs:

```
>>> list(expr)
[('a', 1), ('b', 2), ('c', -3), (None, 100)]
```

Alternatively, just the symbols can be listed:

[27] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[28] https://docs.python.org/3/library/fractions.html#fractions.Fraction

```
>>> list(expr.symbols())
['a', 'b', 'c', None]
```

The coefficients associated with particular symbols can be queried like so:

```
>>> expr["b"]
2
>>> expr["d"]
0
```

21.3 Affine Arithmetic

Affine arithmetic[29] may be used to bound the effects of non-linear operations such as integer truncation within a linear expression. In affine arithmetic, whenever a non-linear operation is performed, an error term is introduced which represents the range of values the non-linear operation could produce. In this way, the bounds of the effects of the non-linearity may be tracked.

For example, consider the following:

```
>>> a = LinExp("a")
>>> a_over_2 = a // 2
>>> print(a_over_2)
(1/2)*a + (1/2)*AAError(id=1) + -1/2
```

Here, the symbol "a" is divded by 2 with truncating integer division. The resulting expression starts with `(1/2)*a` as usual but is followed by an *AAError* (page 114) term and constant representing the rounding error.

In affine arithmetic, *AAError* (page 114) symbols represent unknown values in the range $[-1, +1]$. As a result we can see that our expression defines a range $[\frac{a}{2} - 1, \frac{a}{2}]$. This range represents the lower- and upper-bounds for the result of the truncating integer division. These bounds may be computed using the *affine_lower_bound()* (page 114) and *affine_upper_bound()* (page 114) functions respectively:

```
>>> from vc2_bit_widths.linexp import affine_lower_bound, affine_upper_bound

>>> print(affine_lower_bound(a_over_2))
(1/2)*a + -1
>>> print(affine_upper_bound(a_over_2))
(1/2)*a
```

Since affine arithmetic *AAError* (page 114) symbols are ordinary symbols they will be scaled and manipulated as in ordinary algebra. As such, the affine arithmetic bounds of an expression will remain correct. For example:

```
>>> expr = ((a // 2) * 2) + 100
>>> print(expr)
a + Error(id=1) + 99
>>> print(affine_lower_bound(expr))
a + 98
>>> print(affine_upper_bound(expr))
a + 100
```

As well as the inherent limitations of affine arithmetic, :py;class:*LinExp* is also naive in its implementation leading to additional sources of over-estimates. For example, in the following we might know that 'a' is an integer so the expected result should be just 'a', but *LinExp* (page 114) is unable to use this information:

```
>>> expr = (a * 2) // 2
>>> print(expr)
a + (1/2)*Error(id=4) + -1/2
```

[29] https://en.wikipedia.org/wiki/Affine_arithmetic

As a second example, every truncation produces a new *AAError* (page 114) term meaning that sometimes error terms do not cancel when they otherwise could:

```
>>> # Error terms cancel as expected
>>> print(a_over_2 - a_over_2)
0

>>> # Two different error terms don't cancel as expected
>>> print((a // 2) - (a // 2))
(1/2)*AAError(id=5) + (-1/2)*AAError(id=6)
```

21.4 API

class LinExp(*value=0*)

classmethod new_affine_error_symbol()
Create a *LinExp* (page 114) with a unique *AAError* (page 114) symbol.

symbols()
Return an iterator over the symbols in this *LinExp* (page 114) where the symbol `None` represents a constant term.

property is_constant
True iff this *LinExp* (page 114) represents only a constant value (including zero) and includes no symbols.

property constant
If *is_constant* (page 114) is True, contains the value as a normal numerical Python type. Otherwise raises `TypeError`[30] on access.

property is_symbol
True iff this *LinExp* (page 114) represents only single 1-weighted symbols.

property symbol
Iff this *LinExp* (page 114) contains only a single 1-weighted symbol (see *is_symbol* (page 114)) returns that symbol. Otherwise raises `TypeError`[31] on access.

subs(*substitutions*)
Substitute symbols in this linear expression for new values.

Parameters

substitutions [{symbol: number or symbol or *LinExp* (page 114), ...}] Substitutions to carry out. Substitutions will be carried out as if performed simultaneously.

Returns

linexp [*LinExp* (page 114)] A new expression once the substitution has been carried out.

class AAError(*id*)
An Affine Arithmetic error term. This symbol should be considered to represent an unknown value in the range $[-1, +1]$.

strip_affine_errors(*expression*)
Return the provided expression with all affine error symbols removed (i.e. set to 0).

affine_lower_bound(*expression*)
Calculate the lower-bound of an affine arithmetic expression.

[30] https://docs.python.org/3/library/exceptions.html#TypeError
[31] https://docs.python.org/3/library/exceptions.html#TypeError

affine_upper_bound(*expression*)
Calculate the upper-bound of an affine arithmetic expression.

affine_error_with_range(*lower*, *upper*)
Create an affine arithmetic expression defining the specified range.

CHAPTER

TWENTYTWO

VC2_BIT_WIDTHS.PYEXP: CONSTRUCT PYTHON PROGRAMS IMPLEMENTING ARITHMETIC EXPRESSIONS

This module provides the *PyExp* (page 119) class which tracks arithmetic operations performed on its instances. These objects may later be used to generate a Python function which implements these steps.

22.1 Motivation

When constructing test patterns it is often necessary to compute the output of a single filter output. Typically filters are described in terms of lifting steps which compute the filter output for an entire picture at once, even when only a single pixel or intermediate value is required. Using *PyExp* (page 119), it is possible to automatically extract a function which implements just the computations required to produce a single filter output. In this way, individual filter outputs may be efficiently computed in isolation.

22.2 Getting started

The `pyexp` module provides a series of subclasses of the abstract *PyExp* (page 119) base class. Together these subclasses may be used to define Python expressions which may be compiled into a function.

For example, using *Constant* (page 119) we can define a pair of constants and add them together:

```
>>> from vc2_bit_widths.pyexp import Constant

>>> five = Constant(5)
>>> nine = Constant(9)
>>> five_add_nine = five + nine

>>> print(five_add_nine)
BinaryOperator(Constant(5), Constant(9), '+')
```

Rather than immediately computing a result, adding the *Constant* (page 119)s together produced a new *PyExp* (page 119) (of type *BinaryOperator* (page 119)) representing the computation to be carried out. Using the `make_function()` method, we can create a Python function which actually evaluates the expression:

```
>>> compute_five_add_nine = five_add_nine.make_function()
>>> compute_five_add_nine()
14
```

The `generate_code()` method may be used to inspect the generated code. Here we can see that the generated function does actually perform the computation on demand:

```
>>> print(five_add_nine.generate_code())
def f():
    return (5 + 9)
```

To build more interesting functions we can define *Argument* (page 119)s. The names passed to the *Argument* (page 119) constructor become the argument names in the generated function:

```
>>> from vc2_bit_widths.pyexp import Argument

>>> a = Argument("a")
>>> b = Argument("b")
>>> average_expr = (a + b) / Constant(2.0)
>>> average = average_expr.make_function()

>>> average(a=10, b=15)
12.5
```

PyExp (page 119) instances support many Python operators and will automatically wrap non-*PyExp* (page 119) values in *Constant* (page 119). For example:

```
>>> array = Argument("array")
>>> average_of_first_and_last_expr = (array[0] + array[-1]) / 2.0
>>> average_of_first_and_last = average_of_first_and_last_expr.make_function()

>>> average_of_first_and_last([3, 4, 5, 6])
4.5
```

22.3 Carving functions out of existing code

Because *PyExp* (page 119) objects support the usual Python operators they can be passed as arguments to existing code. For example, consider the following function designed to operate on arrays of numbers:

```
>>> def multiply_by_place(array):
...     return [value * i for i, value in enumerate(array)]

>>> # Demo
>>> multiply_by_place([4, 3, 5, 11])
[0, 3, 10, 33]
```

Now, instead of passing in an array of numbers, lets pass in an array of *PyExp* (page 119) values:

```
>>> values = Argument("values")
>>> out = multiply_by_place([values[i] for i in range(4)])

>>> from pprint import pprint
>>> pprint(out)
[Constant(0),
 Subscript(Argument('values'), Constant(1)),
 BinaryOperator(Subscript(Argument('values'), Constant(2)), Constant(2), '*'),
 BinaryOperator(Subscript(Argument('values'), Constant(3)), Constant(3), '*')]
```

Our function now returns a list of *PyExp* (page 119) values. Using these we can generate functions which just compute one of the output values of `multiply_by_place` in isolation:

```
>>> compute_third_output = out[2].make_function()
>>> compute_third_output([4, 3, 5, 11])
10
```

We can verify that the generated function is genuinely computing only the required value (and not just computing everything and throwing most of it away) by inspecting its source code:

```
>>> print(out[2].generate_code())
def f(values):
    return (values[2] * 2)
```

22.4 Manipulating expressions

PyExp (page 119) supports simple manipulation of expressions in the form of the `subs()` method which substitutes subexpressions within an expression. For example:

```
>>> a = Argument("a")
>>> b = Argument("b")
>>> c = Argument("c")

>>> exp = a["deeply"]["nested"]["subscript"] + b
BinaryOperator(Subscript(Subscript(Subscript(Argument('a'), Constant('deeply')),␣
↪Constant('nested')), Constant('subscript')), Argument('b'), '+')

>>> exp2 = exp.subs({a["deeply"]["nested"]["subscript"]: c})
>>> print(exp2)
BinaryOperator(Argument('c'), Argument("b"), "+")
```

This functionality is intended to allow simple optimisations such as constant folding (substituting constants in place of variables) and replacing subscripted values with *Argument* (page 119)s to reduce overheads.

22.5 API

class PyExp
Abstract Base class for Python Expression objects.

class Constant(*value*)
A constant value.

Parameters

value [any] Must be serialised to a valid Python expression by `repr()`[32]

property value
The value of this constant.

class Argument(*name*)
An named argument which will be expected by the Python function implementing this expression.

Parameters

name [str] A valid Python argument name

property name
The name of this argument.

class Subscript(*exp*, *key*)
Subscript an expression, i.e. `exp[key]`.

Parameters

exp [*PyExp* (page 119)] The value to be subscripted.

key [*PyExp* (page 119)] The key to access.

property exp
The expression being subscripted.

property key
The subscript key.

class BinaryOperator(*lhs*, *rhs*, *operator*)
A binary operation applied to two *PyExp* (page 119)s, e.g. addition.

[32] https://docs.python.org/3/library/functions.html#repr

Parameters

lhs [*PyExp* (page 119)]

rhs [*PyExp* (page 119)] The arguments to the operator.

operator [str] The python operator symbol to apply (e.g. "+").

property lhs
The expression on the left-hand side of the operator.

property rhs
The expression on the right-hand side of the operator.

property operator
The operator, as a string.

CHAPTER

TWENTYTHREE

VC2_BIT_WIDTHS.INFINITE_ARRAYS: INFINITE ARRAYS

InfiniteArray (page 125) and its subclasses provide a way to define and analyse VC-2 filters.

23.1 Tutorial

In the worked example below we'll see how a simple LeGall (5, 3) synthesis filter can be described using *InfiniteArray* (page 125)s. We'll use this description to enumerate all of the resulting filter phases and generate algebraic expressions for them.

23.1.1 Building a VC-2 filter

A horizontal-only LeGall (5, 3) synthesis transform takes as input two arrays of transform coefficients (a low-pass band, `L`, and high-pass band, `H`) and produces a single output array. Lets start by defining the two input arrays:

```
>>> from vc2_bit_widths.infinite_arrays import SymbolArray

>>> L = SymbolArray(2, "L")
>>> H = SymbolArray(2, "H")
```

The *SymbolArray* (page 126) class defines an infinite array of *LinExp* (page 114)s containing a single symbol of the form `LinExp((prefix, x, y))`. Like all *InfiniteArray* (page 125) subclasses, we can access entries in our arrays using the usual Python subscript syntax:

```
>>> L[3, 4]
LinExp(('L', 3, 4))
>>> H[7, -5]
LinExp(('H', 7, -5))
```

Notice that the coordinate system extends to positive and negative infinity.

Next we'll create an *InterleavedArray* (page 127) which contains a horizontal interleaving of the two inputs:

```
>>> from vc2_bit_widths.infinite_arrays import InterleavedArray

>>> interleaved = InterleavedArray(L, H, 0)
```

The new interleaved array can be accessed just as before:

```
>>> interleaved[0, 0]
LinExp(('L', 0, 0))
>>> interleaved[1, 0]
LinExp(('H', 0, 0))
>>> interleaved[2, 1]
LinExp(('L', 1, 1))
>>> interleaved[3, 1]
LinExp(('H', 1, 1))
```

Next we'll apply the two lifting stages which implement the LeGall (5, 3) transform using a *LiftedArray* (page 126):

```
>>> from vc2_bit_widths.infinite_arrays import LiftedArray

>>> from vc2_data_tables import LIFTING_FILTERS, WaveletFilters
>>> wavelet = LIFTING_FILTERS[WaveletFilters.le_gall_5_3]
>>> stage_0, stage_1 = wavelet.stages

>>> lifted_once = LiftedArray(interleaved, stage_0, 0)
>>> lifted_twice = LiftedArray(lifted_once, stage_1, 0)
```

Finally we'll use a *RightShiftedArray* (page 126) to apply the final bit shift operation to the result:

```
>>> from vc2_bit_widths.infinite_arrays import RightShiftedArray

>>> output = RightShiftedArray(lifted_twice, wavelet.filter_bit_shift)
```

This final array now, in effect, contains a complete algebraic description of how each entry is computed in terms of our two input arrays, `L` and `H`:

```
>>> output[0, 0]
LinExp({('L', 0, 0): Fraction(1, 2), ('H', -1, 0): Fraction(-1, 8), ('H', 0, 0):␣
↪Fraction(-1, 8), AAError(id=1): Fraction(-1, 4), AAError(id=2): Fraction(1, 2)})
```

Notice that the expression above refers to transform coefficients with negative coordinates (e.g. `('H', -1, 0)`). This is because the infinite dimensions of *InfiniteArray* (page 125)s allows *LiftedArray* (page 126) to ignore VC-2's filter edge effect behaviour. This makes it easier to analyse filter behaviours without having to carefully avoid edge effected behaviour.

23.1.2 Filter phases

The *InfiniteArray* (page 125) subclasses automatically keep track of the period of each array (see *Terminology* (page 6)) in the *InfiniteArray.period* (page 125) property:

```
>>> L.period
(1, 1)
>>> H.period
(1, 1)

>>> output.period
(2, 1)
```

This makes it easy to automatically obtain an example of each filter phase in our output:

```
>>> all_output_filter_phases = [
...     output[x, y]
...     for x in range(output.period[0])
...     for y in range(output.period[1])
... ]
```

23.1.3 Detecting no-operations

When performing a computationally expensive analysis on a set of filters described by *InfiniteArray* (page 125)s it can be helpful to skip arrays which just consist of interleavings and other non-transformative views of other arrays. For this purpose, the *InfiniteArray.nop* (page 125) property indicates if a given array implements a no-operation (nop) and therefore may be skipped.

From the example above:

```
>>> L.nop
False
>>> H.nop
False

>>> interleaved.nop
True

>>> lifted_once.nop
False
>>> lifted_twice.nop
False

>>> output.nop
False
```

Here we can see that the interleaving stage is identified as having no material effect on the values it contains.

23.1.4 Relative array stepping

InfiniteArray (page 125)s provide a *InfiniteArray.relative_step_size_to()* (page 125) method for calculating the scaling relationships between arrays. This can be useful for finding array values which don't use inputs with negative coordinates (i.e. would not be affected by edge-effect behaviour in a real VC-2 filter).

For example, consider the top-left pixel of the filter output from before:

```
>>> output[0, 0]
LinExp({('L', 0, 0): Fraction(1, 2), ('H', -1, 0): Fraction(-1, 8), ('H', 0, 0):␣
↪Fraction(-1, 8), AAError(id=1): Fraction(-1, 4), AAError(id=2): Fraction(1, 2)})
```

This uses the input `('H', -1, 0)` which has negative coordinates. Thanks to the *period* (page 125) property we know that any array value with an even-numbered X coordinate implements the same filter phase as [0, 0].

We could start trying every even-numbered X coordinate until we find an entry without a negative `H` coordinate, but this would be fairly inefficient. Instead lets use *relative_step_size_to()* (page 125) to work out how many steps we must move in `output` to move all our `H` coordinates right by one step.

```
>>> output.relative_step_size_to(H)
(Fraction(1, 2), Fraction(1, 1))
```

This tells us that for every step we move in the X-dimension of `output`, we move half a step in that direction in `H`. Therefore, we must pick a coordinate at least two steps to the right in `output` to avoid the -1 coordinate in `H`. Moving to [2, 0] also gives an even numbered X coordinate so we know that we're going to see the same filter phase and so we get:

```
>>> output[2, 0]
LinExp({('L', 1, 0): Fraction(1, 2), ('H', 0, 0): Fraction(-1, 8), ('H', 1, 0):␣
↪Fraction(-1, 8), AAError(id=3): Fraction(-1, 4), AAError(id=6): Fraction(1, 2)})
```

Which implements the same filter phase as `output[0, 0]` but without any negative coordinates.

23.1.5 Caching

All of the subclasses of *InfiniteArray* (page 125) which perform a non-trivial operation internally cache array values when they're accessed.

The most obvious benefit of this behaviour is to impart a substantial performance improvement for larger filter designs.

A secondary benefit applies to *InfiniteArray* (page 125)s containing *LinExp* (page 114)s. By caching the results of operations which introduce affine arithmetic error symbols (*AAError* (page 114)), these errors can correctly combine or cancel when that result is reused. As a result, while caching is not essential for correctness, it can materially improve the tightness of the bounds produced.

In some instances, this basic caching behaviour may not go far enough. For example, the contents of `output[0, 0]` and `output[2, 0]` are extremely similar, consisting of essentially the same coefficients, just translated to the right:

```
>>> output[0, 0]
LinExp({('L', 0, 0): Fraction(1, 2), ('H', -1, 0): Fraction(-1, 8), ('H', 0, 0):␣
↪Fraction(-1, 8), AAError(id=1): Fraction(-1, 4), AAError(id=2): Fraction(1, 2)})
>>> output[2, 0]
LinExp({('L', 1, 0): Fraction(1, 2), ('H', 0, 0): Fraction(-1, 8), ('H', 1, 0):␣
↪Fraction(-1, 8), AAError(id=3): Fraction(-1, 4), AAError(id=6): Fraction(1, 2)})
```

In principle, the cached result of `output[0, 0]` could be re-used (and the coefficients suitably translated) to save the computational cost of evaluating `output[2, 0]` from scratch. For extremely large transforms, this optimisation can result in substantial savings in runtime and RAM. For use in such scenarios the *SymbolicPeriodicCachingArray* (page 127) array type is provided:

```
>>> from vc2_bit_widths.infinite_arrays import SymbolicPeriodicCachingArray

>>> output_cached = SymbolicPeriodicCachingArray(output, L, H)
```

The constructor takes the array to be cached along with all *SymbolArray* (page 126)s used its definition. The new array may be accessed as usual but with more aggressive caching taking place internally:

```
>>> output_cached[0, 0]
LinExp({('L', 0, 0): Fraction(1, 2), ('H', -1, 0): Fraction(-1, 8), ('H', 0, 0):␣
↪Fraction(-1, 8), AAError(id=1): Fraction(-1, 4), AAError(id=2): Fraction(1, 2)})
>>> output_cached[2, 0]
LinExp({('L', 1, 0): Fraction(1, 2), ('H', 0, 0): Fraction(-1, 8), ('H', 1, 0):␣
↪Fraction(-1, 8), AAError(id=1): Fraction(-1, 4), AAError(id=2): Fraction(1, 2)})
```

Warning: Note that in the cached version of the array, the *AAError* (page 114) terms are not unique between `output_cached[0, 0]` and `output_cached[2, 0]`, though they should be. This is a result of the caching mechanism having no way to know how error terms should change between entries in the array. As a result, *SymbolicPeriodicCachingArray* (page 127) must not be used when the affine error terms in its output are expected to be significant.

Note: *SymbolicPeriodicCachingArray* (page 127) only works with *InfiniteArray* (page 125)s defined in terms of *SymbolArray* (page 126)s.

23.2 API

23.2.1 `InfiniteArray` (base class)

class InfiniteArray(*ndim*, *cache*)

An abstract base class describing an immutable infinite N-dimensional array of symbolic values.

Subclasses should implement *get()* (page 125) to return the value at a given position in an array.

Instances of this type may be indexed like an N-dimensional array.

The 'cache' argument controls whether array values are cached or not. It is recommended that this argument be set to 'True' for expensive to compute functions or functions which introduce new error terms (to ensure error terms are re-used)

get(*keys*)

Called by `__getitem__()` with a tuple of array indices.

The array of keys is guaranteed to be a tuple with *ndim* (page 125) entries.

Values returned by this method will be memoized (cached) by `__getitem__()`. Consequently, this method will only be called the first time a particular array index is requested. Subsequent accesses will return the cached value.

clear_cache()

Clear the cache (if enabled).

property ndim

The number of dimensions in the array.

property period

Return the period of this array (see *Terminology* (page 6)).

Returns

period [(int, ...)] The period of the array in each dimension.

property nop

True if this *InfiniteArray* (page 125) is a no-operation (nop), i.e. it just views values in other arrays. False if some computation is performed.

relative_step_size_to(*other*)

For a step along a dimension in this array, compute the equivalent step size in the provided array.

Parameters

other [*InfiniteArray* (page 125)] An array to compare the step size with. Must have been used (maybe indirectly) to define this array.

Returns

relative_step_size [(`fractions.Fraction`[33], ...) or None] The relative step sizes for each dimension, or None if the provided array was not used in the computation of this array.

[33] https://docs.python.org/3/library/fractions.html#fractions.Fraction

23.2.2 Base value type arrays

class SymbolArray(*ndim*, *prefix='v'*)
An infinite array of *LinExp* (page 114) symbols.

Symbols will be identified by tuples like `(prefix, n)` for a one dimensional array, `(prefix, n, n)` for a two-dimensional array, `(prefix, n, n, n)` for a three-dimensional array and so-on.

Example usage:

```
>>> a = SymbolArray(3, "foo")
>>> a[1, 2, 3]
LinExp(('foo', 1, 2, 3))
>>> a[100, -5, 0]
LinExp(('foo', 100, -5, 0))
```

Parameters

ndim [int] The number of dimensions in the array.

prefix [object] A prefix to be used as the first element of every symbol tuple.

class VariableArray(*ndim*, *exp*)
An infinite array of subscripted *PyExp* (page 119) expressions.

Example usage:

```
>>> from vc2_bit_widths.pyexp import Argument

>>> arg = Argument("arg")
>>> a = VariableArray(2, arg)
>>> a[1, 2]
Subscript(Argument('arg'), Constant((1, 2)))
>>> a[-10, 3]
Subscript(Argument('arg'), Constant((-10, 3)))
```

Parameters

ndim [int] The number of dimensions in the array.

exp [*PyExp* (page 119)] The expression to be subscripted in each array element.

23.2.3 Computation arrays

class LiftedArray(*input_array*, *stage*, *filter_dimension*)
Apply a one-dimensional lifting filter step to an array, as described in the VC-2 specification (15.4.4).

Parameters

input_array [*InfiniteArray* (page 125)] The input array whose entries will be filtered by the specified lifting stage.

stage [`vc2_data_tables.LiftingStage` ([vc2_data_tables], page 7)] A description of the lifting stage.

interleave_dimension: int The dimension along which the filter will act.

class LeftShiftedArray(*input_array*, *shift_bits=1*)
Apply a left bit shift to every value in an input array.

Parameters

input_array [*InfiniteArray* (page 125)] The array to have its values left-shifted.

shift_bits: int Number of bits to shift by.

class RightShiftedArray(*input_array*, *shift_bits=1*)
Apply a right bit shift to every value in an input array.

The right-shift operation is based on the description in the VC-2 specification (15.4.3). Specifically, $2^{\text{shiftbits}-1}$ is added to the input values prior to the right-shift operation (which is used to implement rounding behaviour in integer arithmetic).

Parameters

input_array [*InfiniteArray* (page 125)] The array to have its values right-sifted

shift_bits: int Number of bits to shift by

23.2.4 Sampling arrays

class SubsampledArray(*input_array*, *steps*, *offsets*)
A subsampled view of another *InfiniteArray* (page 125).

Parameters

input_array [*InfiniteArray* (page 125)] The array to be subsampled.

steps, offsets: (int, ...) Tuples giving the step size and start offset for each dimension.

When this array is indexed, the index into the input array is computed as:

```
input_array_index[n] = (index[n] * step[n]) + offset[n]
```

class InterleavedArray(*array_even*, *array_odd*, *interleave_dimension*)
An array view which interleaves two *InfiniteArray* (page 125)s together into a single array.

Parameters

array_even, array_odd [*InfiniteArray* (page 125)] The two arrays to be interleaved. 'array_even' will be used for even-indexed values in the specified dimension and 'array_odd' for odd.

interleave_dimension: int The dimension along which the two arrays will be interleaved.

23.2.5 Caching arrays

class SymbolicPeriodicCachingArray(*array*, **symbol_arrays*)
A caching view of a *InfiniteArray* (page 125) of *LinExp* (page 114) values.

This view will request at most one value from each filter phase of the input array and compute all other values by altering the indices in the previously computed values.

For example, normally, when accessing values within a *InfiniteArray* (page 125), values are computed from scratch on-demand:

```
>>> s = SymbolArray(2, "s")
>>> stage = <some filter stage>
>>> la = LiftedArray(s, stage, 0)
>>> la[0, 0]  # la[0, 0] worked out from scratch
LinExp(...)
>>> la[0, 1]  # la[0, 1] worked out from scratch
LinExp(...)
>>> la[0, 2]  # la[0, 2] worked out from scratch
LinExp(...)
>>> la[0, 3]  # la[0, 3] worked out from scratch
LinExp(...)
```

By contrast, when values are accessed in a *SymbolicPeriodicCachingArray* (page 127), only the first access to each filter phase is computed from scratch. All other values are found by changing the symbol indices in the cached array value:

```
>>> cached_la = SymbolicPeriodicCachingArray(la, s)
>>> cached_la[0, 0]  # la[0, 0] worked out from scratch
LinExp(...)
>>> cached_la[0, 1]  # la[0, 1] worked out from scratch
LinExp(...)
>>> cached_la[0, 2]  # Cached value of la[0, 0] reused and mutated
LinExp(...)
>>> cached_la[0, 3]  # Cached value of la[0, 1] reused and mutated
LinExp(...)
```

Parameters

array [*InfiniteArray* (page 125)] The array whose values are to be cached. These array values must consist only of symbols from *SymbolArray* (page 126)s passed as arguments, *AAError* (page 114) terms and constants.

Warning: Error terms may be repeated as returned from this array where they would usually be unique. If this is a problem, you should not use this caching view.

For example:

```
>>> s = SymbolArray(2, "s")
>>> sa = RightShiftedArray(s, 1)

>>> # Unique AAError terms (without caching)
>>> sa[0, 0]
LinExp({('s', 0, 0): Fraction(1, 2), AAError(id=1):␣
↪Fraction(1, 2)})
>>> sa[0, 1]
LinExp({('s', 0, 1): Fraction(1, 2), AAError(id=2):␣
↪Fraction(1, 2)})

>>> # Non-unique AAError terms after caching
>>> ca = SymbolicPeriodicCachingArray(sa, s)
>>> ca[0, 0]
LinExp({('s', 0, 0): Fraction(1, 2), AAError(id=1):␣
↪Fraction(1, 2)})
>>> ca[0, 1]
LinExp({('s', 0, 1): Fraction(1, 2), AAError(id=1):␣
↪Fraction(1, 2)})
```

symbol_arrays [*SymbolArray* (page 126)] The symbol arrays which the 'array' argument is constructed from.

BIBLIOGRAPHY

[vc2_data_tables] The vc2_data_tables[5] manual.

[BaFT11] Eric J. Balster; Benjamin T. Fortener; William F. Turri: "Integer Computation of Lossy JPEG2000 Compression". In: IEEE Transactions on Image Processing, August 2011.

[FaGa08] Wenbing Fan; Yingmin Gao: "FPGA Design of Fast Lifting Wavelet Transform". In: Proceedings of the Congress on Image and Signal Processing, 2008.

[SZAA02] Vassilis Spiliotopoulos; N. D. Zcrvas; C. E. Androulidakis; G. Anagnostopoulos; S. Thcoharis: "Quantizing the 9/7 Daubechies filter coefficients for 2D DWT VLSI implementations". In: Proceedings of the International Conference on Digital Signal Processing, 2002.

[BRGT06] Stephen Bishop; Suresh Rai; B. Gunturk; J. Trahan; R. Vaidyanathan: "Reconfigurable Implementation of Wavelet Integer Lifting Transforms for Image Compression". In: Proceedings of IEEE International Conference on Reconfigurable Computing and FPGAs, 2006.

[HTLS08] Chin-Fa Hsieh; Tsung-Han Tsai; Chih-Hung Lai; Tai-An Shan: "A Novel Efficient VLSI Architecture of 2-D Discrete Wavelet Transform". In: Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2008.

[MoBj10] Leonardo de Moura; Nikolaj Bjørner: "Applications and Challenges in Satisfiability Modulo Theories". In: Proceedings of the Workshop on Invariant Generation, 2010.

[KiNi10] Adam B. Kinsman; Nicola Nicolici: "Bit-Width Allocation for Hardware Accelerators for Scientific Computing Using SAT-Modulo Theory". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, March 2010.

[LoCN07] Juan A. López; Carlos Carreras; Octavio Nieto-Taladriz: "Improved Interval-Based Characterization ofFixed-Point LTI Systems With Feedback Loops". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, November 2007.

[5] https://github.com/bbc/vc2_data_tables/

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V