Pseudocode for Riesz Pyramids for Fast Phase-Based Video Magnification

Neal Wadhwa, Michael Rubinstein, Frédo Durand and William T. Freeman

This document contains pseudocode for the 2014 ICCP paper Riesz Pyramids for Fast Phase-Based Video Magnification [3], which presents a real-time algorithm to magnify tiny motions in videos using a new image representation: the Riesz pyramid. The pseudocode uses the quaternion formulation of the Riesz pyramid described in our technical report [2]. The algorithm amplifies tiny motions in a temporal band of interest by amplifying variations in the temporally filtered quaternionic phase of every Riesz pyramid coefficient. Pseudocode for the main function plus some helper functions is included below. Please refer to the technical report for more mathematical justification [2] and refer to Oppenheim and Schafer for more information on the temporal filters used in this pseudocode [1].

Notation The notation in this pseudocode is based on MATLAB's syntax. All variables are either two dimensional images (possibly of size 1×1) or cell arrays: lists that can contain arbitrary elements. Indexing into an image is denoted by $[\cdot,\cdot]$ and indexing into a cell array is denoted by $\{\cdot\}$. A dot (.) preceeding a operator like multiplication ($*$) or exponentiation (ˆ) denotes that the operation is performed element-wise.

In the pseudocode below, we try to use descriptive variable names. However, for variables corresponding to filtered and unfiltered versions of the quaternionic phase

$$\phi\cos(\theta), \phi\sin(\theta), \tag{1}$$

this results in overly long variable names. For brevity, we represent $\cos(\theta)$ by only the word `cos` and $\sin(\theta)$ by only the word `sin`. That is, `phase_cos` represents $\phi\cos(\theta)$, not $\cos(\phi)$.

```
OnlineRieszVideoMagnification(amplification_factor, low_cutoff, high_cutoff, sampling_rate)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Initializes spatial smoothing kernel and temporal filtering
    % coefficients.

    % Compute an IIR temporal filter coefficients. Butterworth filter could be replaced
    % with any IIR temporal filter. Lower temporal_filter_order is faster
    % and uses less memory, but is less accurate. See pages 493-532 of
    % Oppenheim and Schafer 3rd ed for more information
    nyquist_frequency = sampling_rate/2;
    temporal_filter_order = 1;
    [B, A] = GetButterworthFilterCoefficients(temporal_filter_order, ...
                                              low_cutoff/nyquist_frequency, ...
                                              high_cutoff/nyquist_frequency);

    % Computes convolution kernel for spatial blurring kernel used during
    % quaternionic phase denoising step.
    gaussian_kernel_sd = 2; % px
    gaussian_kernel = GetGaussianKernel(gaussian_kernel_sd);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialization of variables before main loop.
% This initialization is equivalent to assuming the motions are zero
% before the video starts.
previous_frame = GetFirstFrameFromVideo();
[previous_laplacian_pyramid, previous_riesz_x, previous_riesz_y] = ...
     ComputeRieszPyramid(previous_frame);
number_of_levels = numel(previous_laplacian_pyramid) - 1; % Do not include lowpass residual
for k = 1:number_of_levels
   % Initializes current value of quaternionic phase. Each coefficient
   % has a two element quaternionic phase that is defined as
   % phase times (cos(orientation), sin(orientation))
   % It is initialized at zero
   phase_cos{k} = zeros(size(previous_laplacian_pyramid{k}));
   phase_sin{k} = zeros(size(previous_laplacian_pyramid{k}));


   % Initializes IIR temporal filter values. These values are used during
   % temporal filtering. See the function IIRTemporalFilter for more
   % details. The initialization is a zero motion boundary condition
   % at the beginning of the video.
   register0_cos{k} = zeros(size(previous_laplacian_pyramid{k}));
   register1_cos{k} = zeros(size(previous_laplacian_pyramid{k}));

   register0_sin{k} = zeros(size(previous_laplacian_pyramid{k}));
   register1_sin{k} = zeros(size(previous_laplacian_pyramid{k}));
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main loop. It is executed on new frames from the video and runs until
% stopped.
while running
    current_frame = GetNextFrameFromVideo();
    [current_laplacian_pyramid, current_riesz_x, current_riesz_y] = ...
        ComputeRieszPyramid(current_frame);

    % We compute a Laplacian pyramid of the motion magnified frame first and then
    % collapse it at the end.
    % The processing in the following loop is processed on each level
    % of the Riesz pyramid independently
    for k = 1:number_of_levels

        % Compute quaternionic phase difference between current Riesz pyramid
        % coefficients and previous Riesz pyramid coefficients.
        [phase_difference_cos, phase_difference_sin, amplitude] = ...
            ComputePhaseDifferenceAndAmplitude(current_laplacian_pyramid{k}, ...
                                               current_riesz_x{k}, ...
                                               current_riesz_y{k}, ...
                                               previous_laplacian_pyramid{k}, ...
                                               previous_riesz_x{k}, ...
                                               previous_riesz_y{k});


        % Adds the quaternionic phase difference to the current value of the quaternionic
        % phase.
        % Computing the current value of the phase in this way is
        % equivalent to phase unwrapping.
        phase_cos{k} = phase_cos{k} + phase_difference_cos;
        phase_sin{k} = phase_sin{k} + phase_difference_sin;


        % Temporally filter the quaternionic phase using current value and stored
        % information
```

```
            [phase_filtered_cos, register0_cos{k}, register1_cos{k}] = ...
                IIRTemporalFilter(B, A, phase_cos{k}, register0_cos{k}, register1_cos{k});
            [phase_filtered_sin, register0_sin{k}, register1_sin{k}] = ...
                IIRTemporalFilter(B, A, phase_sin{k}, register0_sin{k}, register1_sin{k});


            % Spatial blur the temporally filtered quaternionic phase signals.
            % This is not an optional step. In addition to denoising,
            % it smooths out errors made during the various approximations.
            phase_filtered_cos = ...
                AmplitudeWeightedBlur(phase_filtered_cos, amplitude, gaussian_kernel);
            phase_filtered_sin = ...
                AmplitudeWeightedBlur(phase_filtered_sin, amplitude, gaussian_kernel);


            % The motion magnified pyramid is computed by phase shifting
            % the input pyramid by the spatio-temporally filtered quaternionic phase and
            % taking the real part.
            phase_magnified_filtered_cos = amplification_factor * phase_filtered_cos;
            phase_magnified_filtered_sin = amplification_factor * phase_filtered_sin;

            motion_magnified_laplacian_pyramid{k} = ...
                PhaseShiftCoefficientRealPart(current_laplacian_pyramid{k}, ...
                                              current_riesz_x{k}, ...
                                              current_riesz_y{k}, ...
                                              phase_magnified_filtered_cos, ...
                                              phase_magnified_filtered_sin);
        end


        % Take lowpass residual from current frame's lowpass residual
        % and collapse pyramid.
        motion_magnified_laplacian_pyramid{number_of_levels+1} = ...
            current_laplacian_pyramid{number_of_levels+1};
        motion_magnified_frame = CollapseLaplacianPyramid(motion_magnified_laplacian_pyramid);


        % Write or display the motion magnified frame.
        WriteMagnifiedFrame(motion_magnified_frame);
        % DisplayMagnifiedFrame(motion_magnified_frame);


        % Prepare for next iteration of loop
        previous_laplacian_pyramid = current_laplacian_pyramid;
        previous_riesz_x = current_riesz_x;
        previous_riesz_y = current_riesz_y;
    end
```

Helper Functions Pseudocode for helper functions are provide below. They include information on how to build a Riesz pyramid, compute quaternionic phase, phase shift Riesz pyramid coefficients, temporally filtering phase and spatially blurring phase. Pseudocode for functions that compute and collapse Laplacian pyramids, read and write to videos and display images on a screen is *not* included.

```
ComputeRieszPyramid(grayscale_frame)
    % Compute Riesz pyramid of two dimensional frame. This is done by first
    % computing the laplacian pyramid of the frame and then computing the
    % approximate Riesz transform of each level that is not the lowpass
    % residual. The result is stored as an array of grayscale frames.
    % Corresponding locations in the result correspond to the real,
    % i and j components of Riesz pyramid coefficients.
    laplacian_pyramid = ComputeLaplacianPyramid(grayscale_frame);
    number_of_levels = numel(laplacian_pyramid)-1;

```

```
    % The approximate Riesz transform of each level that is not the
    % low pass residual is computed. For more details on the approximation,
    % see supplemental material.
    kernel_x = [0.0  0.0  0.0;
                0.5  0.0 -0.5;
                0.0  0.0  0.0];
    kernel_y = [0.0  0.5  0.0;
                0.0  0.0  0.0;
                0.0 -0.5  0.0];
    for k = 1:number_of_levels
        riesz_x{k} = Convolve(laplacian_pyramid{k}, kernel_x);
        riesz_y{k} = Convolve(laplacian_pyramid{k}, kernel_y);
    end
    return {laplacian_pyramid, riesz_x, riesz_y}
```

```
ComputePhaseDifferenceAndAmplitude(current_real, current_x, current_y, ...
                                   previous_real, previous_x, previous_y)
    % Computes quaternionic phase difference between current frame and previous
    % frame. This is done by dividing the coefficients of the current frame
    % and the previous frame and then taking imaginary part of  the quaternionic
    % logarithm. We assume the orientation at a point is roughly constant to
    % simplify the calcuation.

    % q_current = current_real + i * current_x + j * current_y
    % q_previous = previous_real + i * previous_x + j * previous_y
    % We want to compute the phase difference, which is the phase of
    %    q_current/q_previous
    % This is equal to (Eq. 10 of tech. report)
    %    q_current * conjugate(q_previous)/||q_previous||^2
    % Phase is invariant to scalar multiples, so we want the phase of
    %    q_current * conjugate(q_previous)
    % which we compute now (Eq. 7 of tech. report). Under the constant orientation assumption,
    % we can assume the fourth component of the product is zero.
    q_conj_prod_real = current_real.*previous_real + ...
                       current_x.*previous_x + ...
                       current_y.*previous_y;
    q_conj_prod_x = -current_real.*previous_x + previous_real.*current_x;
    q_conj_prod_y = -current_real.*previous_y + previous_real.*current_y;

    % Now we take the quaternion logarithm of this (Eq. 12 in tech. report)
    % Only the imaginary part corresponds to quaternionic phase.
    q_conj_prod_amplitude = sqrt(q_conj_prod_real.^2 + q_conj_prod_x.^2 + q_conj_prod_y.^2);
    phase_difference = acos(q_conj_prod_real./q_conj_prod_amplitude);
    cos_orientation = q_conj_prod_x ./ sqrt(q_conj_prod_x.^2+q_conj_prod_y.^2);
    sin_orientation = q_conj_prod_y ./ sqrt(q_conj_prod_x.^2+q_conj_prod_y.^2);

    % This is the quaternionic phase (Eq. 2 in tech. report)
    phase_difference_cos = phase_difference .* cos_orientation;
    phase_difference_sin = phase_difference .* sin_orientation;

    % Under the assumption that changes are small between frames, we can
    % assume that the amplitude of both coefficients is the same. So,
    % to compute the amplitude of one coefficient, we just take the square root
    % of their conjugate product
    amplitude = sqrt(q_conj_prod_amplitude);

    return {phase_difference_cos, phase_difference_sin, amplitude)
```

```
IIRTemporalFilter(B, A, phase, register0, register1)
    % Temporally filters phase with IIR filter with coefficients B, A.
    % Given current phase value and value of previously computed registers,
    % comptues current temporally filtered phase value and updates registers.
    % Assumes filter given by B, A is first order IIR filter, so that
```

```
    % B and A have 3 coefficients each. Also, assumes A(1) = 1. Computation
    % is Direct Form Type II (See pages 388-390 of Oppenheim and Schafer 3rd Ed.)
    temporally_filtered_phase = B(1) * phase + register0;
    register0 = B(2) * phase + register1 - A(2) * temporally_filtered_phase;
    register1 = B(3) * phase             - A(3) * temporally_filtered_phase;
    return {temporally_filtered_phase, register0, register1}
```

```
AmplitudeWeightedBlur(temporally_filtered_phase, amplitude, blur_kernel)
    % Spatially blurs phase, weighted by amplitude. One half of Eq. 23 in tech. report.
    denominator = Convolve(amplitude, blur_kernel);
    numerator   = Convolve(temporally_filtered_phase.*amplitude, blur_kernel);
    spatially_smooth_temporally_filtered_phase = numerator./denominator;
    return spatially_smooth_temporally_filtered_phase;
```

```
PhaseShiftCoefficientRealPart(riesz_real, riesz_x, riesz_y, phase_cos, phase_sin)
    % Phase shifts a Riesz pyramid coefficient and returns the real part of the
    % resulting coefficient. The input coefficient is a three
    % element quaternion. The phase is two element imaginary quaternion.
    % The phase is exponentiated and then the result is mutiplied by the first
    % coefficient. All operations are defined on quaternions.

    % Quaternion Exponentiation
    phase_magnitude = sqrt(phase_cos.^2+phase_sin.^2);  % \|v\| in Eq. 11 in tech. report.
    exp_phase_real = cos(phase_magnitude);
    exp_phase_x = phase_cos./phase_magnitude.*sin(phase_magnitude);
    exp_phase_y = phase_sin./phase_magnitude.*sin(phase_magnitude);

    % Quaternion Multiplication (just real part)
    result = exp_phase_real.*riesz_real ...
             - exp_phase_x.*riesz_x ...
             - exp_phase_y.*riesz_y;
    return result;
```

References

[1] Oppenheim, A. V., and Schafer, R. W. *Discrete-Time Signal Processing*. Prentice Hall Press, 2009.

[2] Wadhwa, N., Rubinstein, M., Durand, F., and Freeman, W. T. Quaternionic representation of the riesz pyramid for video magnification.

[3] Wadhwa, N., Rubinstein, M., Durand, F., and Freeman, W. T. Riesz pyramids for fast phase-based video magnification. In *Computational Photography (ICCP), 2014 IEEE International Conference on* (2014), IEEE, pp. 1–10.