Project 2

by Jiangyong Huang on Oct 18, 2022

```
[ ]: # import packages
import os
import sys
import math
import numpy as np
import skimage.io as skio
import ctypes
from numpy.ctypeslib import ndpointer, as_array
from tqdm import trange
from scipy.ndimage import correlate
from skimage.transform import resize
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1: Julesz ensemble

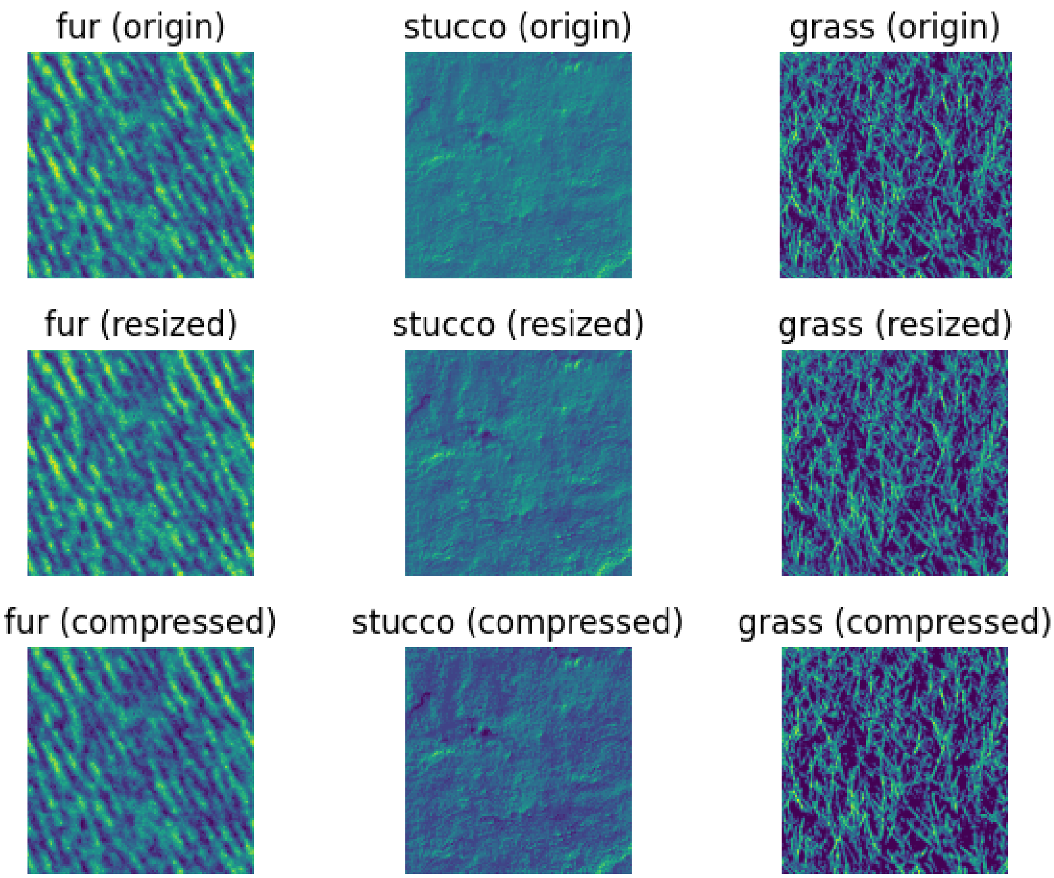
- First, we load the three target images whose texture distributions are what we are going to approximate. To obtain a unified view, we resize them to 256x256. For computation efficiency, we map the domain of each pixel from 8-bit unsigned integer to $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Codes and visualizations are shown as below:

```
[ ]: # load examples
example_fur = skio.imread('images/fur_obs.jpg', as_gray=True)
example_stucco = skio.imread('images/stucco.bmp', as_gray=True)
example_grass = skio.imread('images/grass_obs.bmp', as_gray=True) / 255   #␣
 ↪uint8 to [0, 1]

plt.subplot(3, 3, 1)
plt.title('fur (origin)')
plt.imshow(example_fur)
plt.axis('off')

plt.subplot(3, 3, 2)
plt.title('stucco (origin)')
```

```
plt.imshow(example_stucco)
plt.axis('off')

plt.subplot(3, 3, 3)
plt.title('grass (origin)')
plt.imshow(example_grass)
plt.axis('off')

print('Original size:', example_fur.shape, example_stucco.shape, example_grass.
 ↪shape)

# resize
example_fur = resize(example_fur, (256, 256), mode='symmetric',␣
 ↪preserve_range=True)
example_stucco = resize(example_stucco, (256, 256), mode='symmetric',␣
 ↪preserve_range=True)
example_grass = resize(example_grass, (256, 256), mode='symmetric',␣
 ↪preserve_range=True)

plt.subplot(3, 3, 4)
plt.title('fur (resized)')
plt.imshow(example_fur)
plt.axis('off')

plt.subplot(3, 3, 5)
plt.title('stucco (resized)')
plt.imshow(example_stucco)
plt.axis('off')

plt.subplot(3, 3, 6)
plt.title('grass (resized)')
plt.imshow(example_grass)
plt.axis('off')

print('Resized:', example_fur.shape, example_stucco.shape, example_grass.shape)

# mapping from 8-bit uint to {0,1,2,3,4,5,6,7}
example_fur = (example_fur*8).astype(int)
example_stucco = (example_stucco*8).astype(int)
example_grass = (example_grass*8).astype(int)

plt.subplot(3, 3, 7)
plt.title('fur (compressed)')
plt.imshow(example_fur/8)
plt.axis('off')

plt.subplot(3, 3, 8)
```

```
plt.title('stucco (compressed)')
plt.imshow(example_stucco/8)
plt.axis('off')

plt.subplot(3, 3, 9)
plt.title('grass (compressed)')
plt.imshow(example_grass/8)
plt.axis('off')

plt.tight_layout()
```

```
Original size: (469, 469) (512, 511) (291, 299)
Resized: (256, 256) (256, 256) (256, 256)
```



- Next, we define a series of functions for Julesz ensemble modeling.
  - `fspecial_log` and `gaborfilter` are used for filters generation, and `get_filters` will return a bank of pre-defined filters. In addition to aforementioned filters, I also define filters for horizontal or vertical gradients, Laplacian operator, and Dirac δ response. Among these filters, Dirac δ response is used to extract the proportion of a specific pixel intensity, and its histogram is derived via a binary response. More details can be found

in the codes.

- `compute_responses` applies the input filters on the image and returns the responses in continuous values. To convert these responses into histograms, we implement `histogram_matching` to provide aligned bins with equal interval bounded by pre-computed minimum value and maximum value. To address the importance of histogram tails, we assign a symmetric array with higher weights on tails for the weighted summation on errors.
- `sample_update` and `gibbs_sample_python` depicts the procedures of Gibbs sampling. During each sweep, we enumerate each pixel to compute the conditional probabilities on its domain and next sample a value according to the probabilities. Considering the annealing schema, we gradually reduce the temperature along the dimension of sweeping. Specifically, since the joint distribution of Julesz ensemble is defined as $p(I) = \frac{1}{Z_T} \exp\left(-\frac{\sum_{\alpha=1}^{K} |H_I^{(\alpha)} - H_{gt}^{(\alpha)}|}{T}\right)$, the conditional distribution can be computed by tentatively taking each candidate value and modifying corresponding histograms.
- However, limited by the poor efficiency of `Python`, particularly the bottleneck at Gibbs sampling, we have to leverage `C` extension to accelerate this process. The module `gibbs_sample_C` incorporates the implementation in `C` programming.
- Finally, we can define the whole pipeline of Julesz ensemble in `Julesz_ensemble`. To visualize the experimental results, we make `visualize_sequence` function for the visualization of the pursuit process.

[ ]:
```
def fspecial_log(p2, p3):
    """
    equivalent to MATLAB's fspecial('log',...) function
    case 'log' % Laplacian of Gaussian
        % first calculate Gaussian
        siz   = (p2-1)/2;
        std2   = p3^2;

        [x,y] = meshgrid(-siz(2):siz(2),-siz(1):siz(1));
        arg   = -(x.*x + y.*y)/(2*std2);

        h     = exp(arg);
        h(h<eps*max(h(:))) = 0;

        sumh = sum(h(:));
        if sumh ~= 0,
        h  = h/sumh;
        end;
        % now calculate Laplacian
        h1 = h.*(x.*x + y.*y - 2*std2)/(std2^2);
        h     = h1 - sum(h1(:))/prod(p2); % make the filter sum to zero
    """

    siz = int((p2-1)/2)
    std = p3
```

```python
    x = y = np.linspace(-siz, siz, 2*siz+1)
    x, y = np.meshgrid(x, y)
    arg = -(x**2 + y**2) / (2*std**2)
    h = np.exp(arg)
    h[h < sys.float_info.epsilon * h.max()] = 0
    h = h/h.sum() if h.sum() != 0 else h
    h1 = h*(x**2 + y**2 - 2*std**2) / (std**4)
    return h1 - h1.mean()


def gaborfilter(size, orientation):
    """
      [Cosine, Sine] = gaborfilter(scale, orientation)

      Defintion of "scale": the sigma of short-gaussian-kernel used in gabor.
      Each pixel corresponds to one unit of length.
      The size of the filter is a square of size n by n.
      where n is an odd number that is larger than scale * 6 * 2.
    """

    assert size % 2 != 0

    halfsize = math.ceil(size / 2)
    theta = (math.pi * orientation) / 180
    Cosine = np.zeros((size, size))
    Sine = np.zeros((size, size))
    gauss = np.zeros((size, size))
    scale = size / 6

    for i in range(size):
        for j in range(size):
            x = ((halfsize - (i+1)) * np.cos(theta) + (halfsize-(j+1)) * np.
↪sin(theta)) / scale
            y = (((i+1) - halfsize) * np.sin(theta) + (halfsize-(j+1)) * np.
↪cos(theta)) / scale

            gauss[i, j] = np.exp(-(x**2 + y**2/4) / 2)
            Cosine[i, j] = gauss[i, j] * np.cos(2*x)
            Sine[i, j] = gauss[i, j] * np.sin(2*x)

    k = np.sum(np.sum(Cosine)) / np.sum(np.sum(gauss))
    Cosine = Cosine - k * gauss

    return Cosine, Sine
```

```python
def get_filters():
    """
    define set of filters
    """
    array = lambda x: np.array(x)

    # filters for naive gradients
    F = [array([-1.0, 1.0]).reshape((1, 2)), array([-1.0, 1.0]).reshape((2, 1))]

    # Laplacian filter
    F += [array([[0, 1, 0], [1, -4, 1], [0, 1, 0]]).astype(np.float32)]

    # filters for dirac delta response, binary response
    # F += list(range(8))

    F += [fspecial_log((i-2)*2+1, (i-2)/3) for i in range(3, 5+1)]

    for i in range(7, 8):
        for j in range(0, 150+1, 30):
            F += gaborfilter(i, j)

    n_F = len(F)
    width, height = np.zeros(n_F), np.zeros(n_F)
    filters = np.zeros((n_F, np.max([f.shape[0]*f.shape[1] for f in F])))
    for i in range(n_F):
        [m, n] = F[i].shape

        F[i] -= F[i].mean()
        F[i] /= np.abs(F[i]).sum()

        filters[i, 0:(m*n)] = F[i].reshape(-1)
        height[i] = n
        width[i] = m

    return F, filters, width, height


def compute_responses(img, filter_bank):
    num_filters = len(filter_bank)
    responses = [correlate(img, filter, output=np.float64, mode='constant',␣
 ↪cval=0) for filter in filter_bank]
    responses = np.stack(responses, axis=0)
    return responses


BAND_WEIGHTS = np.array([8, 7, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8])
```

```python
def histogram_matching(responses_syn, responses_gt, num_bins=15,␣
↪weighted=False):
    """
    :responses_syn: [num_filters, H, W]
    :responses_gt: [num_filters, H, W]
    """
    num_filters, H, W = responses_gt.shape
    bounds = np.zeros((num_filters, 2))
    bounds[:, 0] = responses_gt.min(axis=(1, 2)) - 1e-6
    bounds[:, 1] = responses_gt.max(axis=(1, 2)) + 1e-6

    for i in range(num_filters):
        responses_syn[i] = np.clip(responses_syn[i], bounds[i, 0], bounds[i, 1])

    hists_syn = np.zeros((num_filters, num_bins))
    hists_gt = np.zeros((num_filters, num_bins))
    for i in range(num_filters):
        bins = np.linspace(bounds[i, 0], bounds[i, 1], num=num_bins+1)
        hists_syn[i] = np.histogram(responses_syn[i], bins=bins,␣
↪density=False)[0] / (H*W)
        hists_gt[i] = np.histogram(responses_gt[i], bins=bins,␣
↪density=False)[0] / (H*W)

    if weighted:
        errors = ( np.abs(hists_syn-hists_gt)*BAND_WEIGHTS ).sum(axis=-1)
    else:
        errors = np.abs(hists_syn-hists_gt).sum(axis=-1)

    return hists_syn, hists_gt, bounds, errors


def sample_update(img_syn, responses_syn, hists_syn, hists_gt, current_pos,␣
↪filters, bounds, num_bins, T, multipliers=None):
    """
    Gibbs sampling on each pixel, implemented in python, very slow

    Args:
        img_syn: image sample at shape [H, W], with assignment attempt of `k`␣
↪at `current_pos`
        responses_syn: image responses at shape [num_chosen_filters, H, W]
        hists_syn: normalized response histograms at shape [num_chosen_filters,␣
↪num_bins]
        hists_gt: normalized response histograms at shae [num_chosen_filters,␣
↪num_bins]
        current_pos: tuple, pixel at (row, column) where the value is re-sampled
        filters: list of chosen filters
```

```
        bounds: lower/upper bounds of histograms for each filter, at shape [num_chosen_filters, 2]
        T: temperature
    """
    H, W = img_syn.shape
    i, j = current_pos
    gray_ori = img_syn[i, j]
    num_chosen_filters = len(filters)
    hists_tmp = hists_syn.copy()[None, :, :].repeat(8, 0) * (H*W)

    # compute conditional probabilities
    for k in range(8):
        for f in range(num_chosen_filters):
            h, w = filters[f].shape
            bin_width = (bounds[f, 1] - bounds[f, 0]) / num_bins
            for y in range(h):
                for x in range(w):
                    # `current_pos` multiplied by pixel (y, x) on filter
                    modified_response_x, modified_response_y = j + int((w-1)/2) - x, i + int((h-1)/2) - y

                    # TODO: in case of boundary (solve reflect padding)
                    if modified_response_x < 0 or modified_response_x > W-1 or modified_response_y < 0 or modified_response_y > H-1:
                        continue

                    response_z = responses_syn[f, modified_response_y, modified_response_x]
                    response_bin = np.clip(np.floor((response_z-bounds[f, 0])/bin_width), 0, num_bins-1).astype(int)
                    hists_tmp[k, f, response_bin] -= 1

                    response_z += ( (k-gray_ori) * filters[f][y, x] )
                    response_bin = np.clip(np.floor((response_z-bounds[f, 0])/bin_width), 0, num_bins-1).astype(int)
                    hists_tmp[k, f, response_bin] += 1

    hists_tmp /= (H*W)
    if multipliers is None:
        # Julesz ensemble
        probs = np.abs(hists_tmp - hists_gt).sum(axis=(1, 2))
    else:
        # FRAME
        probs = (multipliers*hists_tmp).sum(axis=(1, 2))

    probs = np.exp(-probs/T)
```

```
    probs /= probs.sum()

    # sample
    img_syn[i, j] = np.random.choice(8, size=1, p=probs)

    # update histograms
    for f in range(num_chosen_filters):
        h, w = filters[f].shape
        bin_width = (bounds[f, 1] - bounds[f, 0]) / num_bins
        for y in range(h):
            for x in range(w):
                # `current_pos` multiplied by pixel (y, x) on filter
                modified_response_x, modified_response_y = j + int((w-1)/2) -␣
↪x, i + int((h-1)/2) - y

                # TODO: in case of boundary (solve reflect padding)
                if modified_response_x < 0 or modified_response_x > W-1 or␣
↪modified_response_y < 0 or modified_response_y > H-1:
                    continue

                responses_syn[f, modified_response_y, modified_response_x] +=␣
↪((img_syn[i, j] - gray_ori) * filters[f][y, x])

    hists_syn = hists_tmp[img_syn[i, j]]

    return img_syn, responses_syn, hists_syn


def gibbs_sample_python(img_syn, responses_syn, hists_syn, hists_gt, filters,␣
↪bounds, num_bins, sweep=50, multipliers=None):
    # pipeline in python
    T = 1
    for s in trange(sweep):
        for i in range(img_syn.shape[0]):
            for j in range(img_syn.shape[1]):
                img_syn, responses_syn, hists_syn = sample_update(
                    img_syn=img_syn, responses_syn=responses_syn,␣
↪hists_syn=hists_syn,
                    hists_gt=hists_gt, current_pos=(i, j), filters=filters,␣
↪bounds=bounds,
                    num_bins=num_bins, T=T, multipliers=multipliers
                )

        if s % 10 == 9:
            # mean L-1 distance between histograms averaged over filters
```

```
            print(f'Gibbs iteration {s+1}: error = {np.abs(hists_syn-hists_gt).sum(axis=-1).mean()}')
        T *= 0.96

    return img_syn


def gibbs_sample_C(lib, img_syn, responses_syn, hists_syn, hists_gt, filtermatrix, hs, ws, bounds, num_bins, sweep=50, multipliers=None):
    # leveraging C extension
    H, W = img_syn.shape
    num_filters, max_size = filtermatrix.shape

    ndpointerpointer = lambda: ndpointer(dtype=np.uintp, ndim=1, flags='C')
    c_array = lambda a: (a.__array_interface__['data'][0] + np.arange(a.shape[0]) * a.strides[0]).astype(np.uintp)

    Gibbs = lib.Gibbs
    Gibbs.restype = None
    Gibbs.argtypes = [
        ndpointerpointer(), ndpointer(dtype=np.int32, ndim=1), ndpointer(dtype=np.int32, ndim=1), ctypes.c_int, ctypes.c_int,
        ctypes.c_int, ctypes.c_int, ndpointerpointer(), ndpointer(dtype=np.float64, ndim=1), ctypes.c_int,
        ctypes.c_int, ndpointerpointer(),
        ndpointer(dtype=np.int32, ndim=1), ndpointerpointer(), ndpointerpointer()
    ]

    img_syn = np.ascontiguousarray(img_syn.reshape(-1)).astype(np.int32)
    Gibbs(
        c_array(filtermatrix), hs.astype(np.int32), ws.astype(np.int32), H, W,
        num_filters, num_bins, c_array(hists_gt), bounds.reshape(-1), sweep,
        1 if multipliers is not None else 0, c_array(multipliers) if multipliers is not None else c_array(np.zeros((1, 1), dtype=np.float64)),
        img_syn, c_array(responses_syn.reshape(num_filters, -1)), c_array(hists_syn)
    )

    return np.ascontiguousarray(img_syn.reshape(H, W)).astype(np.int32)


def Julesz_ensemble(target, filter_bank, num_bins=15):
    F, filters, width, height = filter_bank
    lib = ctypes.cdll.LoadLibrary('./lib_gibbs.so')
```

```
    # compute responses of target image
    responses_gt = compute_responses(target, F)

    # initialize chosen filters and synthetic samples
    filters_chosen_idx = []
    imgs_syn = []
    error_list = []
    current_error = 100    # dummy assignment
    threshold = 1e-2
    round = 0

    # repeat adding filters
    while current_error >= threshold:
        round += 1
        print(f'Generation round {round}')
        if len(imgs_syn) == 0:
            img_syn = np.random.choice(8, size=(256, 256), p=[0.05, 0.1, 0.15, 0.2, 0.2, 0.15, 0.1, 0.05])
            responses_syn = compute_responses(img_syn, F)
            hists_syn, hists_gt, bounds, errors = histogram_matching(responses_syn, responses_gt, num_bins, weighted=False)
            imgs_syn.append(img_syn.copy())
        else:
            img_syn = gibbs_sample_C(
                lib=lib, img_syn=img_syn.copy(), responses_syn=responses_syn[filters_chosen_idx].copy(),
                hists_syn=hists_syn[filters_chosen_idx].copy(), hists_gt=hists_gt[filters_chosen_idx].copy(),
                filtermatrix=filters[filters_chosen_idx].copy(), hs=height[filters_chosen_idx], ws=width[filters_chosen_idx],
                bounds=bounds[filters_chosen_idx].copy(), num_bins=num_bins, sweep=50, multipliers=None
            )
            responses_syn = compute_responses(img_syn, F)
            hists_syn, _, _, errors = histogram_matching(responses_syn, responses_gt, num_bins, weighted=False)
            imgs_syn.append(img_syn.copy())

        print(f'Updating filters round {round}')
        errors[filters_chosen_idx] = 0
        idx_to_choose = np.argmax(errors)
        filters_chosen_idx.append(idx_to_choose)

        current_error = errors[idx_to_choose]
        error_list.append(current_error)
        print(f'Error after round {round}: {current_error:.4f}\n\n')
```

```
        if len(filters_chosen_idx) == len(F):
            break

    # one last synthesis
    img_syn = gibbs_sample_C(
        lib=lib, img_syn=img_syn.copy(),␣
 ↪responses_syn=responses_syn[filters_chosen_idx].copy(),
        hists_syn=hists_syn[filters_chosen_idx].copy(),␣
 ↪hists_gt=hists_gt[filters_chosen_idx].copy(),
        filtermatrix=filters[filters_chosen_idx].copy(),␣
 ↪hs=height[filters_chosen_idx], ws=width[filters_chosen_idx],
        bounds=bounds[filters_chosen_idx].copy(), num_bins=num_bins, sweep=50,␣
 ↪multipliers=None
    )
    imgs_syn.append(img_syn.copy())

    return imgs_syn, filters_chosen_idx, error_list


def visualize_sequence(filters_chosen, imgs_syn, show_reso=256):
    # jointly visualize sequences of chosen filters and synthetic samples
    num_fig = len(filters_chosen) + len(imgs_syn)
    num_row = num_fig // 8 + 1
    for i in range(num_fig):
        plt.subplot(num_row, 8, i+1)
        if i % 2 == 0:
            plt.title(f'img {i//2}')
            plt.imshow(resize(imgs_syn[i//2], output_shape=(show_reso,␣
 ↪show_reso), preserve_range=True)/8)
        else:
            plt.title(f'flt {i//2+1}')
            plt.imshow(resize(filters_chosen[i//2], output_shape=(show_reso,␣
 ↪show_reso), preserve_range=True))
        plt.axis('off')
    plt.tight_layout()
```

- After defining the above functions, now we can run the following cells to solve Julesz ensemble models. Each call of `Julesz_ensemble` solves a kind of texture. Note that they may take long to obtain the final solutions, roughly an hour for each.

[ ]:
```
filter_bank = get_filters()
F, filters, width, height = filter_bank
```

[ ]:
```
# fur
imgs_syn_fur_1, filters_chosen_idx_fur_1, error_list_fur =␣
 ↪Julesz_ensemble(example_fur, filter_bank, num_bins=15)
```

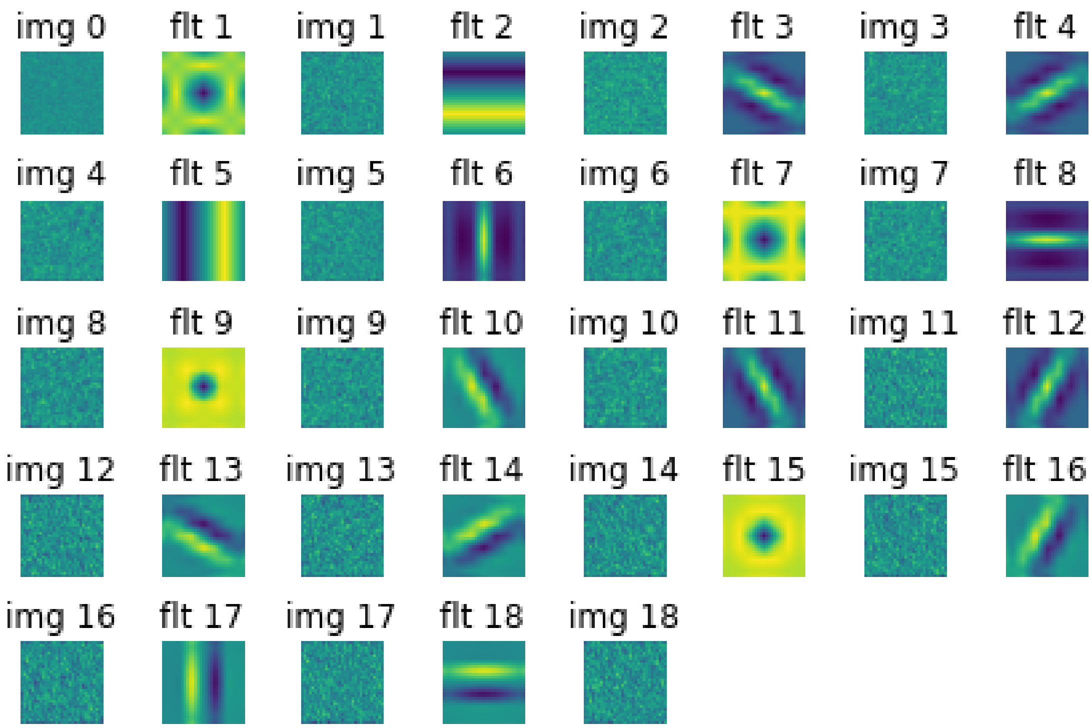
```
[ ]: # stucco
imgs_syn_stucco_1, filters_chosen_idx_stucco_1, error_list_stucco =␣
 ↪Julesz_ensemble(example_stucco, filter_bank, num_bins=15)
```

```
[ ]: # grass
imgs_syn_grass_1, filters_chosen_idx_grass_1, error_list_grass =␣
 ↪Julesz_ensemble(example_grass, filter_bank, num_bins=15)
```

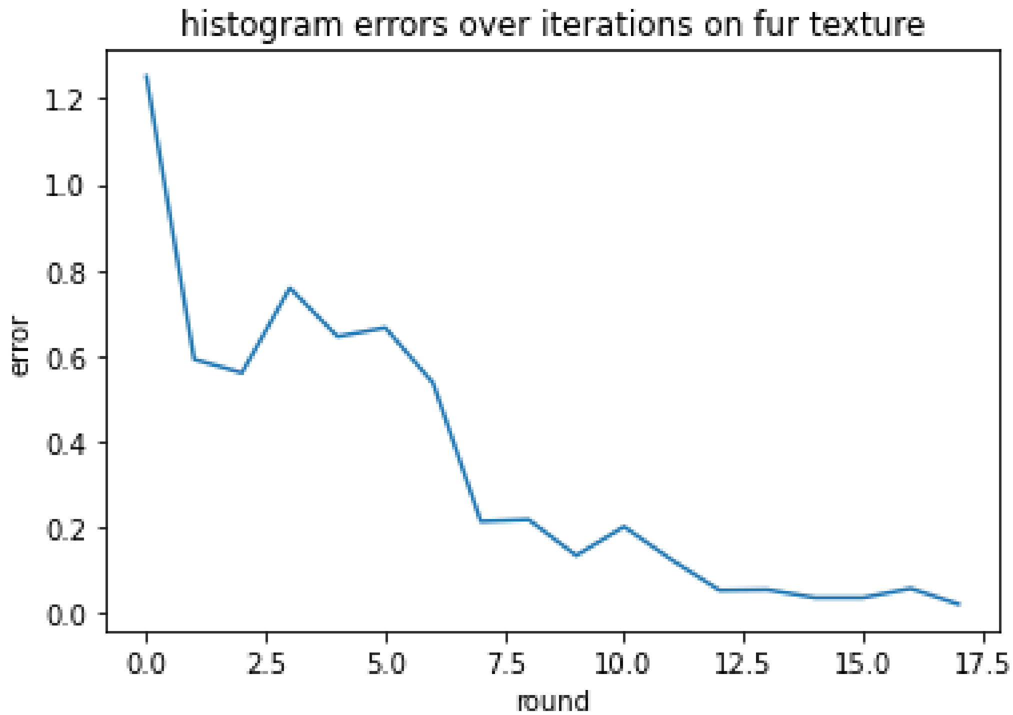
- Now we get the solutions for validating the final results, as well as intermediate variables for visualization of the pursuit process. Run the following several cells to visualize pursuit processes, histogram errors and final comparison for each type of texture.

```
[ ]: print('Evolution of filters and synthesized samples on fur texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_fur_1],␣
 ↪imgs_syn=imgs_syn_fur_1)
```

```
Evolution of filters and synthesized samples on fur texture
```

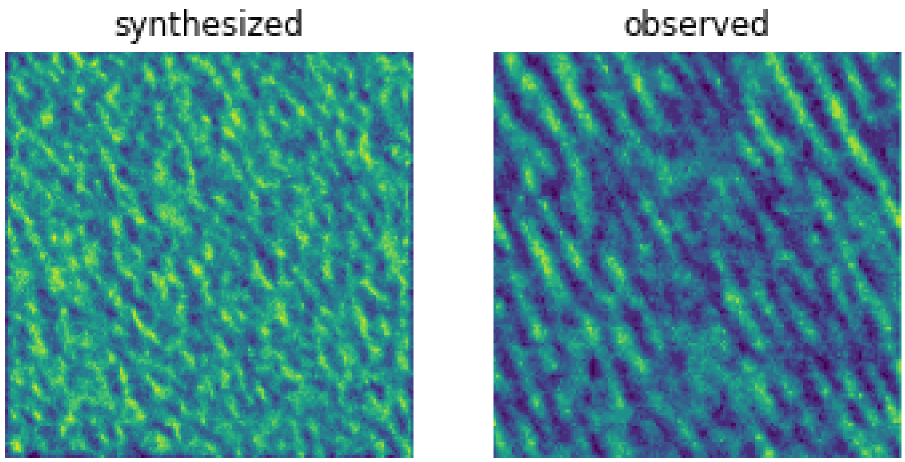



```
[ ]: plt.plot(error_list_fur)
plt.title('histogram errors over iterations on fur texture')
plt.xlabel('round')
plt.ylabel('error')
plt.show()
```



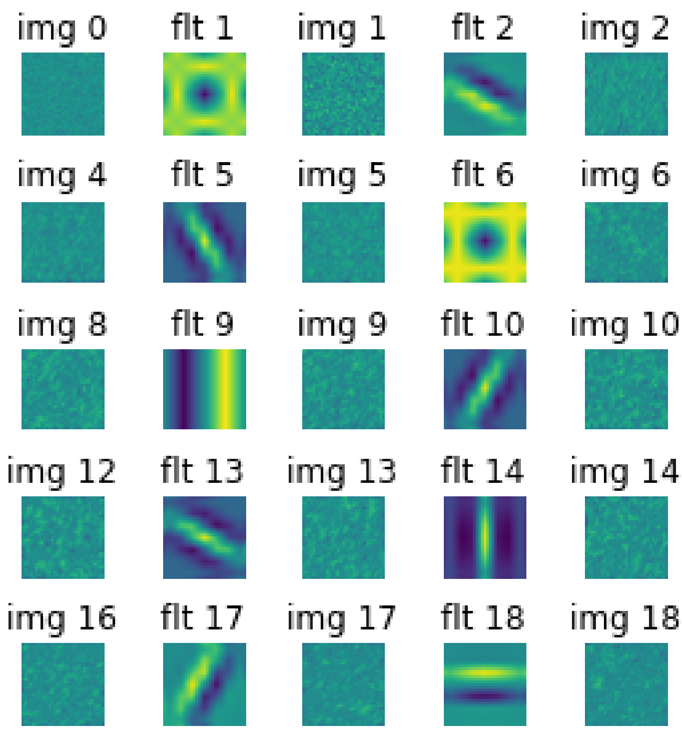
[ ]:
```
print('Final result of Julesz ensemble on fur texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_fur_1[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_fur/8)
plt.axis('off')
plt.show()
```

```
Final result of Julesz ensemble on fur texture
```

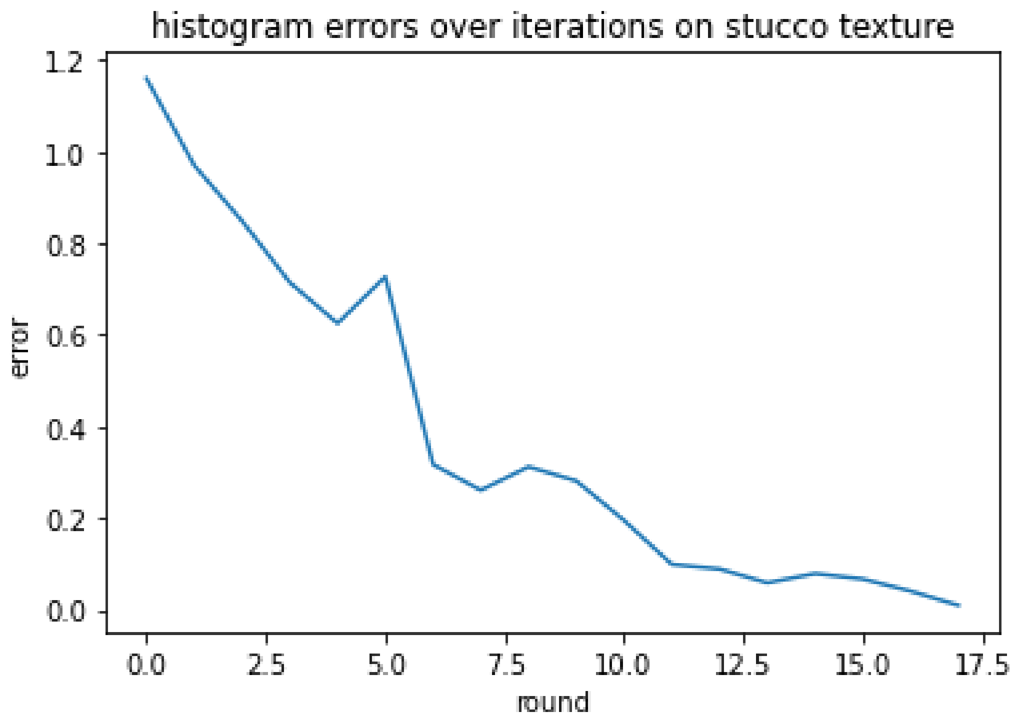



```
[ ]: print('Evolution of filters and synthesized samples on stucco texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_stucco_1],
 ↪imgs_syn=imgs_syn_stucco_1)
```

```
Evolution of filters and synthesized samples on stucco texture
```

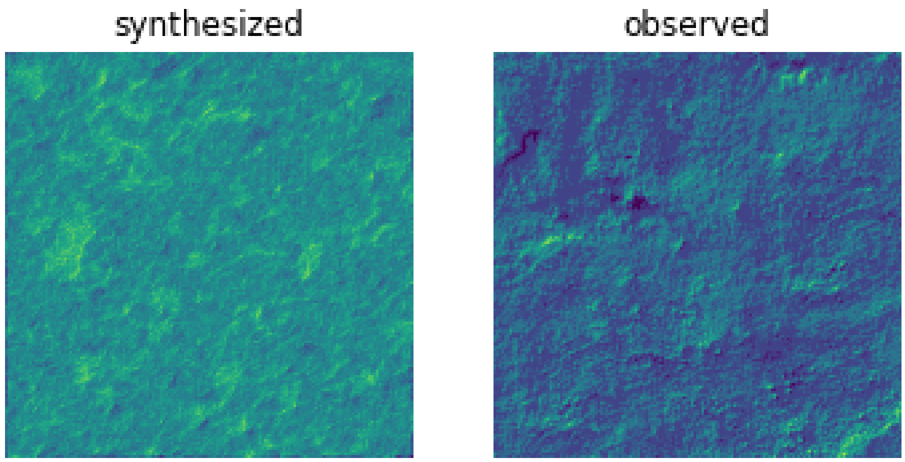


[ ]:
```
plt.plot(error_list_stucco)
plt.title('histogram errors over iterations on stucco texture')
plt.xlabel('round')
plt.ylabel('error')
plt.show()
```



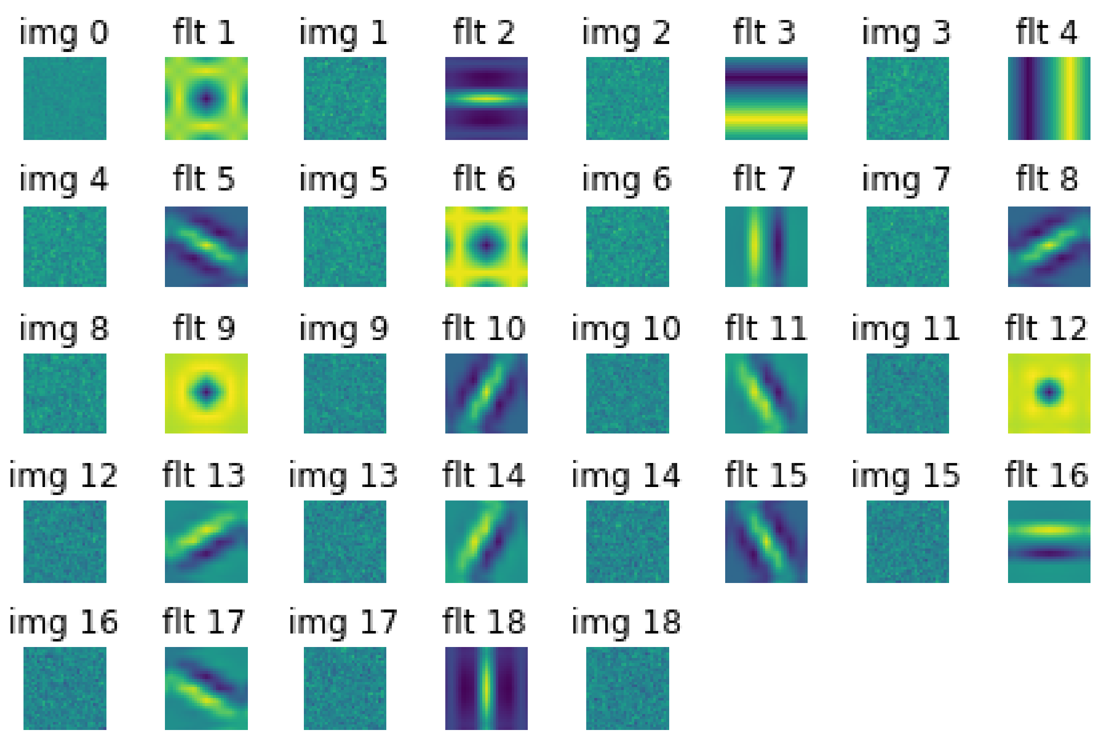
[ ]:
```
print('Final result of Julesz ensemble on stucco texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_stucco_1[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_stucco/8)
plt.axis('off')
plt.show()
```

```
Final result of Julesz ensemble on stucco texture
```

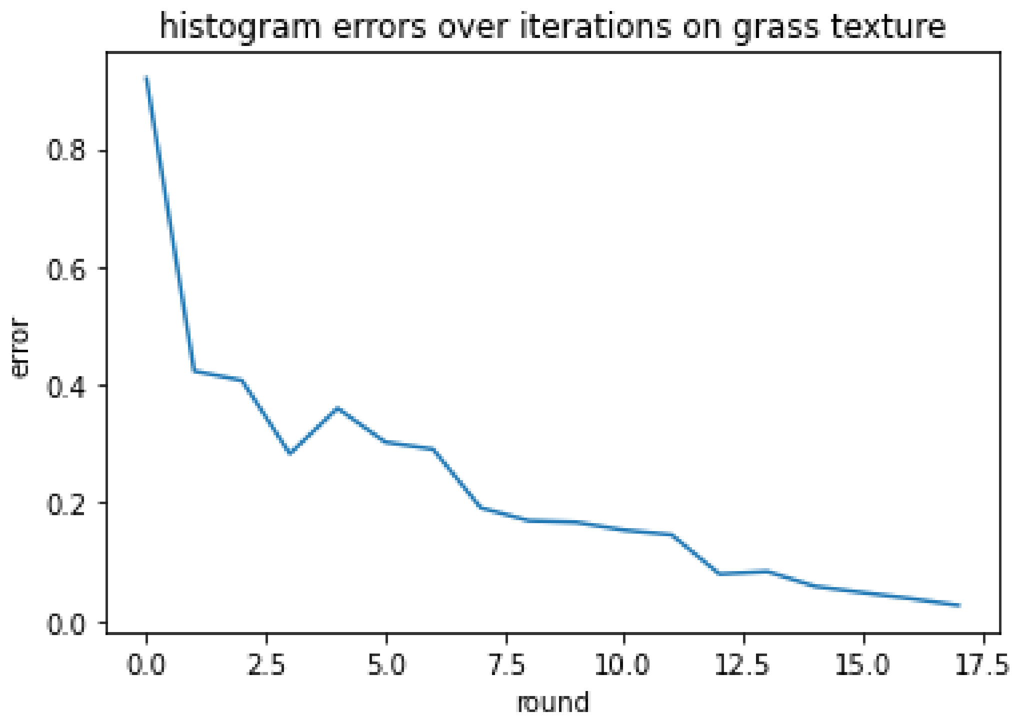



```
[ ]: print('Evolution of filters and synthesized samples on grass texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_grass_1],␣
 ↪imgs_syn=imgs_syn_grass_1)
```

```
Evolution of filters and synthesized samples on grass texture
```

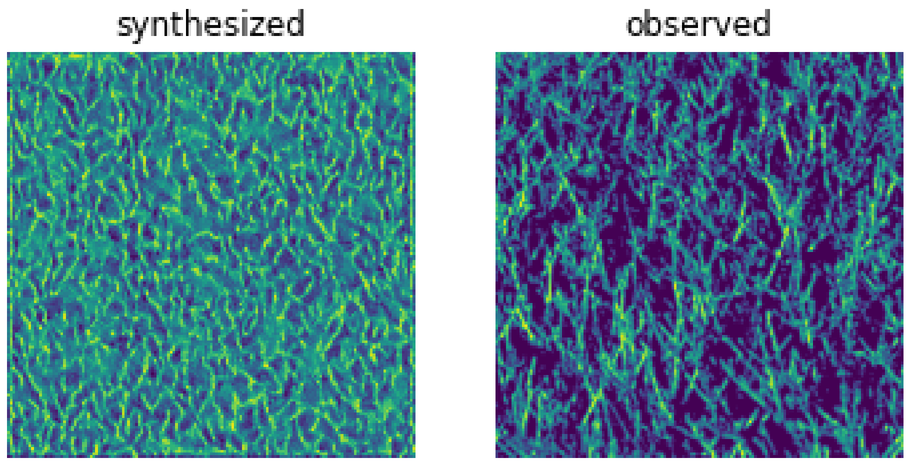


[ ]:
```
plt.plot(error_list_grass)
plt.title('histogram errors over iterations on grass texture')
plt.xlabel('round')
plt.ylabel('error')
plt.show()
```



[ ]:
```
print('Final result of Julesz ensemble on grass texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_grass_1[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_grass/8)
plt.axis('off')
plt.show()
```

```
Final result of Julesz ensemble on grass texture
```



Problem 2: FRAME

- Filters, Random field, And Maximum Entropy (FRAME) is another statistical model that depicts the distribution of visual images regarding texture. The FRAME model can be expressed as $p(I) = \frac{1}{Z_T} \exp\left(-\frac{\sum_{\alpha=1}^{K} <\lambda^{(\alpha)}, H^{(\alpha)}>}{T}\right)$. The equivalence between Julesz ensemble and FRAME model can be derived under some assumptions, and using FRAME to pursue the distribution of texture only involves a few modifications based on our previous pipeline:
  - The enerny term needs to be modified, from $-\sum_{\alpha=1}^{K} |H_I^{(\alpha)} - H_{gt}^{(\alpha)}|$ to $-\sum_{\alpha=1}^{K} < \lambda^{\alpha}, H_I^{\alpha} >$.
  - An extra operation for updating the multipliers λ^{α} is required, which includes deriving the error $H_I^{\alpha} - H_{obs}^{\alpha}$ as gradient and then performing gradient ascent on the multipliers.

[ ]:
```
def FRAME(target, filter_bank, num_bins=15):
    """
    adapted from `julesz_ensemble`
    modify Gibbs sampler, add multipliers and return their trajectories
    """
    F, filters, width, height = filter_bank
    lib = ctypes.cdll.LoadLibrary('./lib_gibbs.so')

    # compute responses of target image
    responses_gt = compute_responses(target, F)

    # initialize chosen filters and synthetic samples
    filters_chosen_idx = []
    imgs_syn = []
    multipliers = np.zeros((len(filters), num_bins))
    multipliers_traj = [multipliers.copy()]
    step_size = 0.1
```

```
    current_error = 100    # dummy assignment
    threshold = 1e-2
    round = 0

    # repeat adding filters
    while current_error >= threshold:
        round += 1
        print(f'Generation round {round}')
        if len(imgs_syn) == 0:
            img_syn = np.random.randint(low=0, high=8, size=(256, 256))
            responses_syn = compute_responses(img_syn, F)
            hists_syn, hists_gt, bounds, errors = histogram_matching(responses_syn, responses_gt, num_bins, weighted=False)
            imgs_syn.append(img_syn.copy())
        else:
            img_syn = gibbs_sample_C(
                lib=lib, img_syn=img_syn.copy(), responses_syn=responses_syn[filters_chosen_idx].copy(),
                hists_syn=hists_syn[filters_chosen_idx].copy(), hists_gt=hists_gt[filters_chosen_idx].copy(),
                filtermatrix=filters[filters_chosen_idx].copy(), hs=height[filters_chosen_idx], ws=width[filters_chosen_idx],
                bounds=bounds[filters_chosen_idx].copy(), num_bins=num_bins, sweep=50, multipliers=multipliers[filters_chosen_idx]
            )
            responses_syn = compute_responses(img_syn, F)
            hists_syn, _, _, errors = histogram_matching(responses_syn, responses_gt, num_bins, weighted=False)
            imgs_syn.append(img_syn.copy())

        print(f'\nUpdating multipliers round {round}')
        gradients = np.zeros_like(multipliers)
        gradients[filters_chosen_idx] = (hists_syn-hists_gt)[filters_chosen_idx]
        # only update multipliers for chosen filters
        multipliers += step_size * gradients
        multipliers_traj.append(multipliers.copy())

        print(f'Updating filters round {round}')
        errors[filters_chosen_idx] = 0
        idx_to_choose = np.argmax(errors)
        filters_chosen_idx.append(idx_to_choose)

        current_error = errors[idx_to_choose]
        print(f'Error after round {round}: {current_error:.4f}\n\n')

        if len(filters_chosen_idx) == len(F):
```

```
            break

    # one last synthesis
    img_syn = gibbs_sample_C(
        lib=lib, img_syn=img_syn.copy(),␣
↪responses_syn=responses_syn[filters_chosen_idx].copy(),
        hists_syn=hists_syn[filters_chosen_idx].copy(),␣
↪hists_gt=hists_gt[filters_chosen_idx].copy(),
        filtermatrix=filters[filters_chosen_idx].copy(),␣
↪hs=height[filters_chosen_idx], ws=width[filters_chosen_idx],
        bounds=bounds[filters_chosen_idx].copy(), num_bins=num_bins, sweep=50,␣
↪multipliers=multipliers[filters_chosen_idx]
    )
    imgs_syn.append(img_syn.copy())

    return imgs_syn, filters_chosen_idx, multipliers_traj
```

- With the updated version of FRAME, we can derive another set of results.

[ ]:
```
# fur
imgs_syn_fur_2, filters_chosen_idx_fur_2, multipliers_traj_fur =␣
↪FRAME(example_fur, filter_bank, num_bins=15)

multipliers_traj_fur = np.stack(multipliers_traj_fur, axis=-1)
multipliers_traj_fur = multipliers_traj_fur[filters_chosen_idx_fur_2]
```

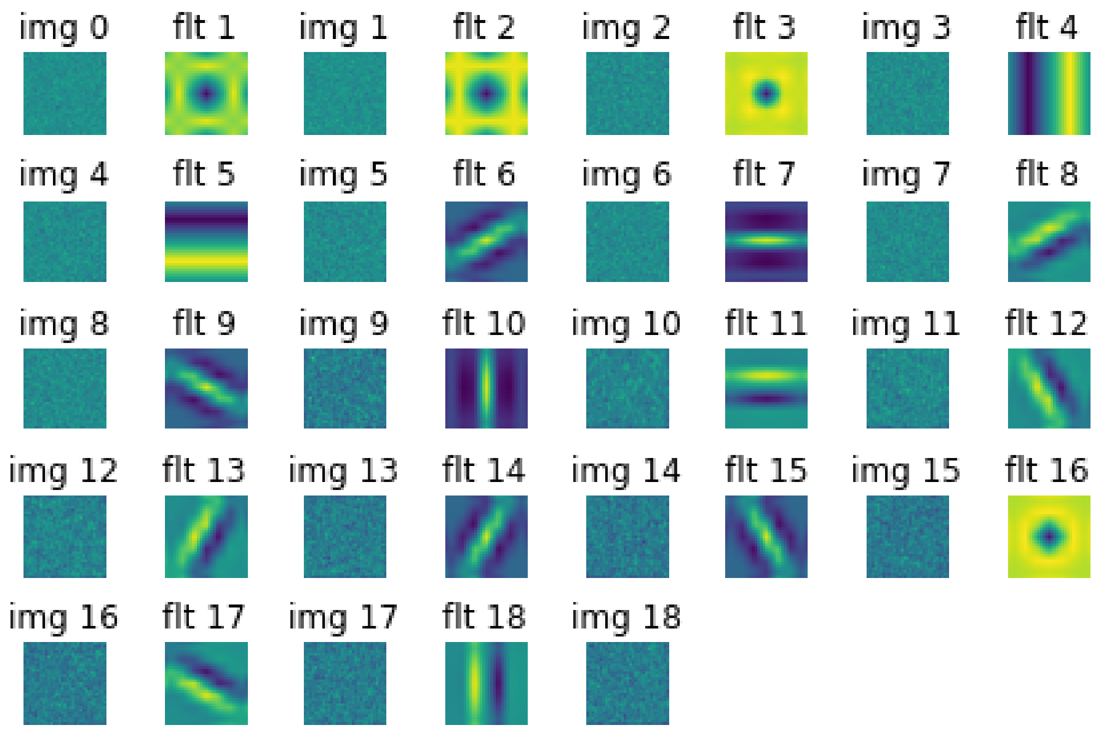
[ ]:
```
# stucco
imgs_syn_stucco_2, filters_chosen_idx_stucco_2, multipliers_traj_stucco =␣
↪FRAME(example_stucco, filter_bank, num_bins=15)

multipliers_traj_stucco = np.stack(multipliers_traj_stucco, axis=-1)
multipliers_traj_stucco = multipliers_traj_stucco[filters_chosen_idx_stucco_2]
```

[ ]:
```
# grass
imgs_syn_grass_2, filters_chosen_idx_grass_2, multipliers_traj_grass =␣
↪FRAME(example_grass, filter_bank, num_bins=15)

multipliers_traj_grass = np.stack(multipliers_traj_grass, axis=-1)
multipliers_traj_grass = multipliers_traj_grass[filters_chosen_idx_grass_2]
```

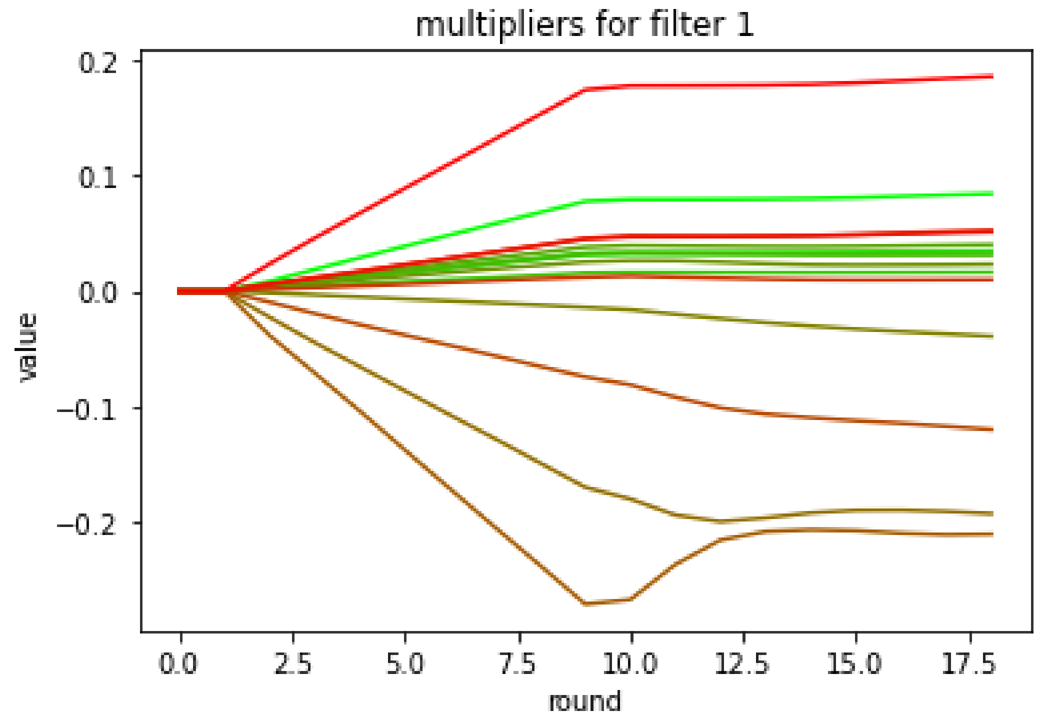
- Now after we derive the solutions, we make the following visualizations. Note that multipliers for a specific filter start to update (become non-zero) only after the filter is chosen, so the curves of multipliers may show a step of being zero at the beginning. Particularly, the multipliers for the last-chosen filter stay zero until finished, since the loop ends right after this filter is chosen, without updating its multipliers.

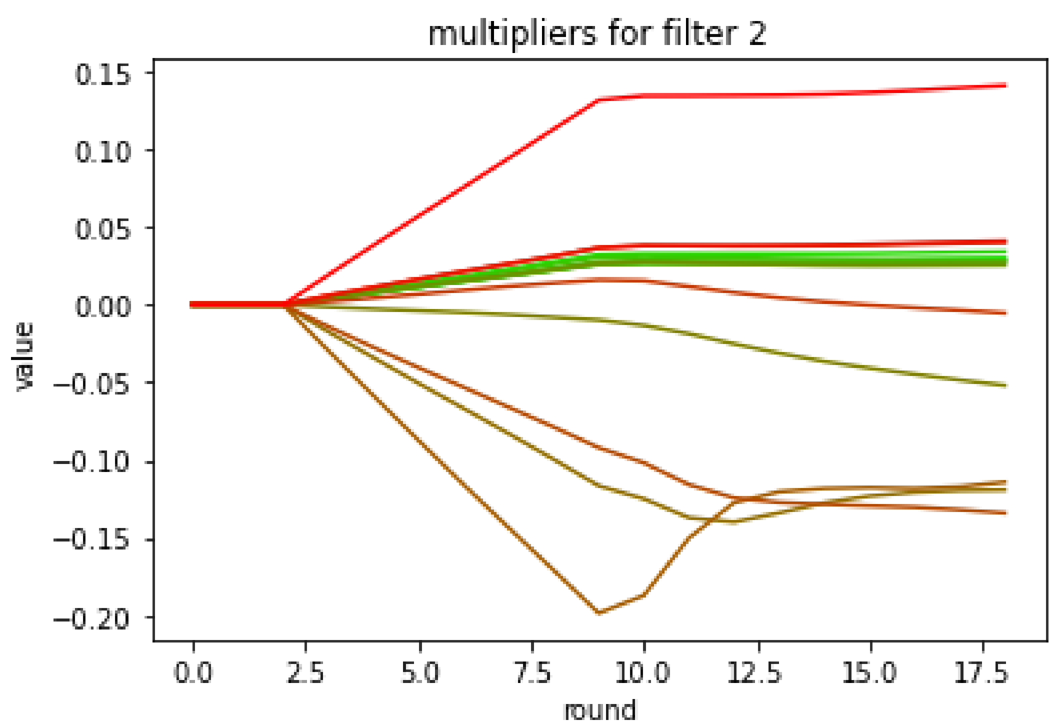
[ ]:
```
print('Evolution of filters and synthesized samples on fur texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_fur_2],␣
↪imgs_syn=imgs_syn_fur_2)
```

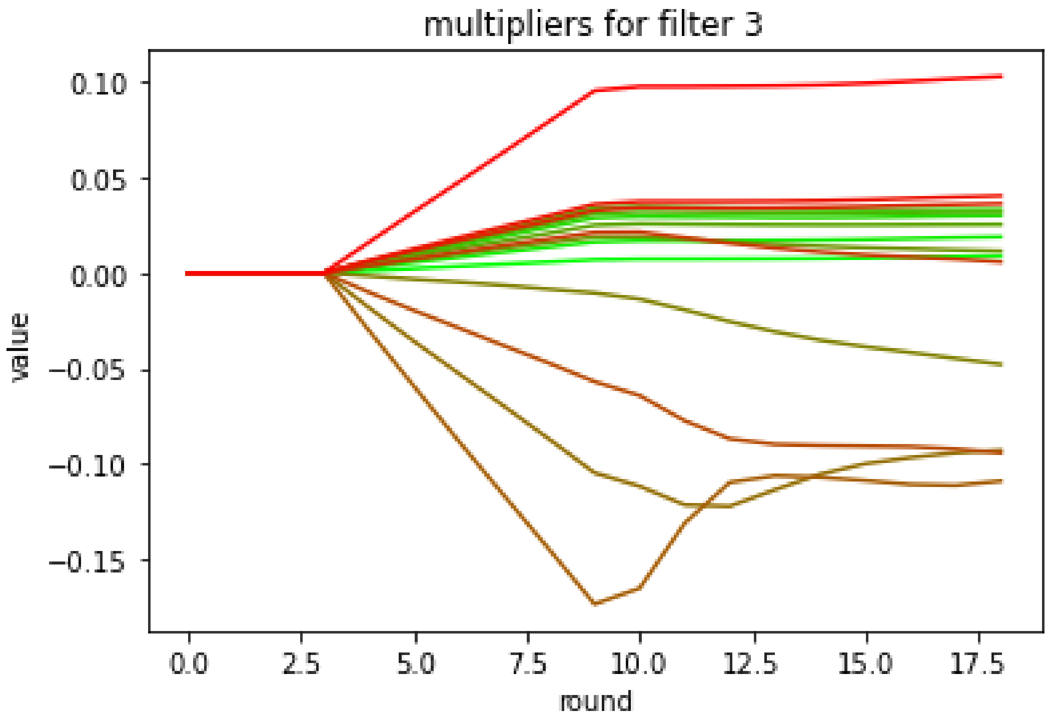
```
Evolution of filters and synthesized samples on fur texture
```

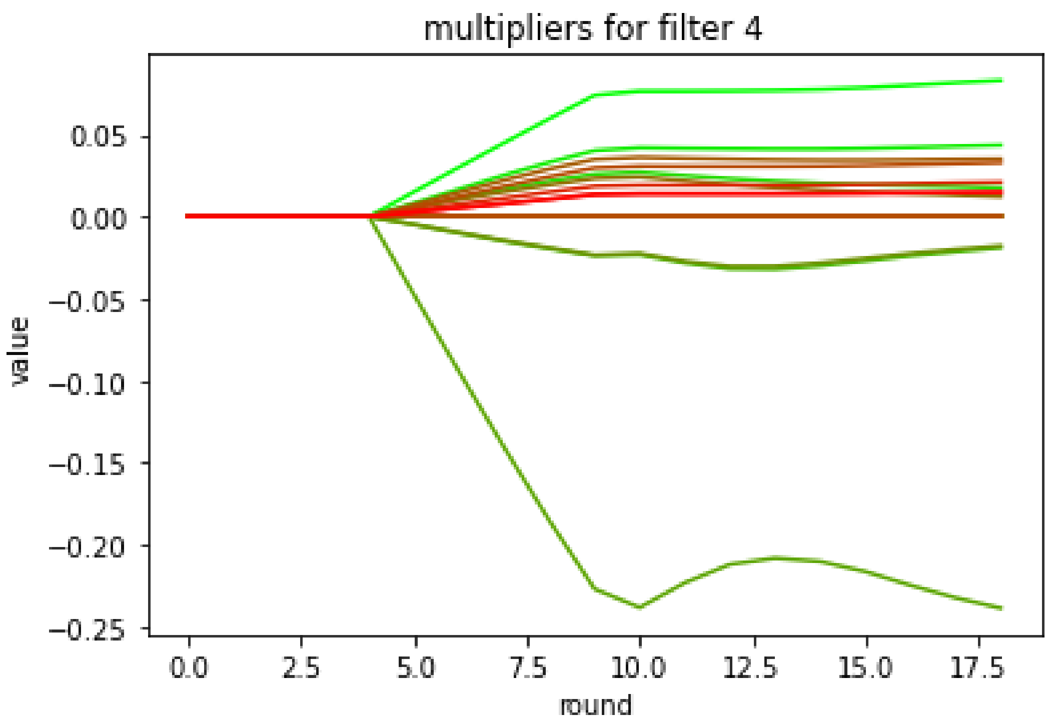


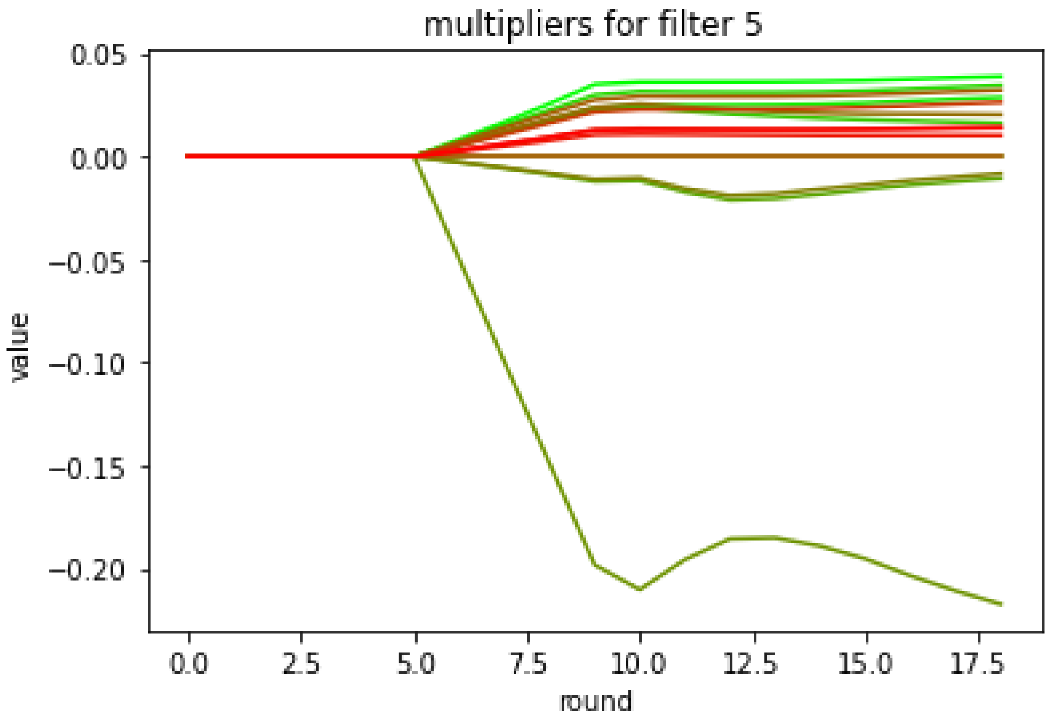
[ ]:
```
num_filters, num_bins, num_rounds = multipliers_traj_fur.shape

colors = np.zeros((num_bins, 3))
colors[:, 0] = np.linspace(0, 1, num_bins)
colors[:, 1] = np.linspace(1, 0, num_bins)

for i in range(num_filters):
    plt.figure(i)
    for j in range(num_bins):
        plt.title(f'multipliers for filter {i+1}')
        plt.xlabel('round')
        plt.ylabel('value')
        plt.plot(np.arange(num_rounds), multipliers_traj_fur[i, j], c=colors[j])
```

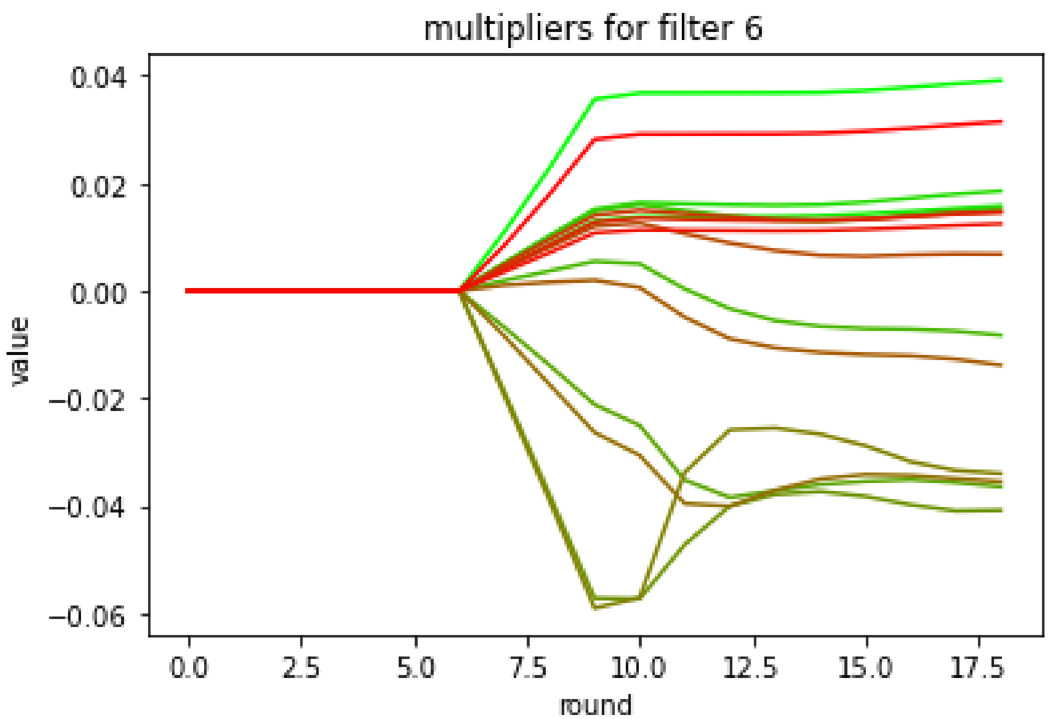

multipliers for filter 1
value
round
0.2
0.1
0.0
−0.1
−0.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

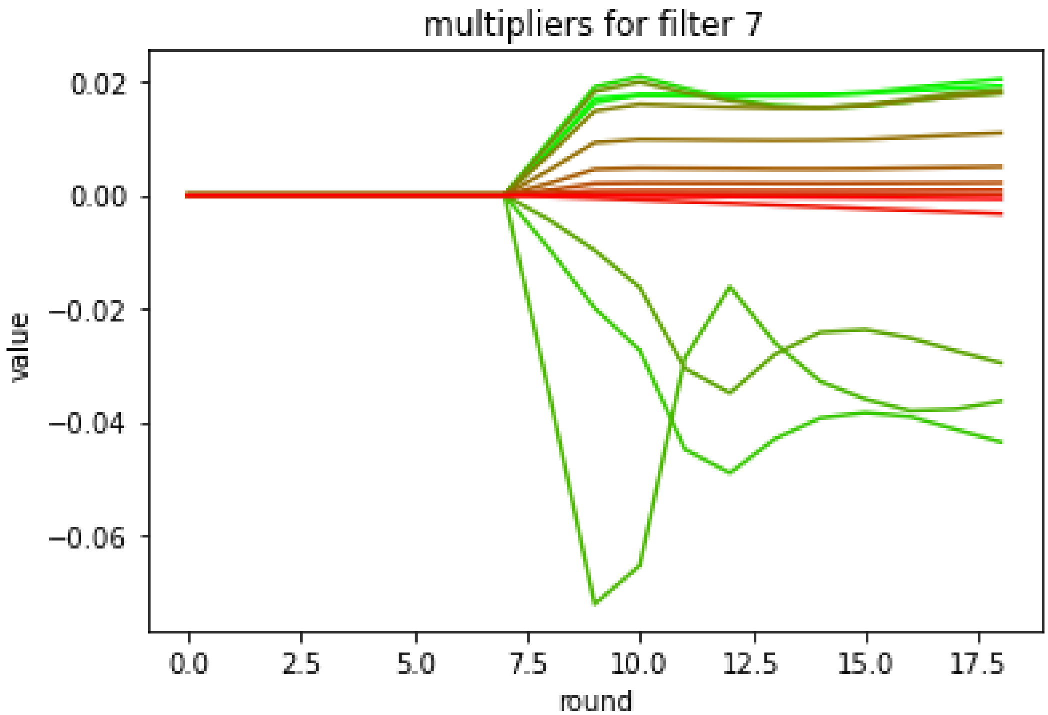

multipliers for filter 2
value
round
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

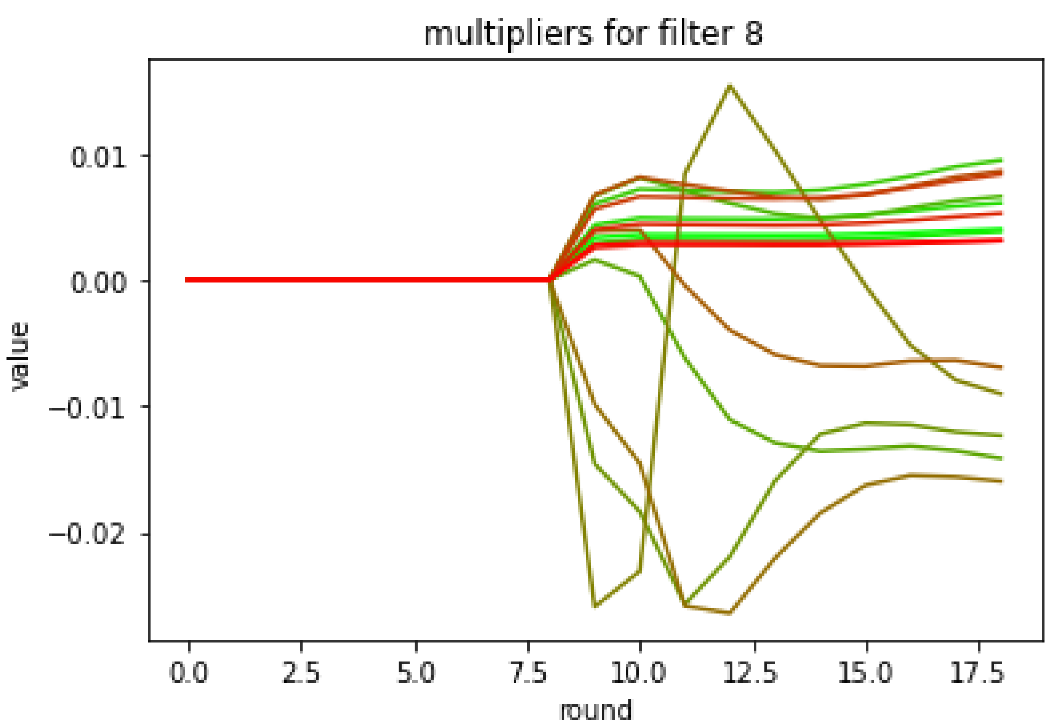

multipliers for filter 3
value
round
0.10
0.05
0.00
−0.05
−0.10
−0.15
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

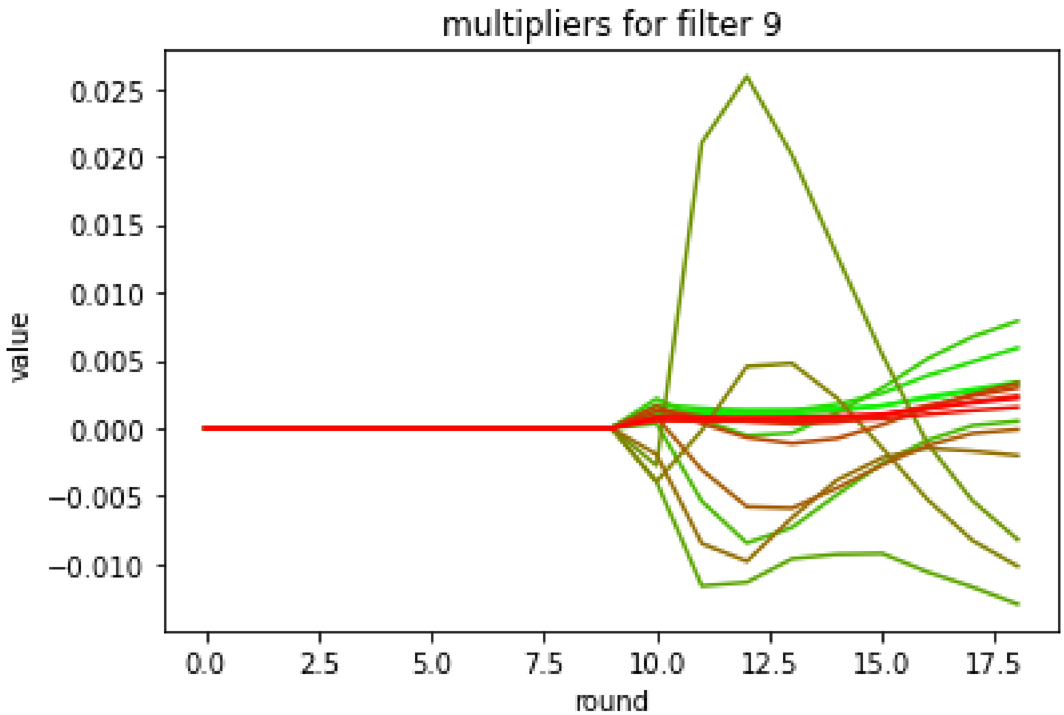

multipliers for filter 4
value
round
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

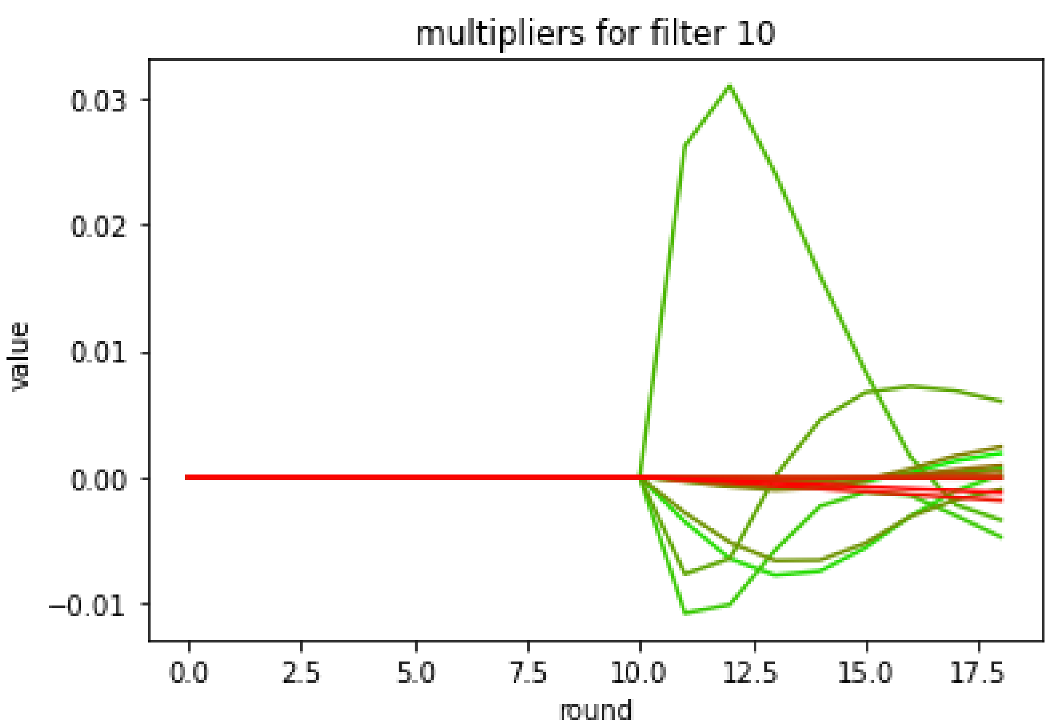

multipliers for filter 5
0.05
0.00
−0.05
−0.10
−0.15
−0.20
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

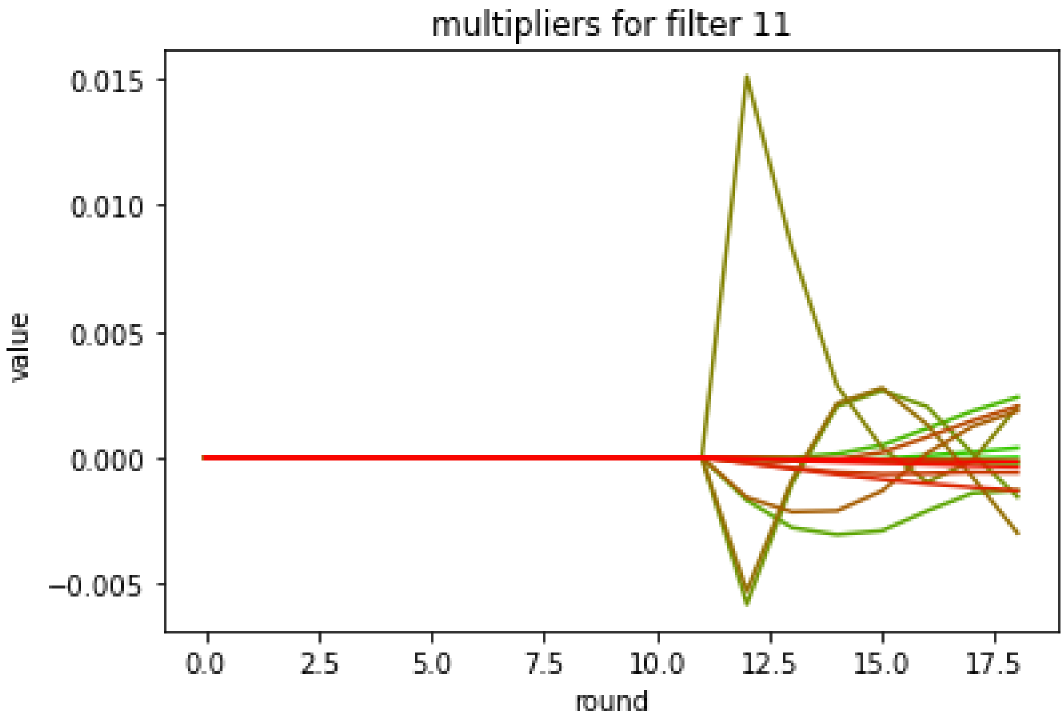

multipliers for filter 6
0.04
0.02
0.00
−0.02
−0.04
−0.06
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

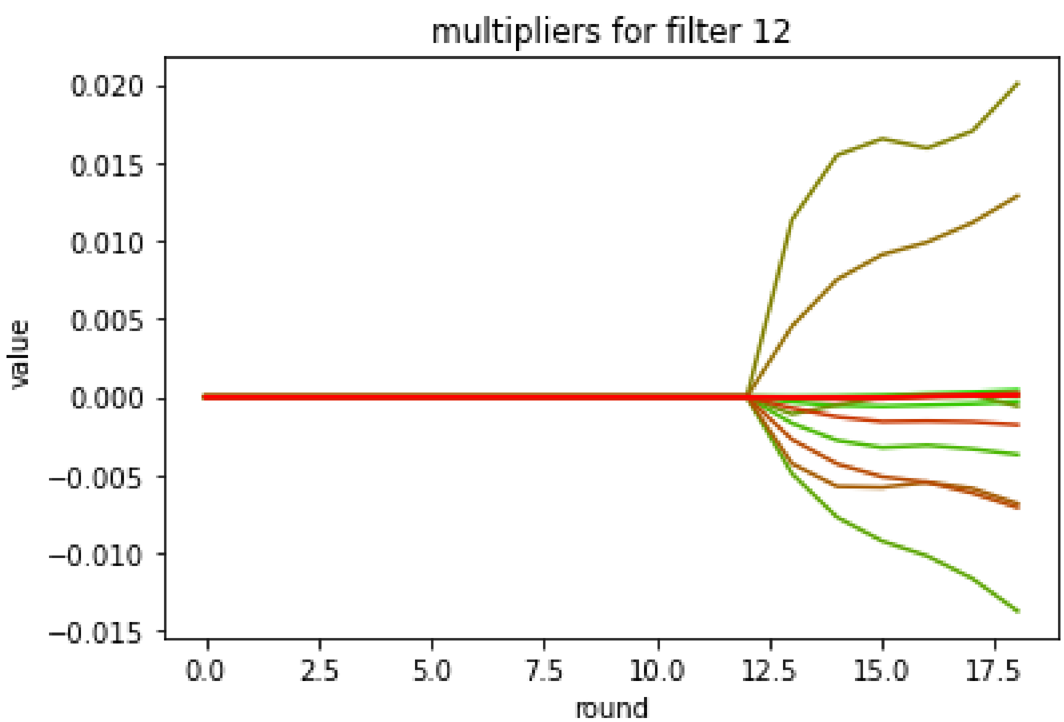

multipliers for filter 7
0.02
0.00
−0.02
−0.04
−0.06
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

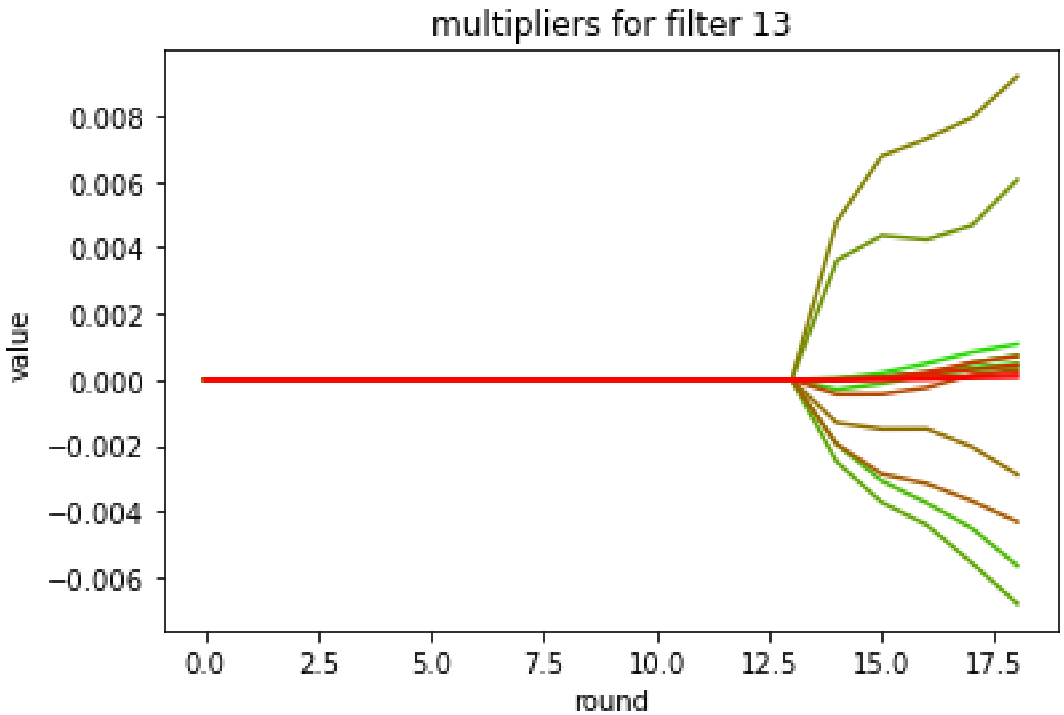

multipliers for filter 8
0.01
0.00
−0.01
−0.02
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

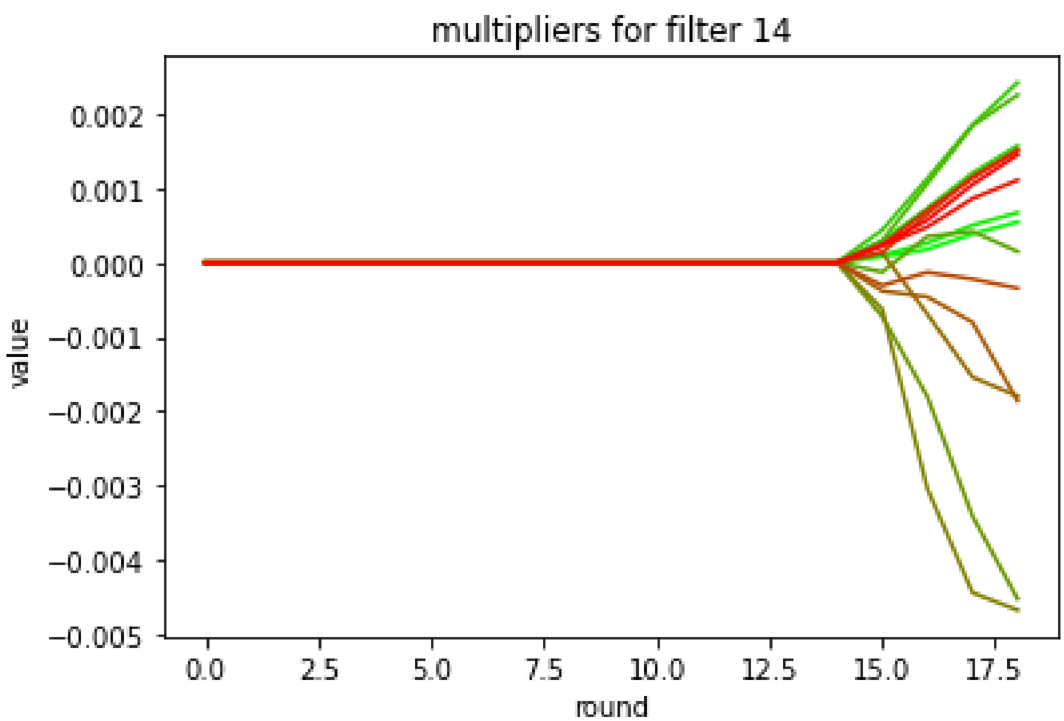

multipliers for filter 9
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

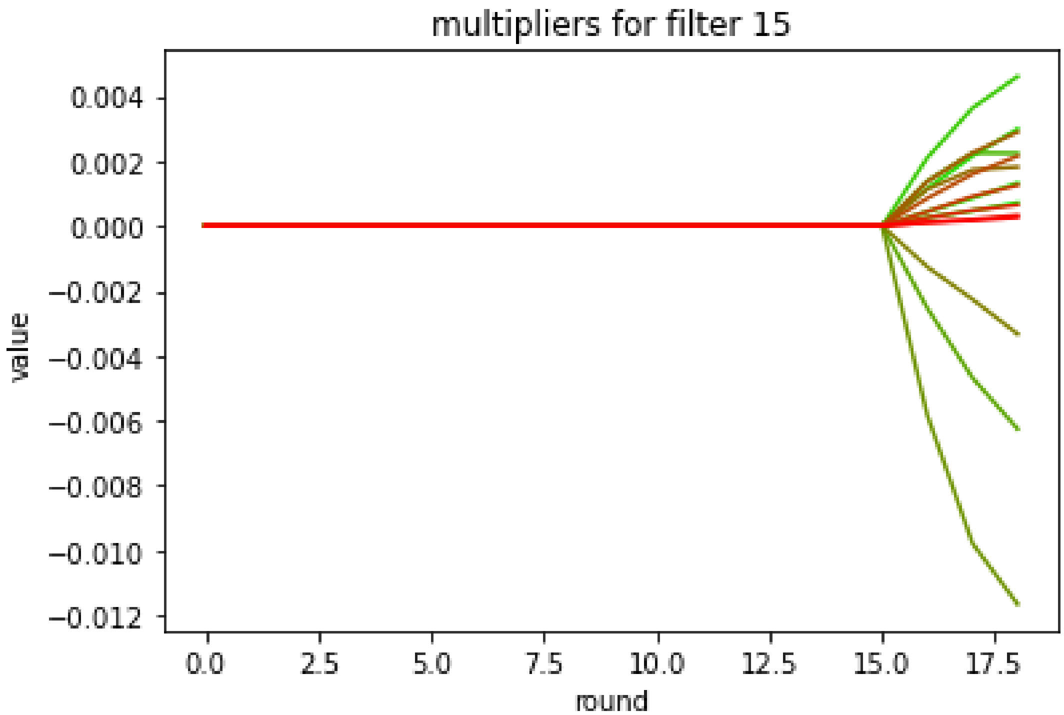

multipliers for filter 10
0.03
0.02
0.01
0.00
−0.01
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

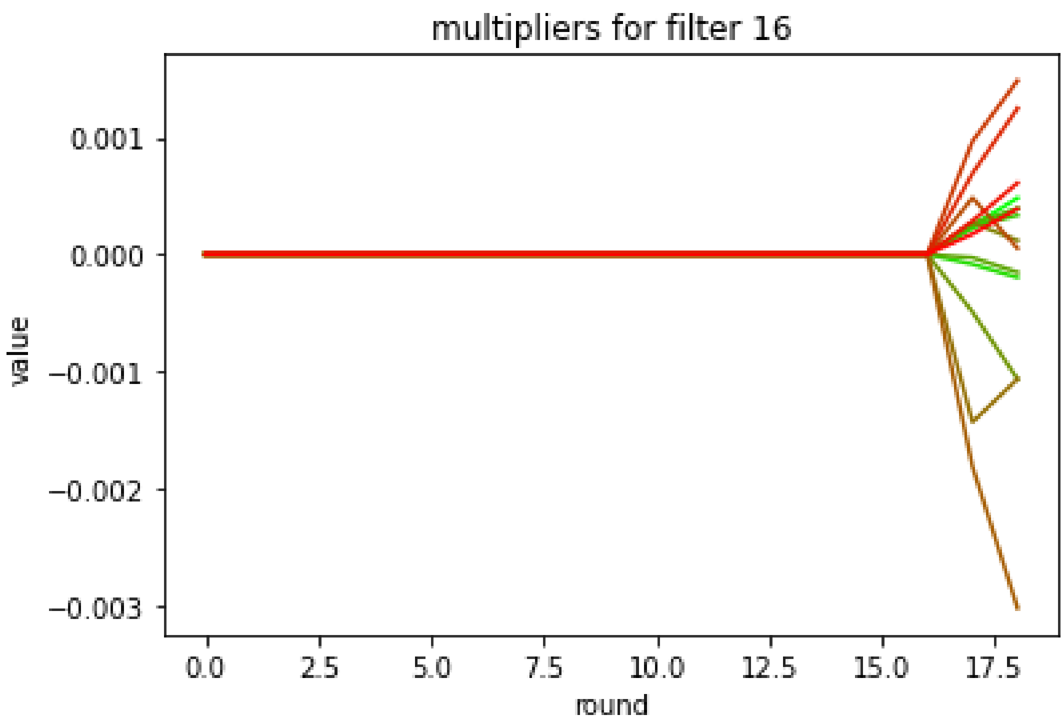

multipliers for filter 11
0.015
0.010
0.005
0.000
−0.005
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

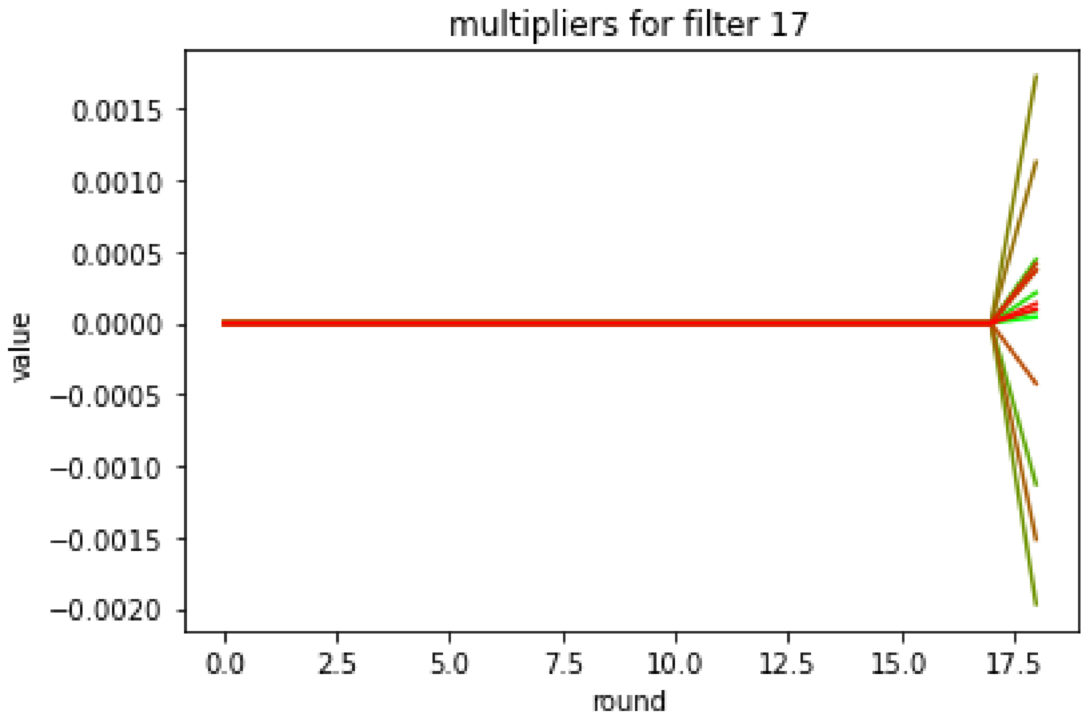

multipliers for filter 12
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

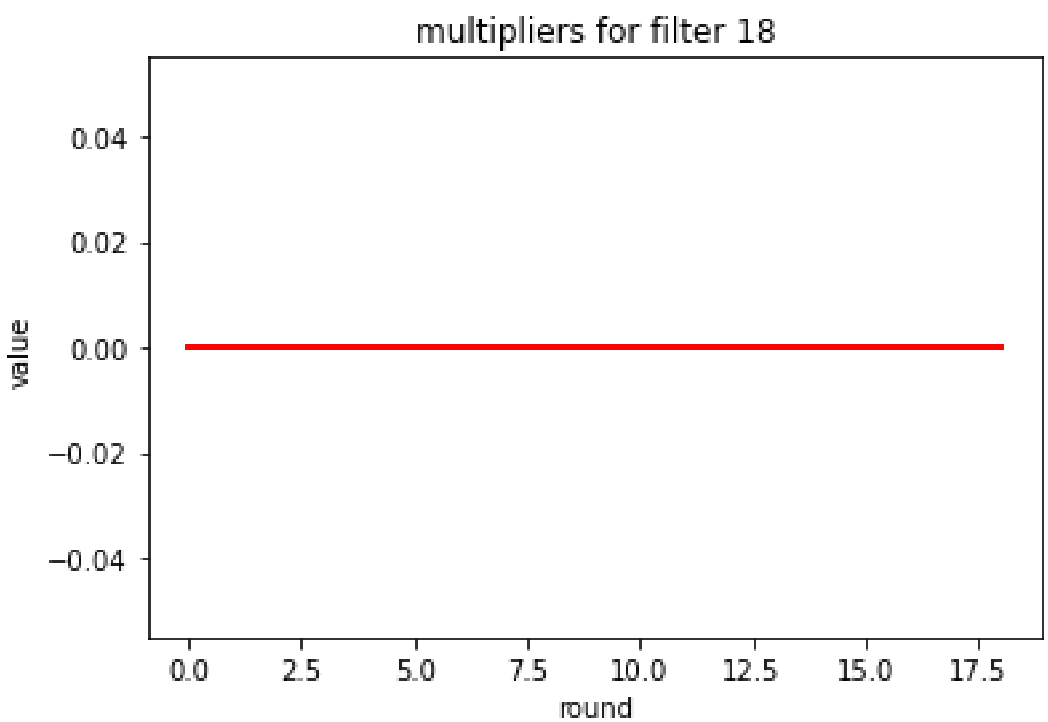

multipliers for filter 13
value
round
0.008
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


multipliers for filter 14
value
round
0.002
0.001
0.000
−0.001
−0.002
−0.003
−0.004
−0.005
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


multipliers for filter 15
value
0.004
0.002
0.000
−0.002
−0.004
−0.006
−0.008
−0.010
−0.012
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

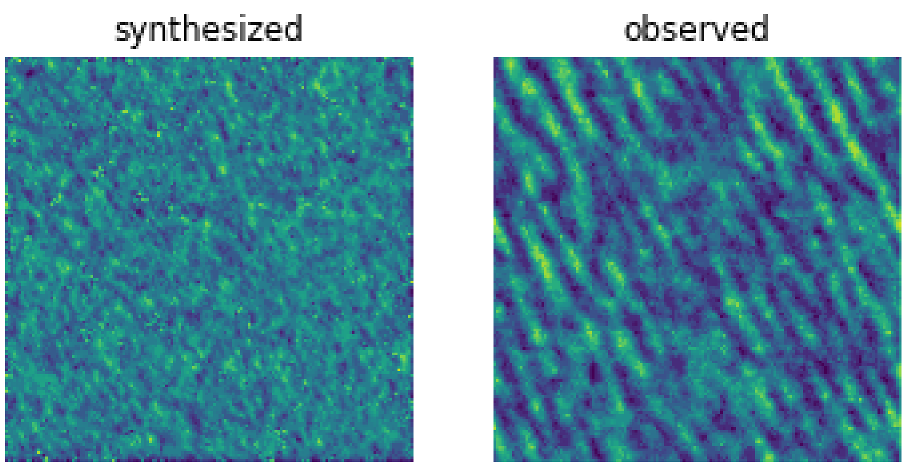

multipliers for filter 16
value
0.001
0.000
−0.001
−0.002
−0.003
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 17
0.0015
0.0010
0.0005
0.0000
−0.0005
−0.0010
−0.0015
−0.0020
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 18
0.04
0.02
0.00
−0.02
−0.04
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

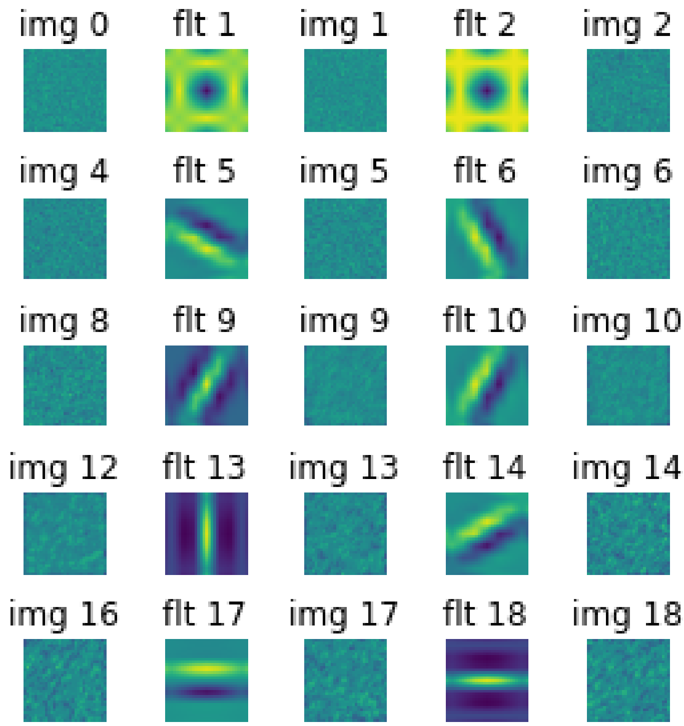
[ ]:
```
print('Final result of FRAME on fur texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_fur_2[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_fur/8)
plt.axis('off')
plt.show()
```

```
Final result of FRAME on fur texture
```

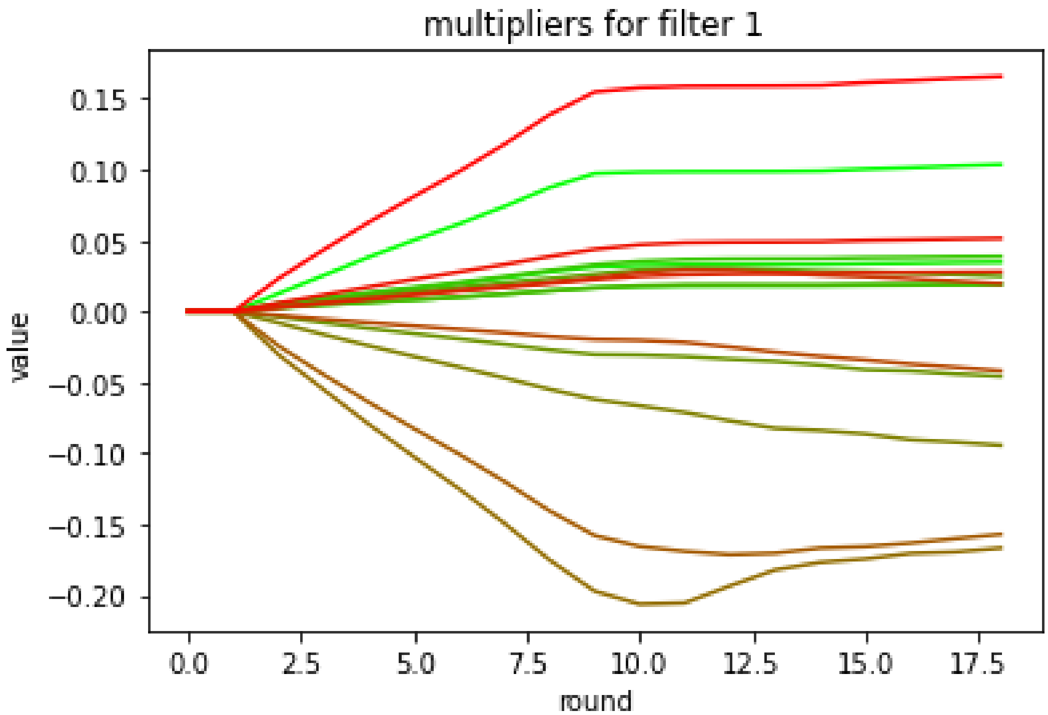


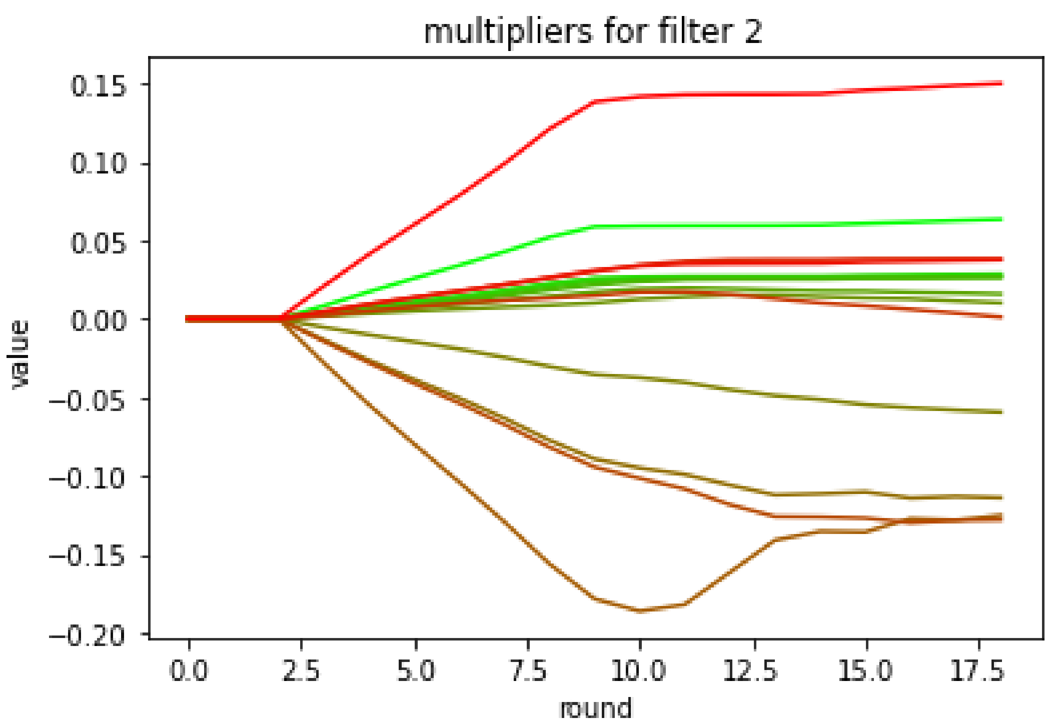
[ ]:
```
print('Evolution of filters and synthesized samples on stucco texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_stucco_2],␣
 ↪imgs_syn=imgs_syn_stucco_2)
```

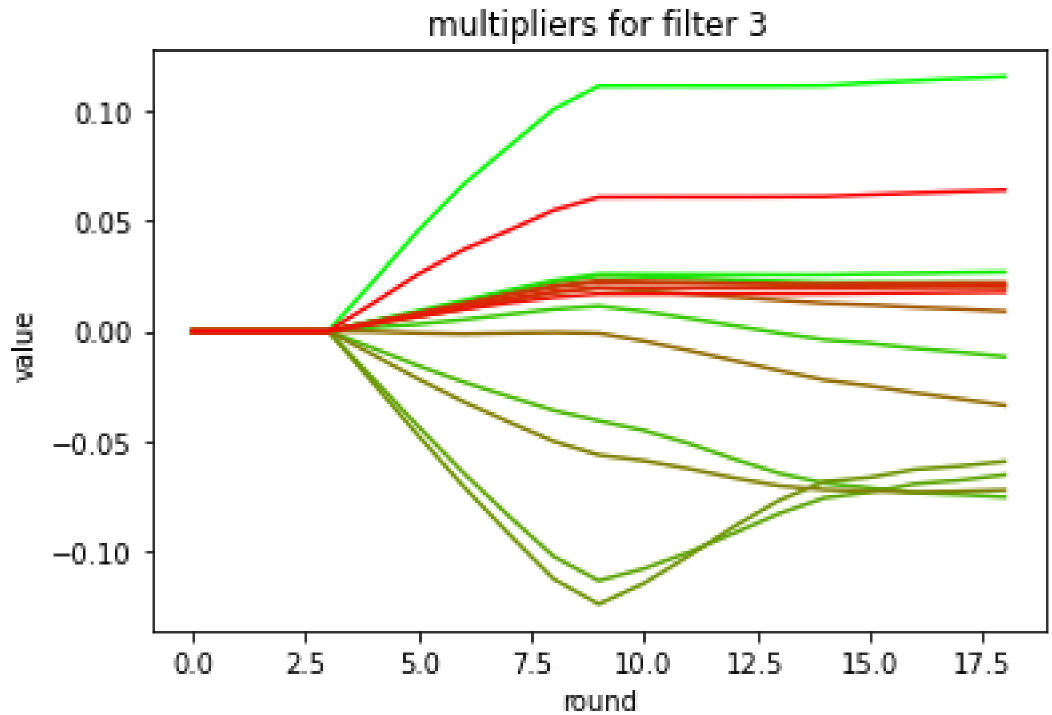
```
Evolution of filters and synthesized samples on stucco texture
```

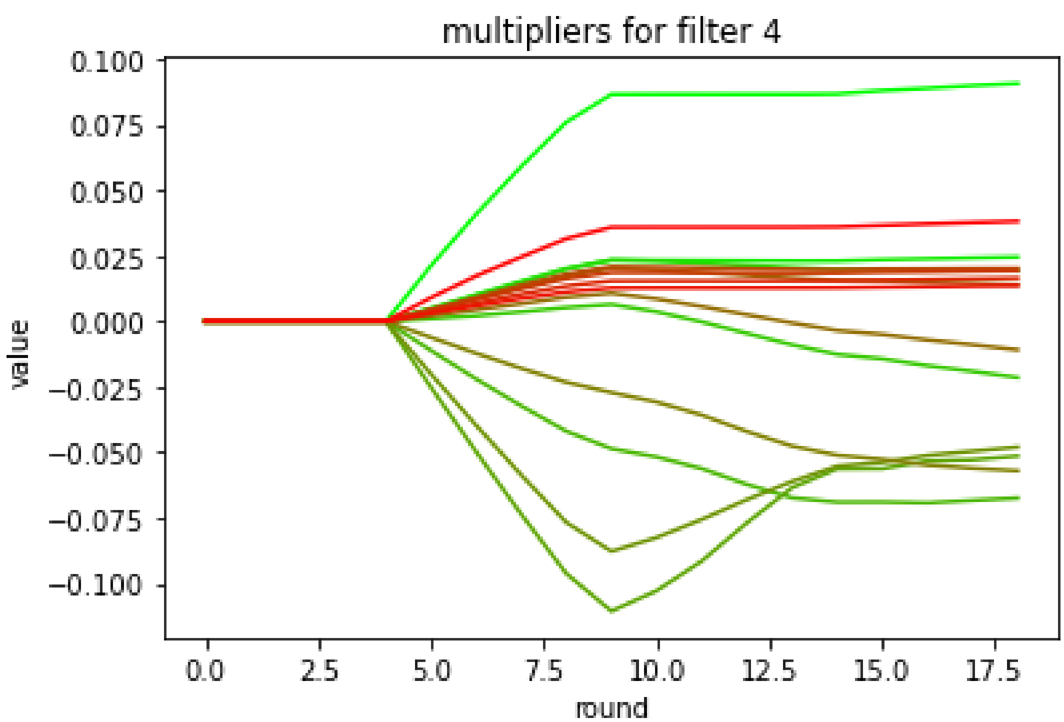


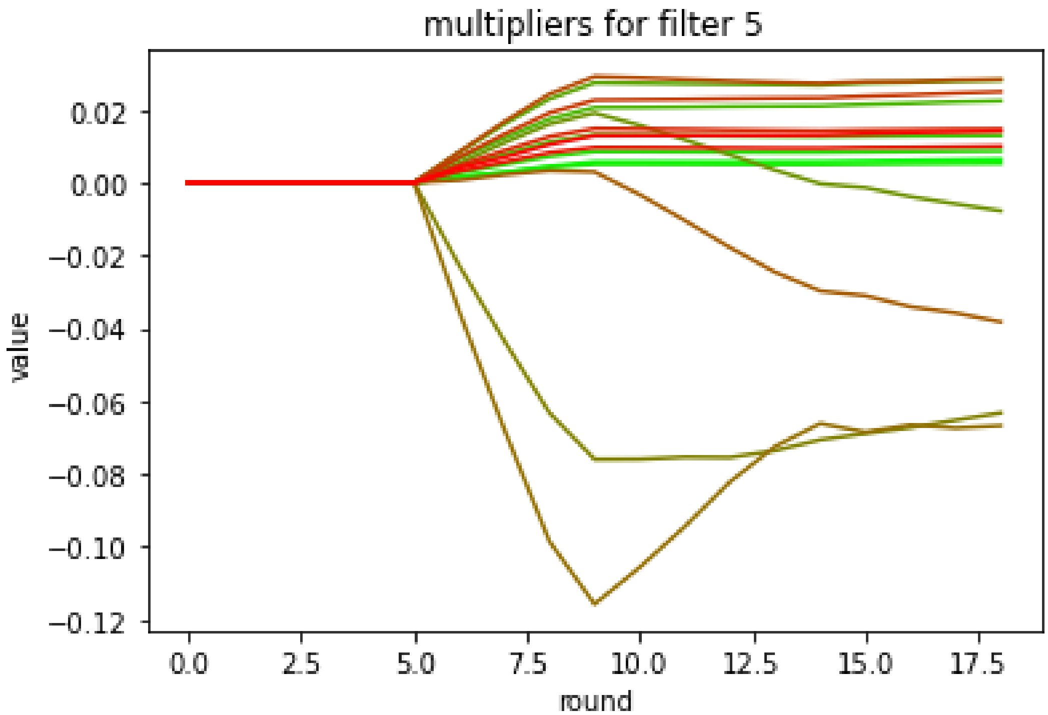

```
[ ]: num_filters, num_bins, num_rounds = multipliers_traj_stucco.shape

     colors = np.zeros((num_bins, 3))
     colors[:, 0] = np.linspace(0, 1, num_bins)
     colors[:, 1] = np.linspace(1, 0, num_bins)

     for i in range(num_filters):
         plt.figure(i)
         for j in range(num_bins):
             plt.title(f'multipliers for filter {i+1}')
             plt.xlabel('round')
             plt.ylabel('value')
             plt.plot(np.arange(num_rounds), multipliers_traj_stucco[i, j],␣
      ↪c=colors[j])
```

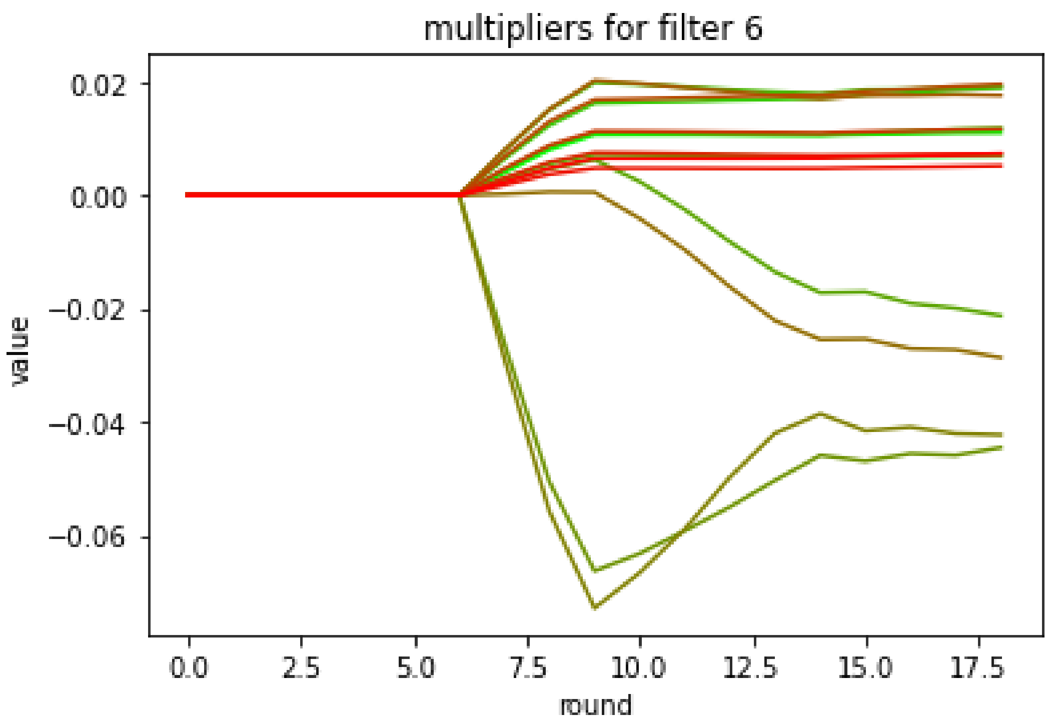

multipliers for filter 1
value
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

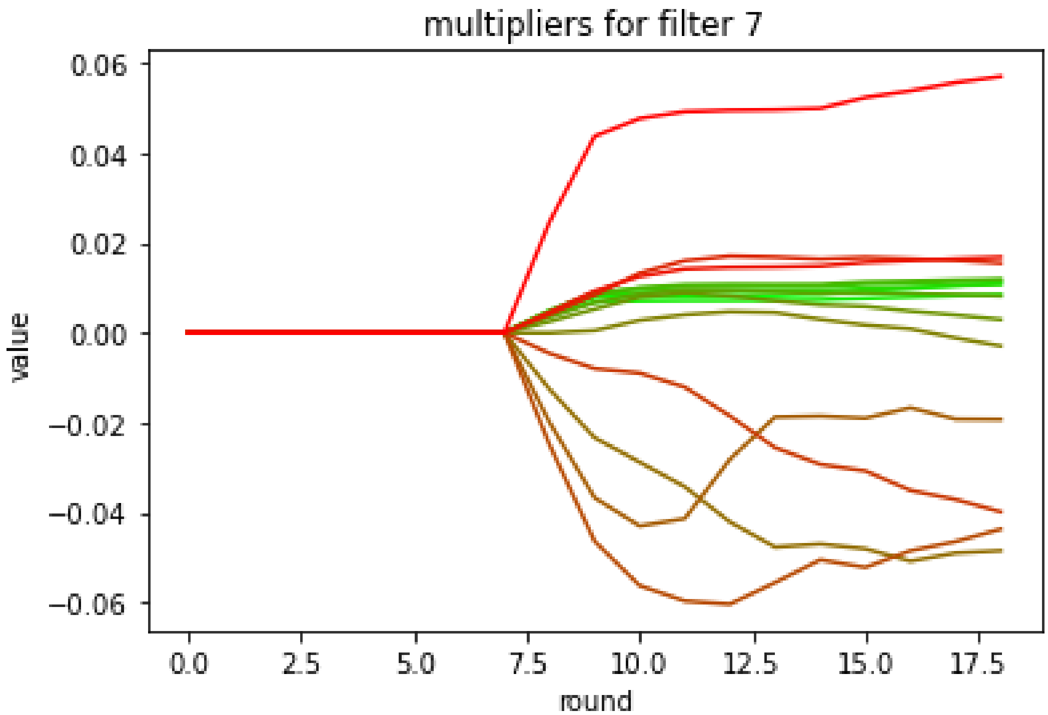

multipliers for filter 2
value
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

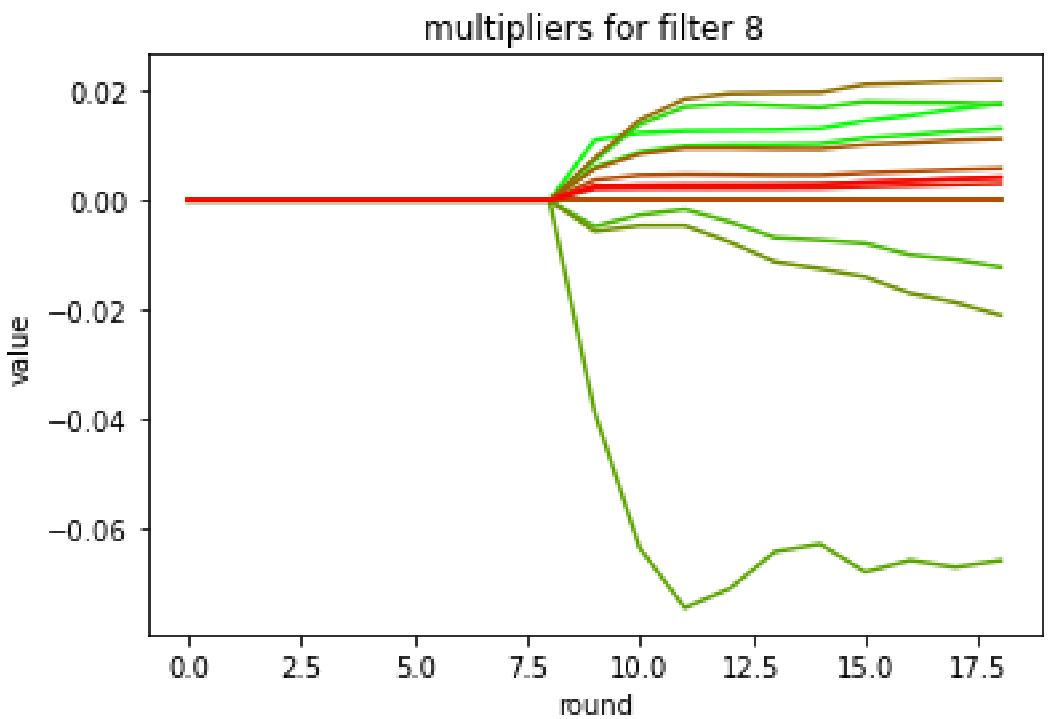

multipliers for filter 3
value
round
0.10
0.05
0.00
−0.05
−0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

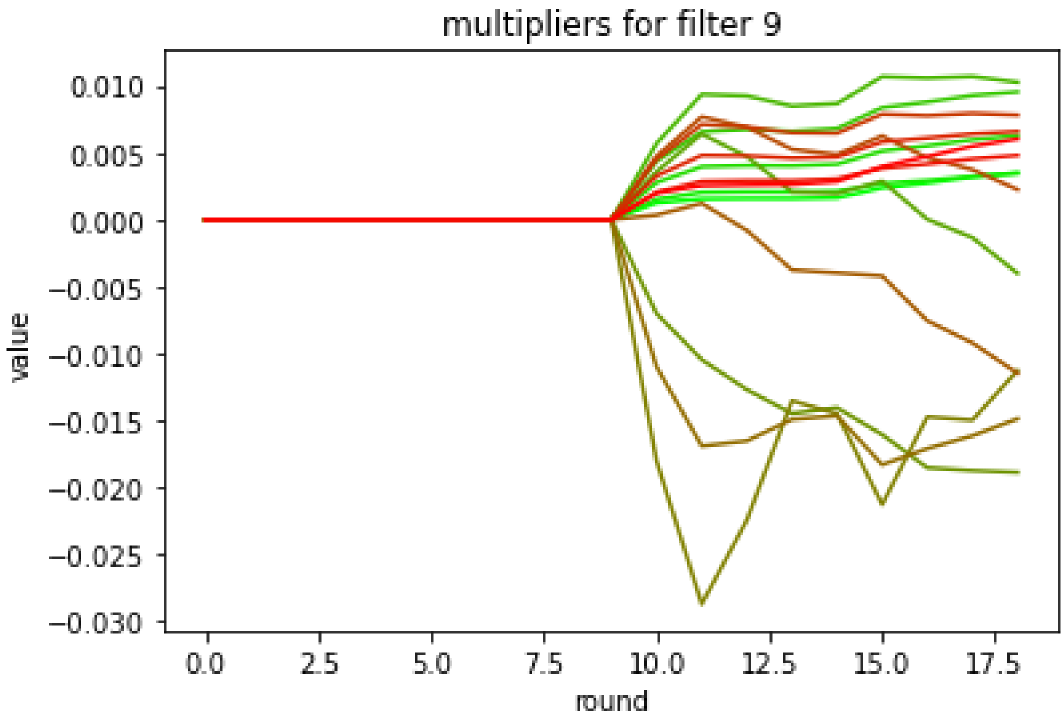

multipliers for filter 4
value
round
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


multipliers for filter 5
value
round
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

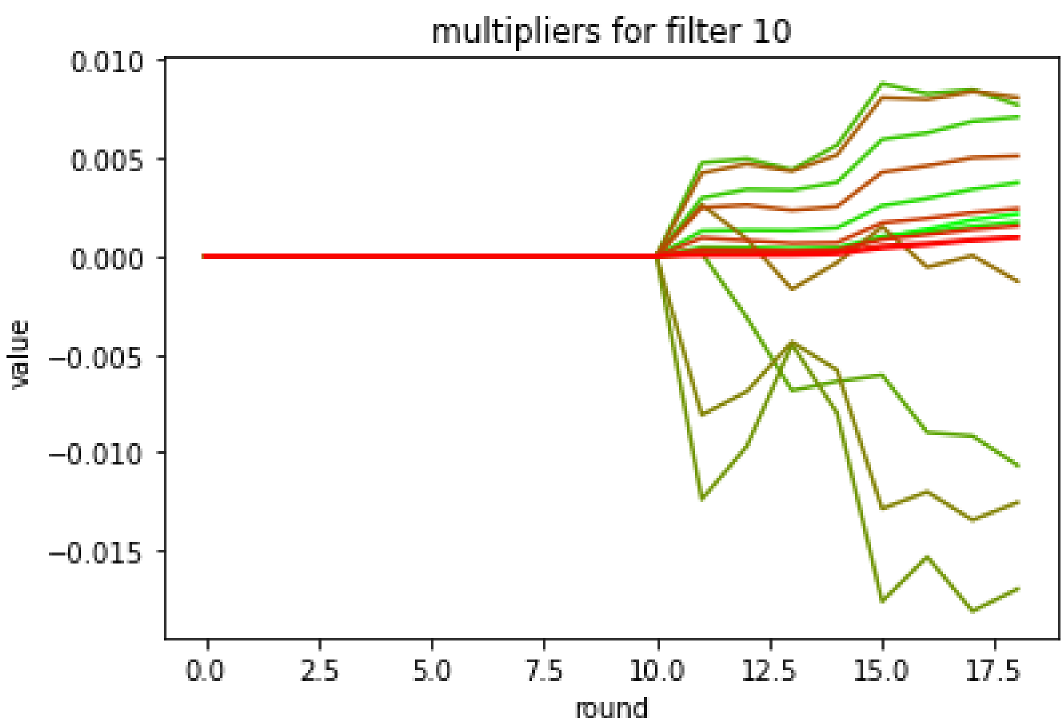

multipliers for filter 6
value
round
0.02
0.00
−0.02
−0.04
−0.06
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


multipliers for filter 7
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

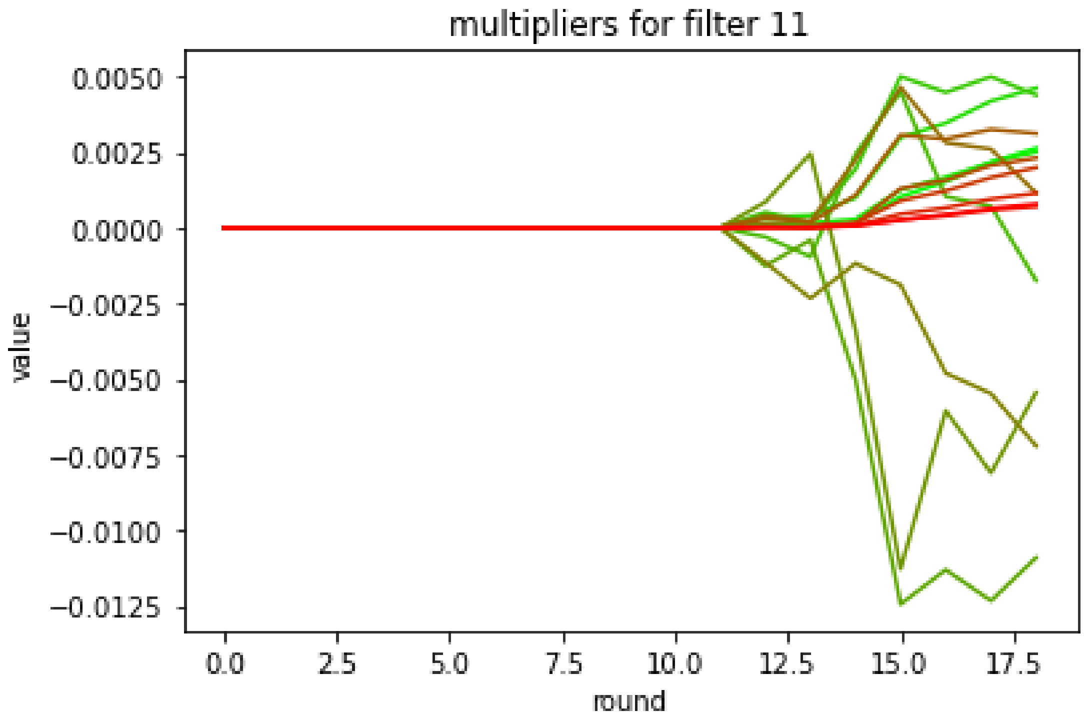

multipliers for filter 8
0.02
0.00
−0.02
−0.04
−0.06
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

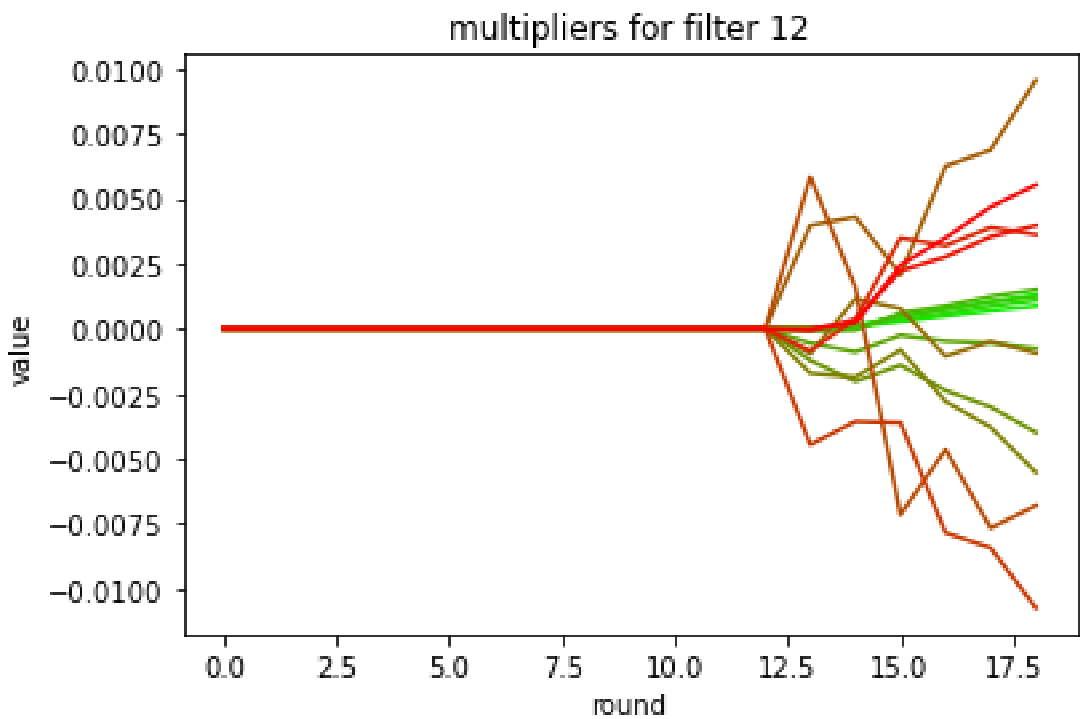


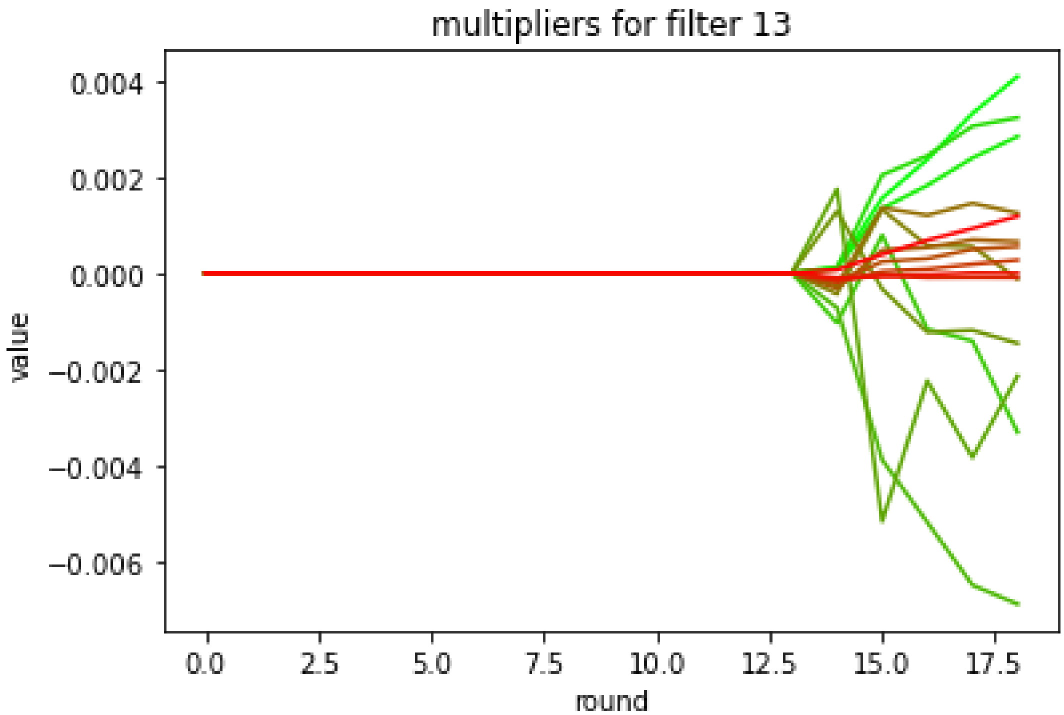
multipliers for filter 9
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−0.025
−0.030
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

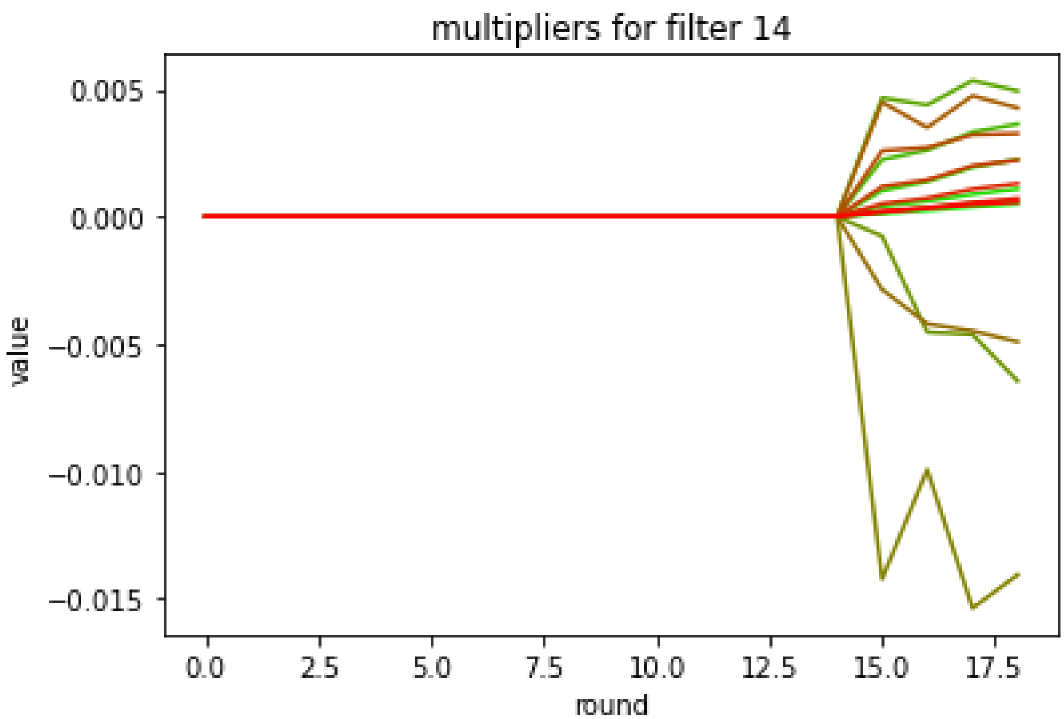


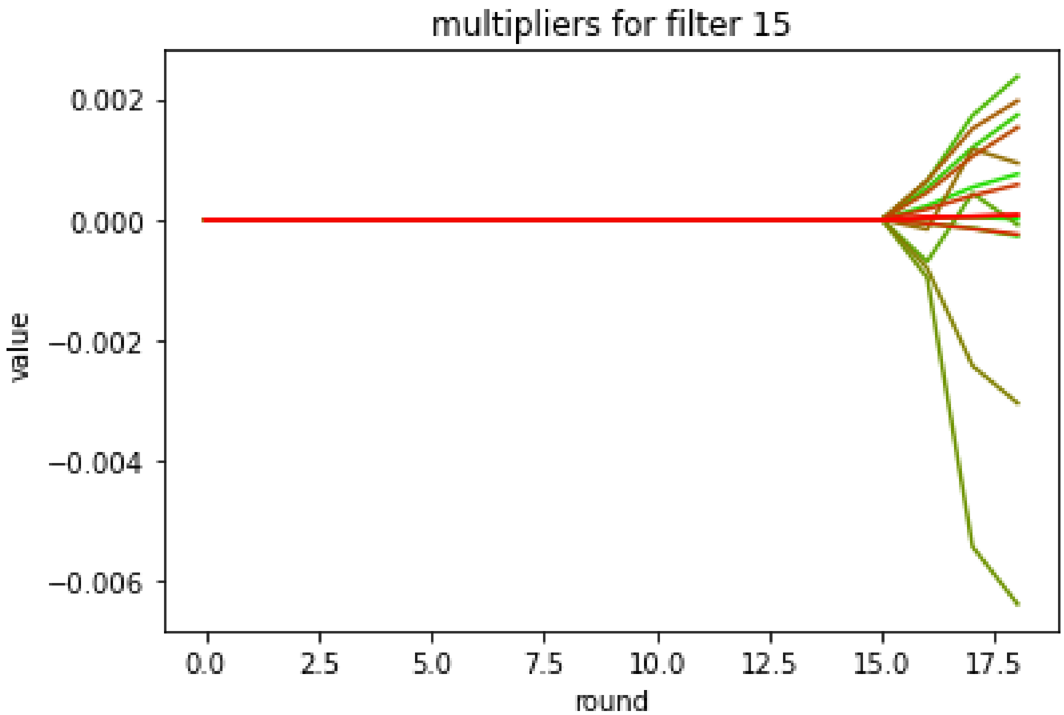
multipliers for filter 10
0.010
0.005
0.000
−0.005
−0.010
−0.015
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

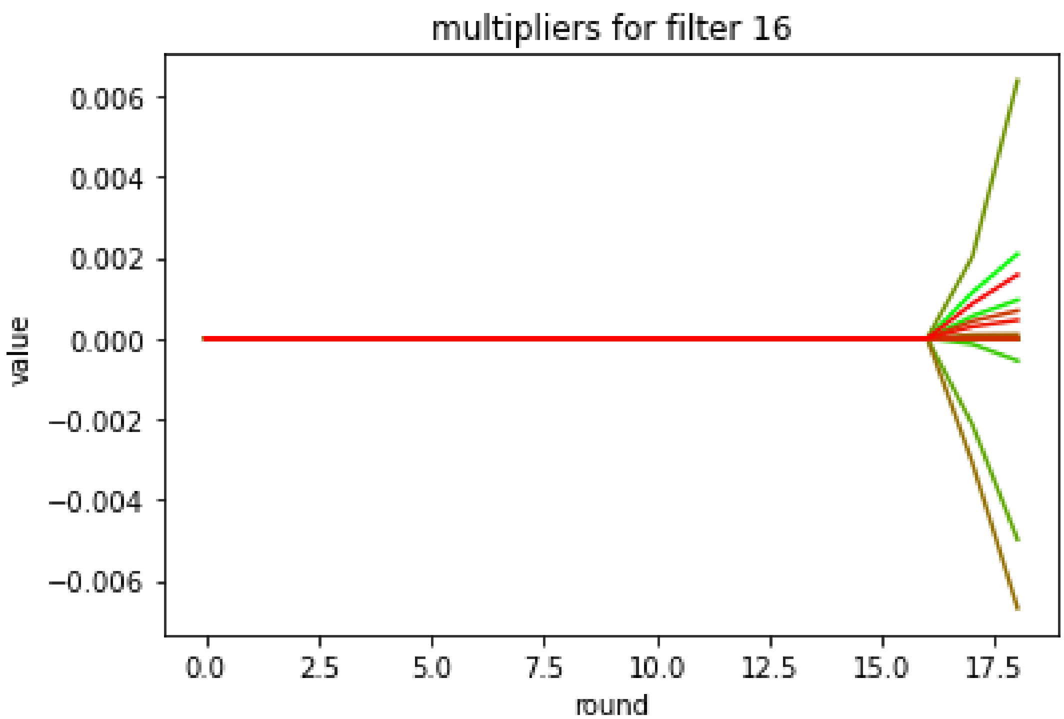

multipliers for filter 11
value
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

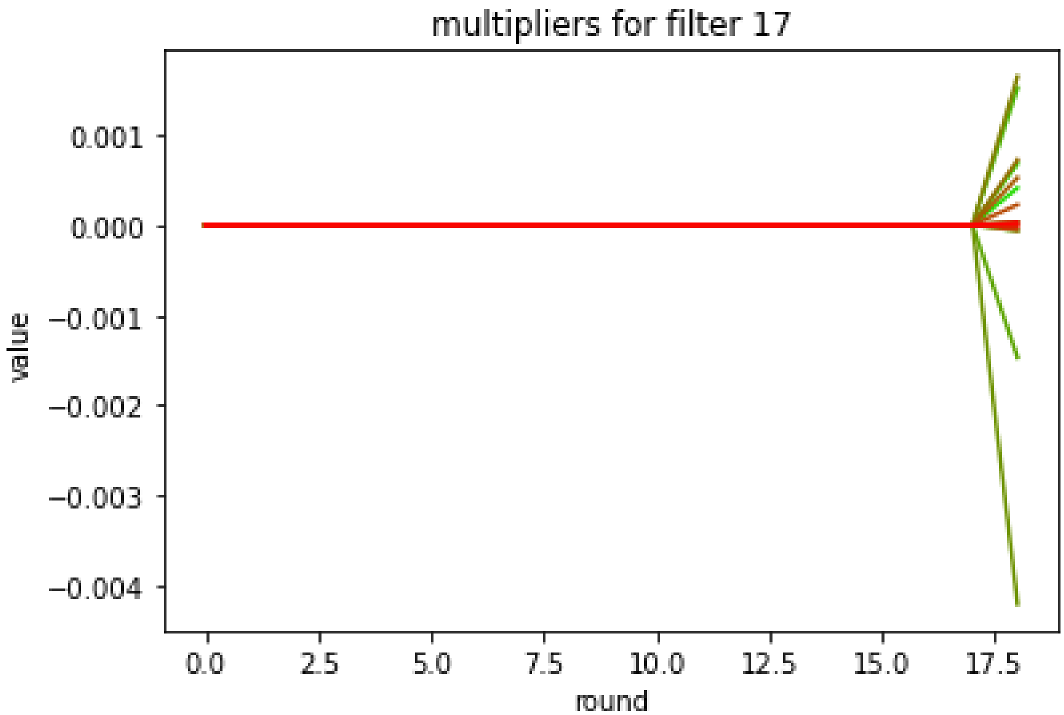

multipliers for filter 12
value
0.0100
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 13
0.004
0.002
0.000
−0.002
−0.004
−0.006
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

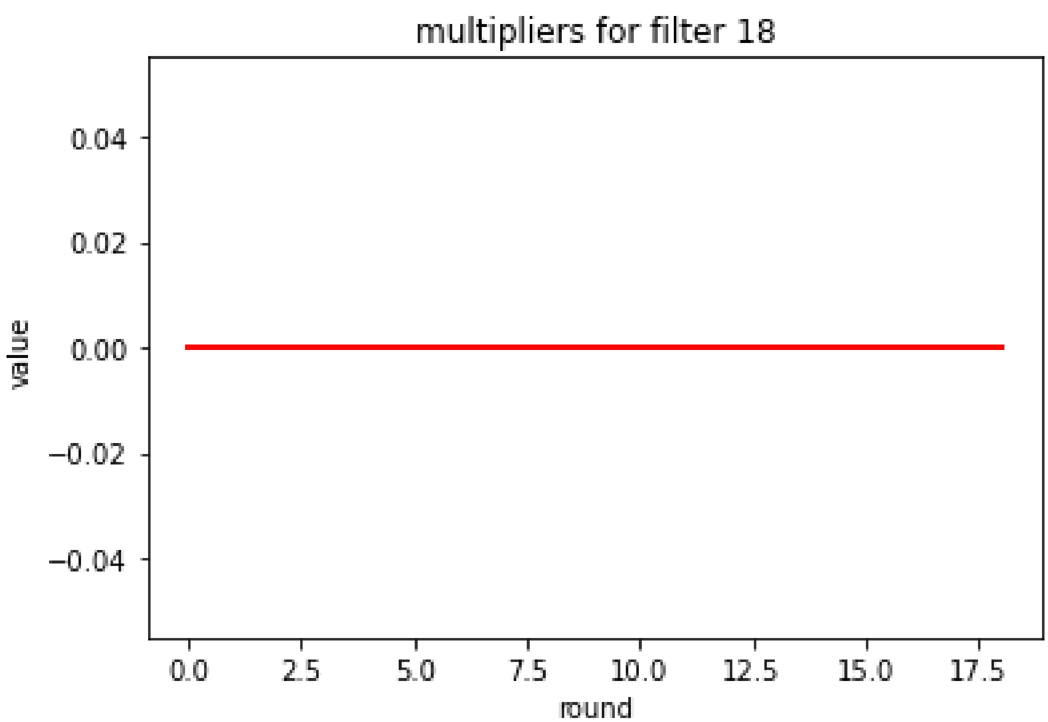

multipliers for filter 14
0.005
0.000
−0.005
−0.010
−0.015
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 15
0.002
0.000
−0.002
−0.004
−0.006
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 16
0.006
0.004
0.002
0.000
−0.002
−0.004
−0.006
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round



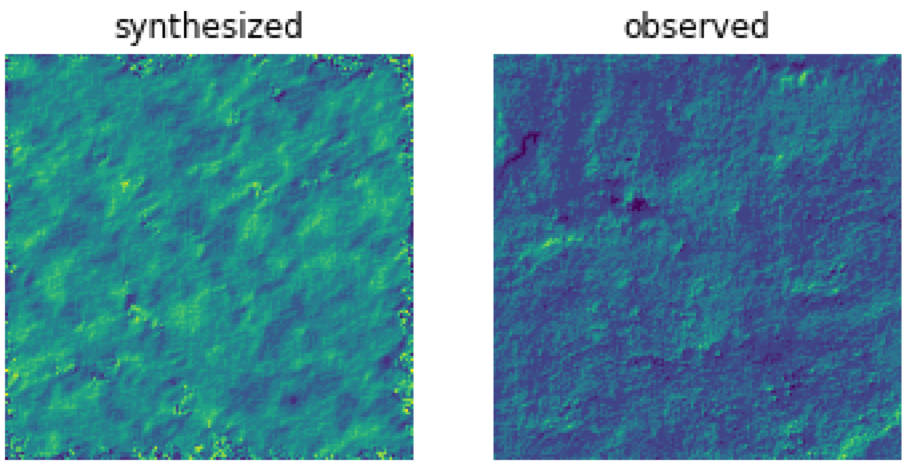
multipliers for filter 17
value
0.001
0.000
−0.001
−0.002
−0.003
−0.004
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round



multipliers for filter 18
value
0.04
0.02
0.00
−0.02
−0.04
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

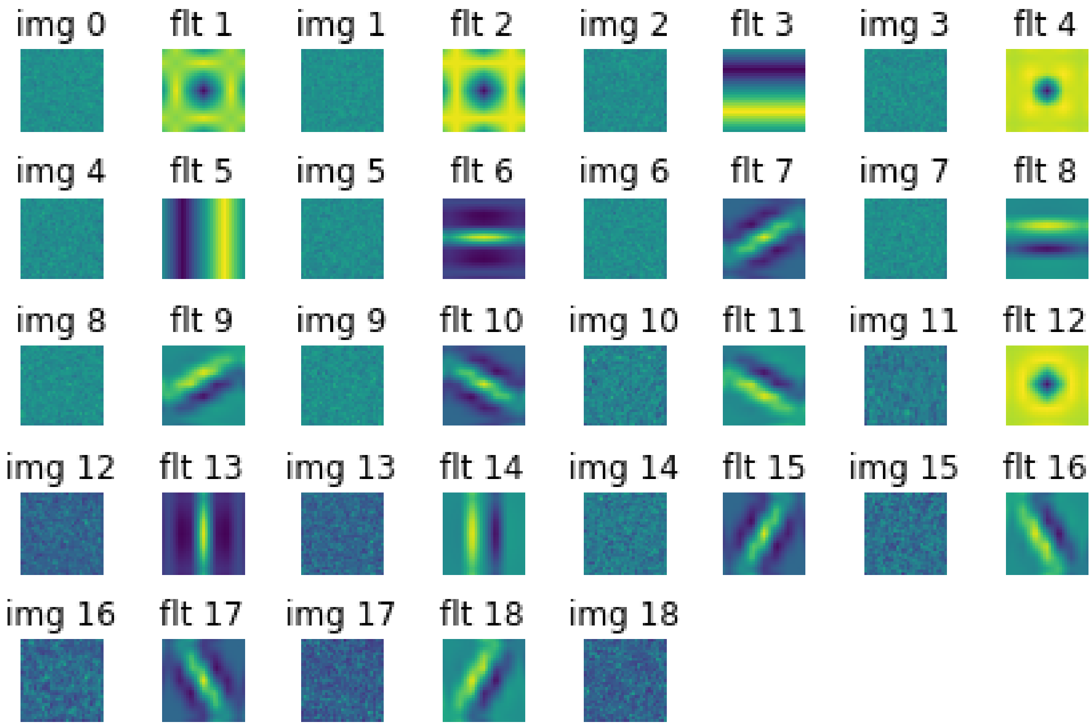
[ ]:
```
print('Final result of FRAME on stucco texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_stucco_2[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_stucco/8)
plt.axis('off')
plt.show()
```

```
Final result of FRAME on stucco texture
```

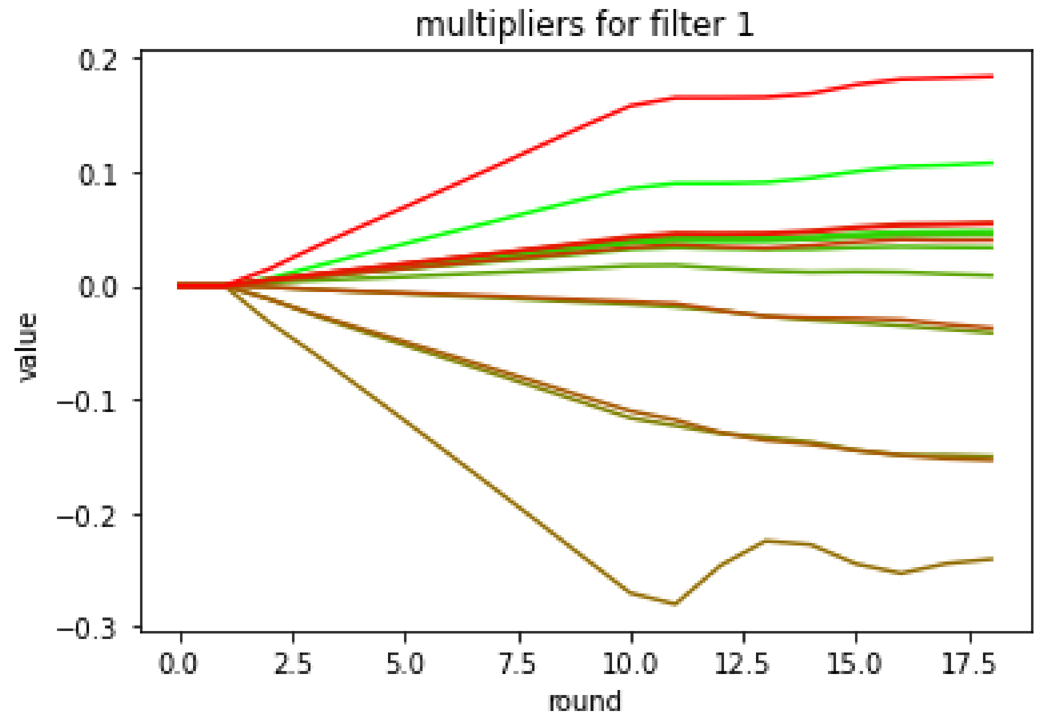


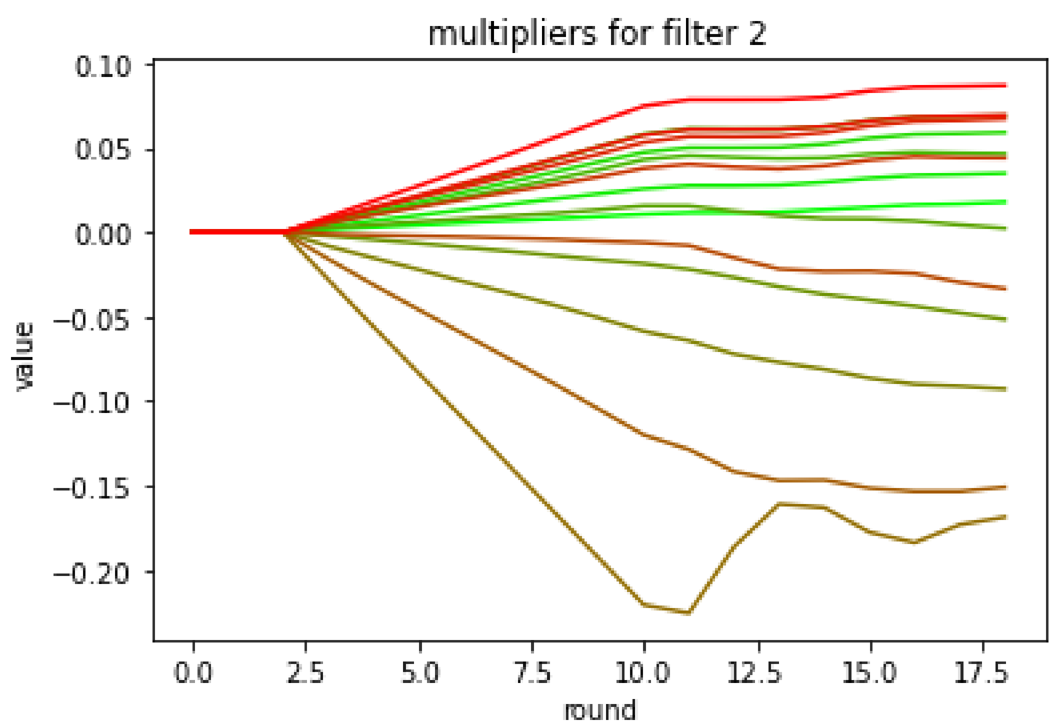
[ ]:
```
print('Evolution of filters and synthesized samples on grass texture')
visualize_sequence(filters_chosen=[F[i] for i in filters_chosen_idx_grass_2],
 ↪imgs_syn=imgs_syn_grass_2)
```

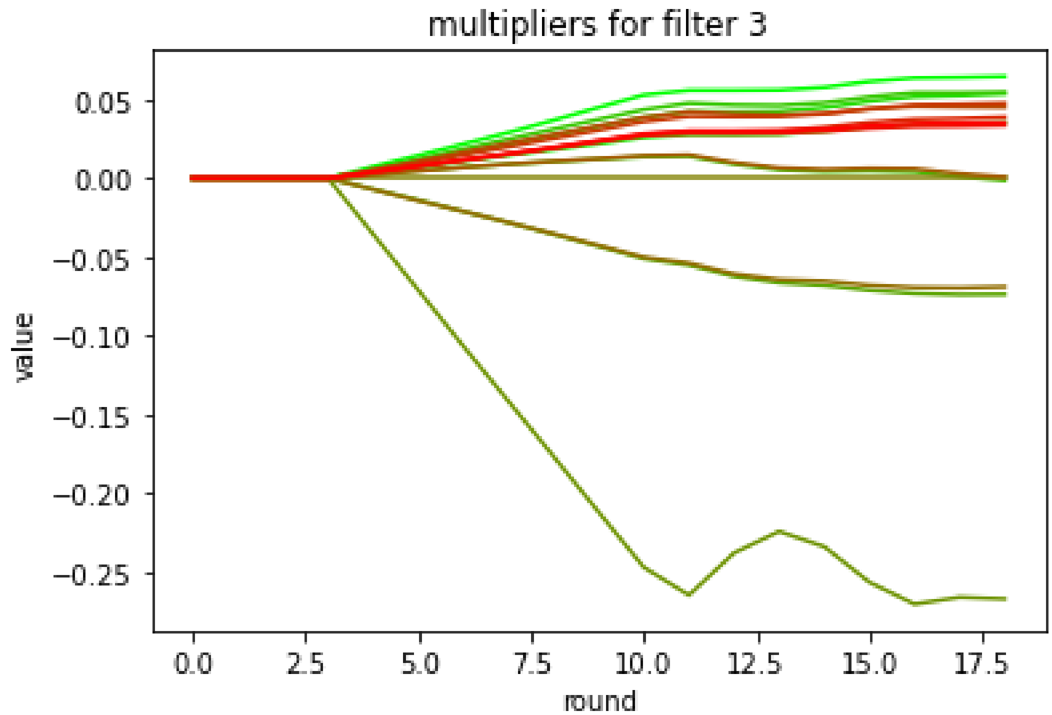
```
Evolution of filters and synthesized samples on grass texture
```

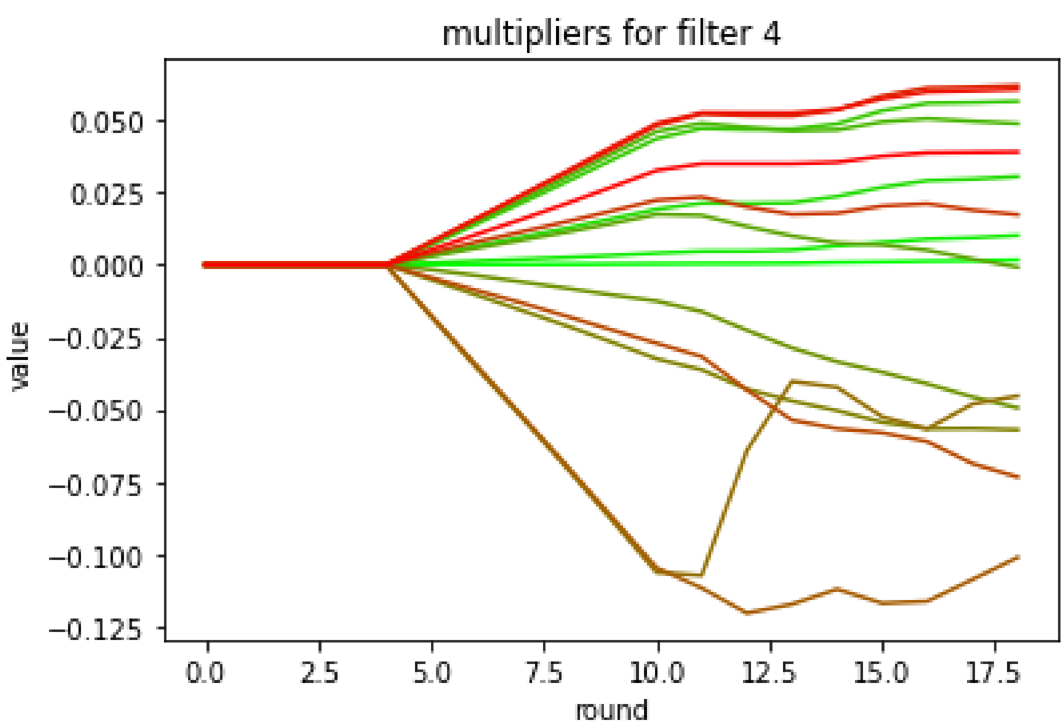


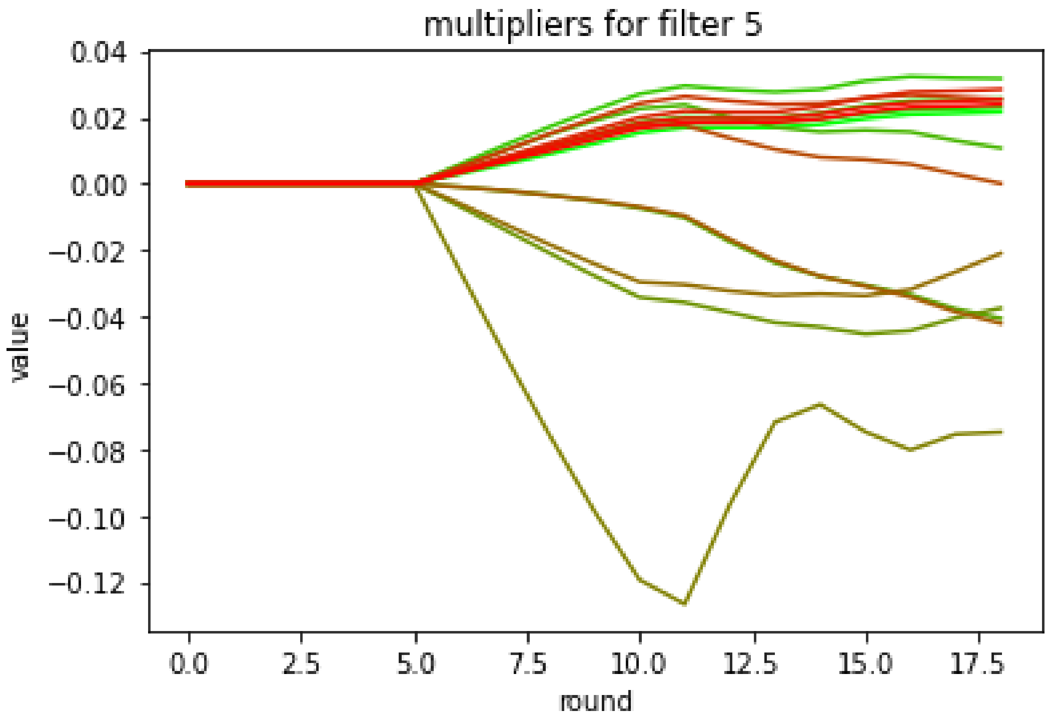
[ ]:
```
num_filters, num_bins, num_rounds = multipliers_traj_grass.shape

colors = np.zeros((num_bins, 3))
colors[:, 0] = np.linspace(0, 1, num_bins)
colors[:, 1] = np.linspace(1, 0, num_bins)

for i in range(num_filters):
    plt.figure(i)
    for j in range(num_bins):
        plt.title(f'multipliers for filter {i+1}')
        plt.xlabel('round')
        plt.ylabel('value')
        plt.plot(np.arange(num_rounds), multipliers_traj_grass[i, j],␣
 ↪c=colors[j])
```

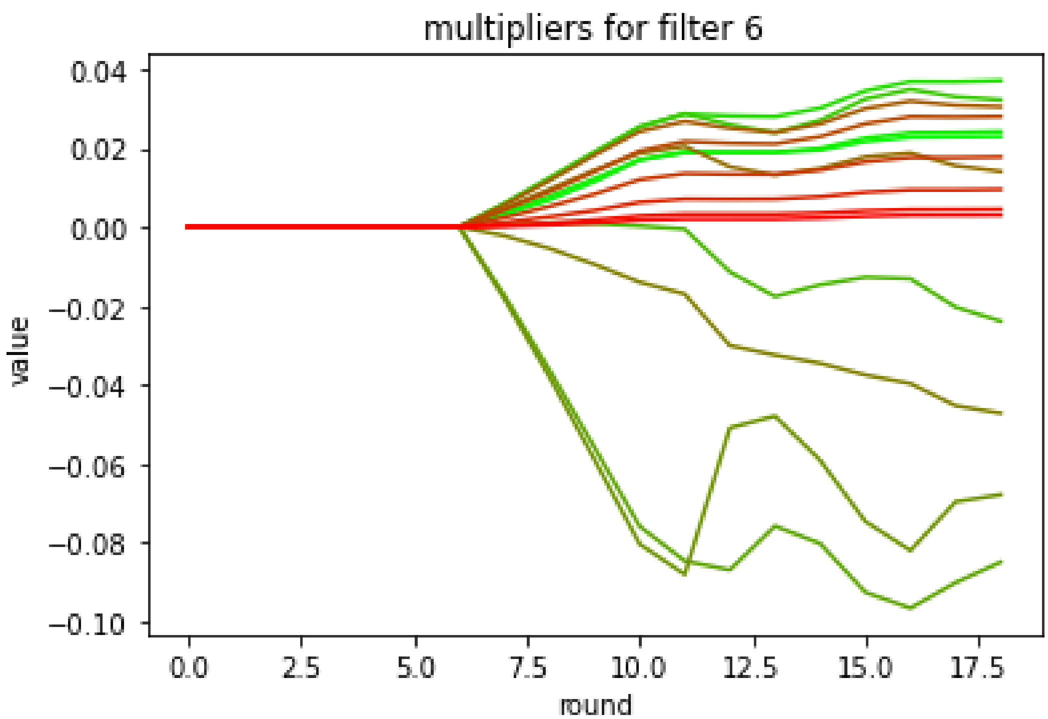

multipliers for filter 1
0.2
0.1
0.0
−0.1
−0.2
−0.3
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

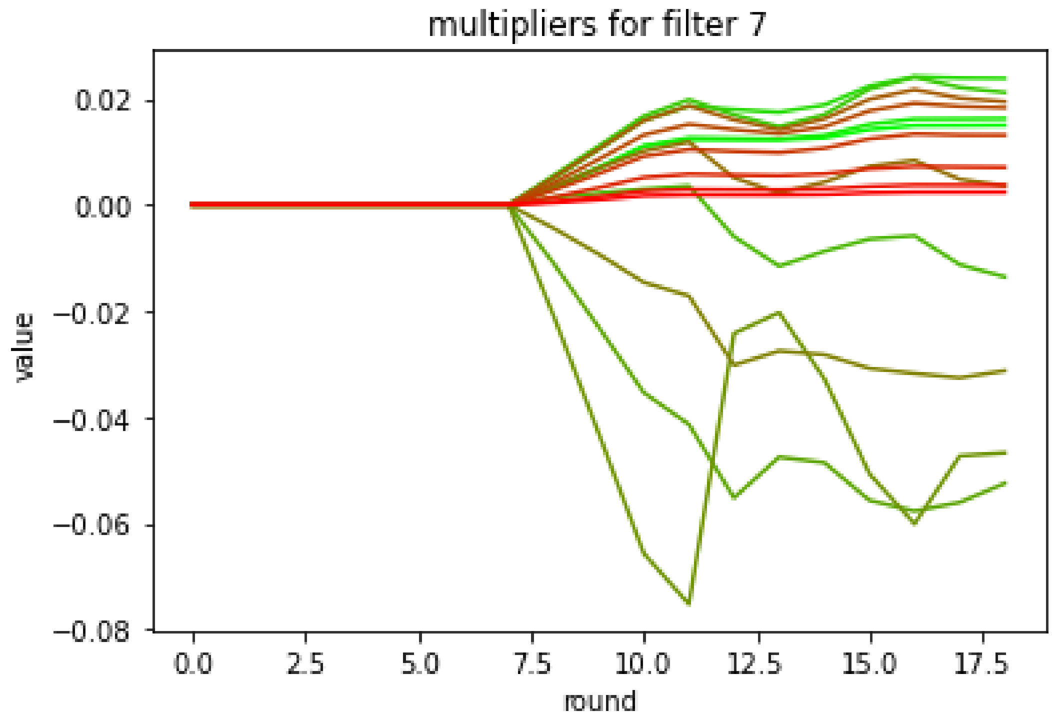

multipliers for filter 2
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

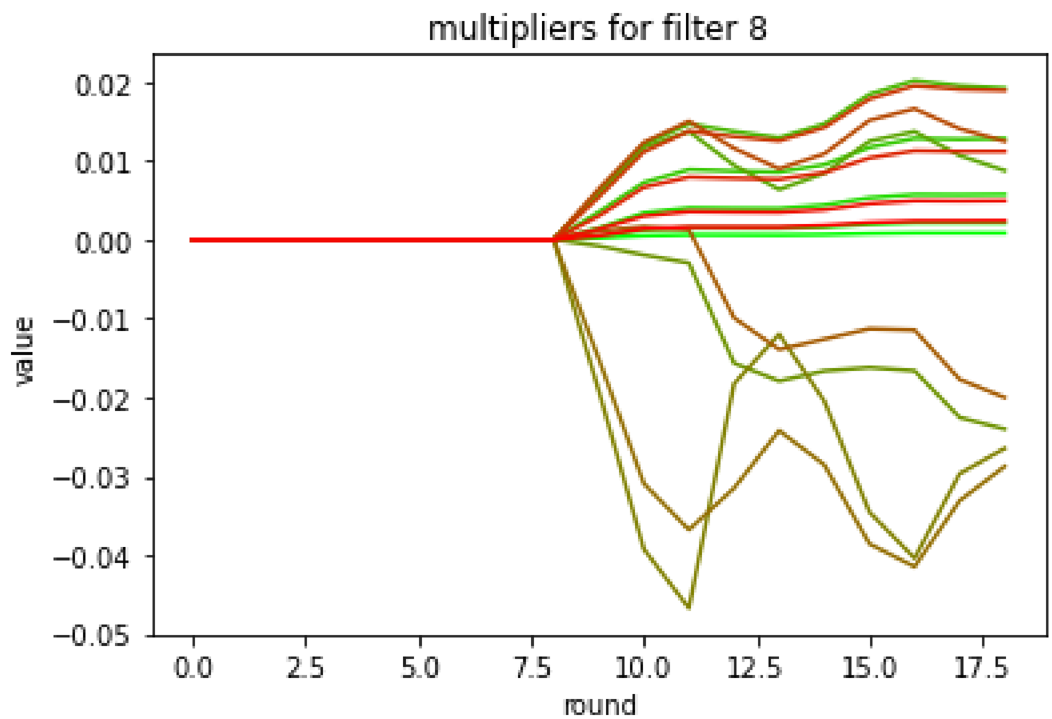

multipliers for filter 3
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

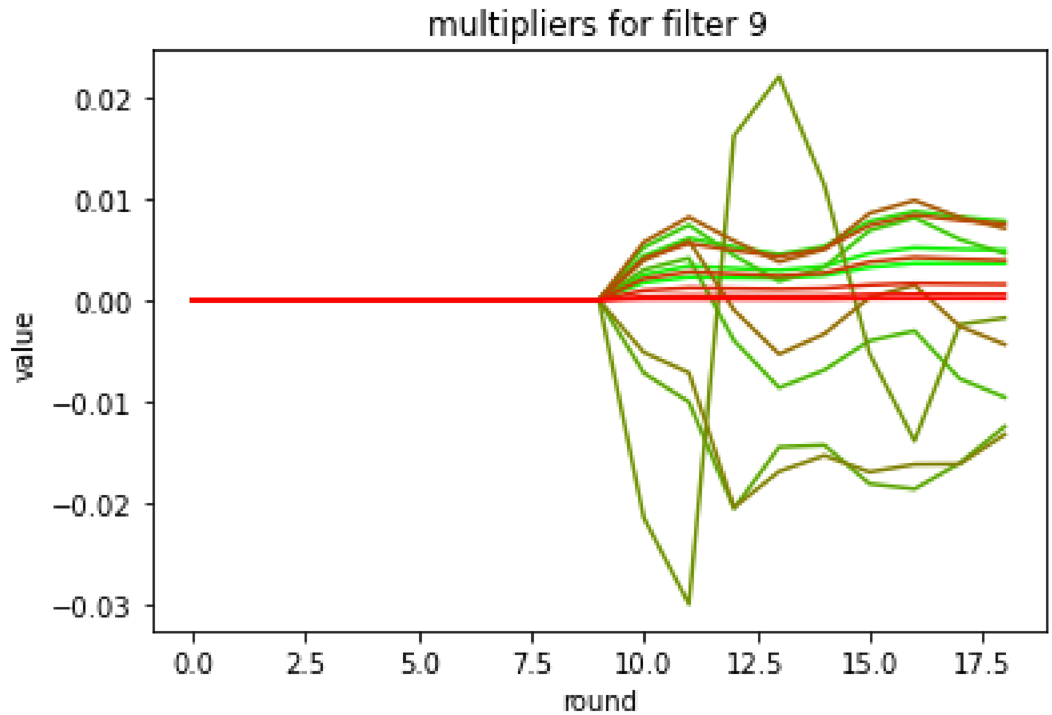

multipliers for filter 4
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.125
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

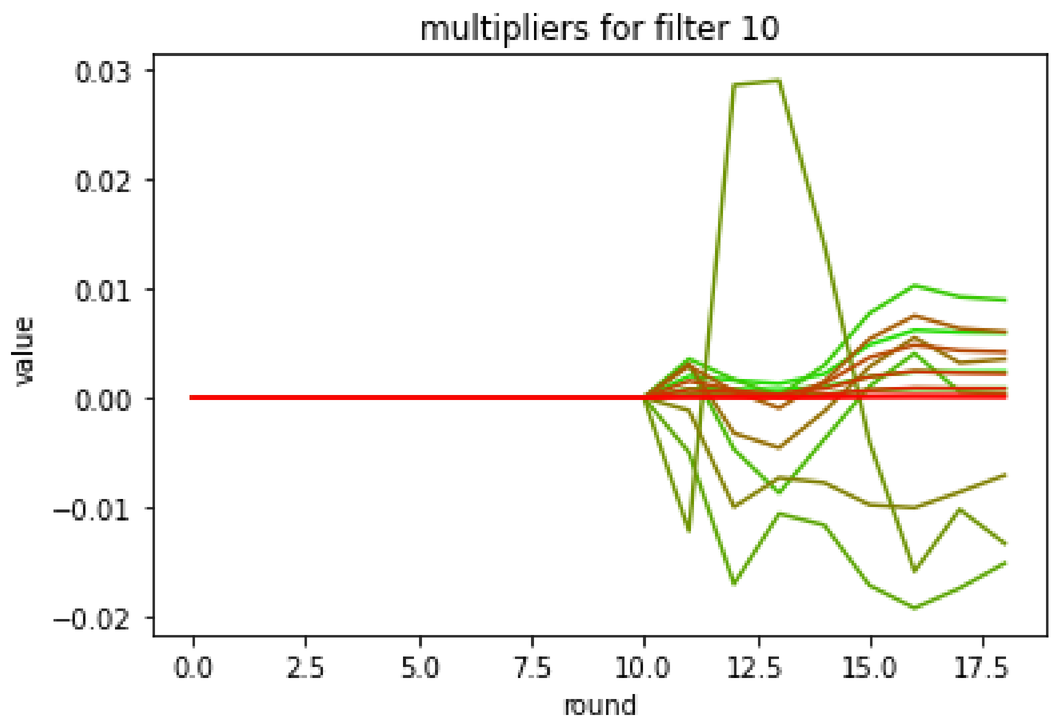

multipliers for filter 5
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

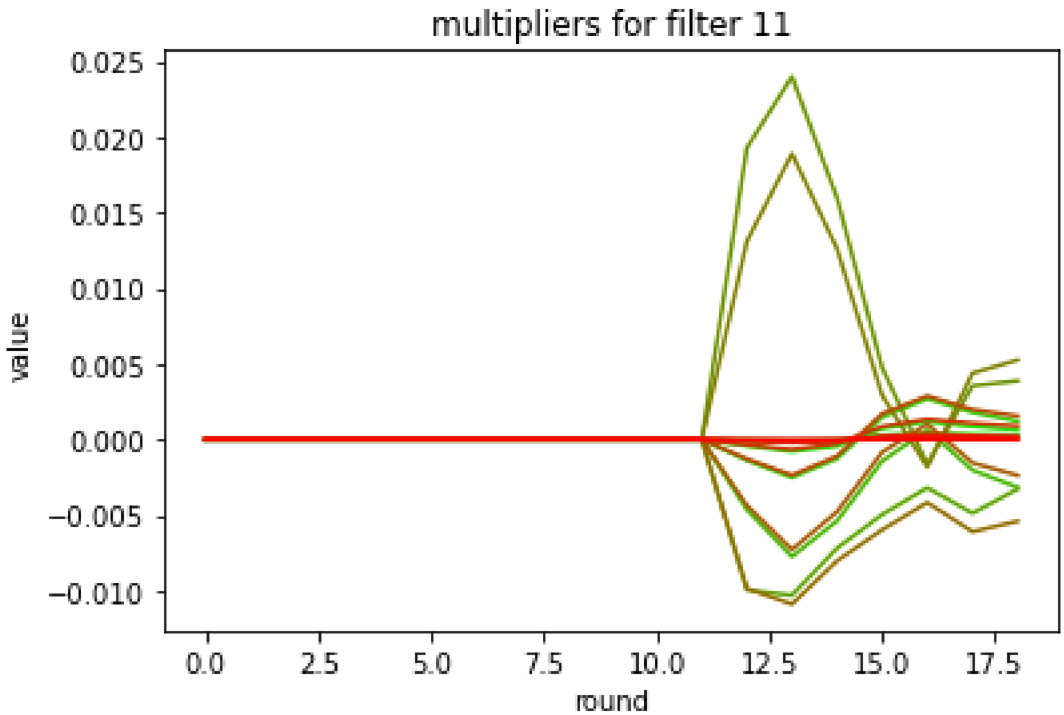

multipliers for filter 6
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

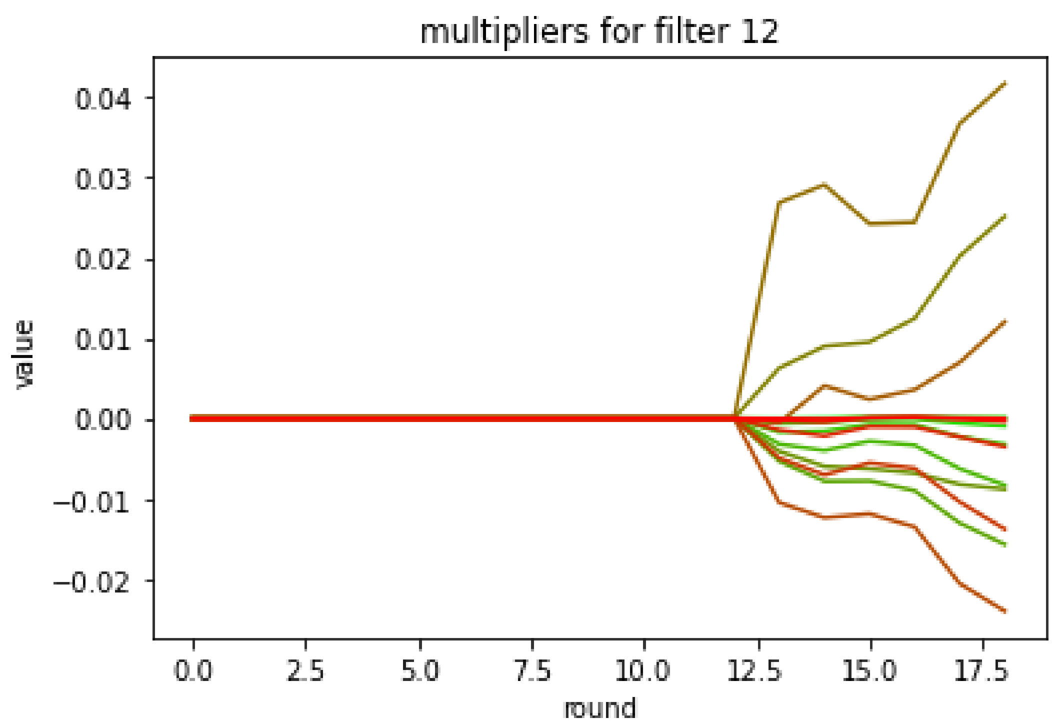

multipliers for filter 7
value
round
0.02
0.00
−0.02
−0.04
−0.06
−0.08
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

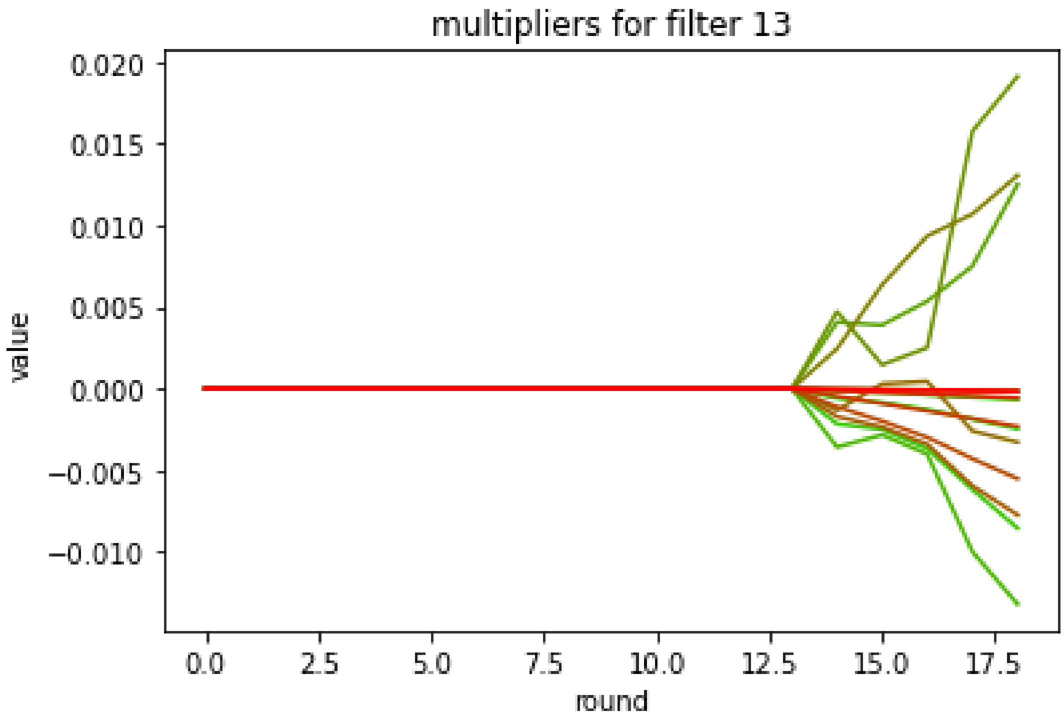

multipliers for filter 8
value
round
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


multipliers for filter 9
0.02
0.01
0.00
−0.01
−0.02
−0.03
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

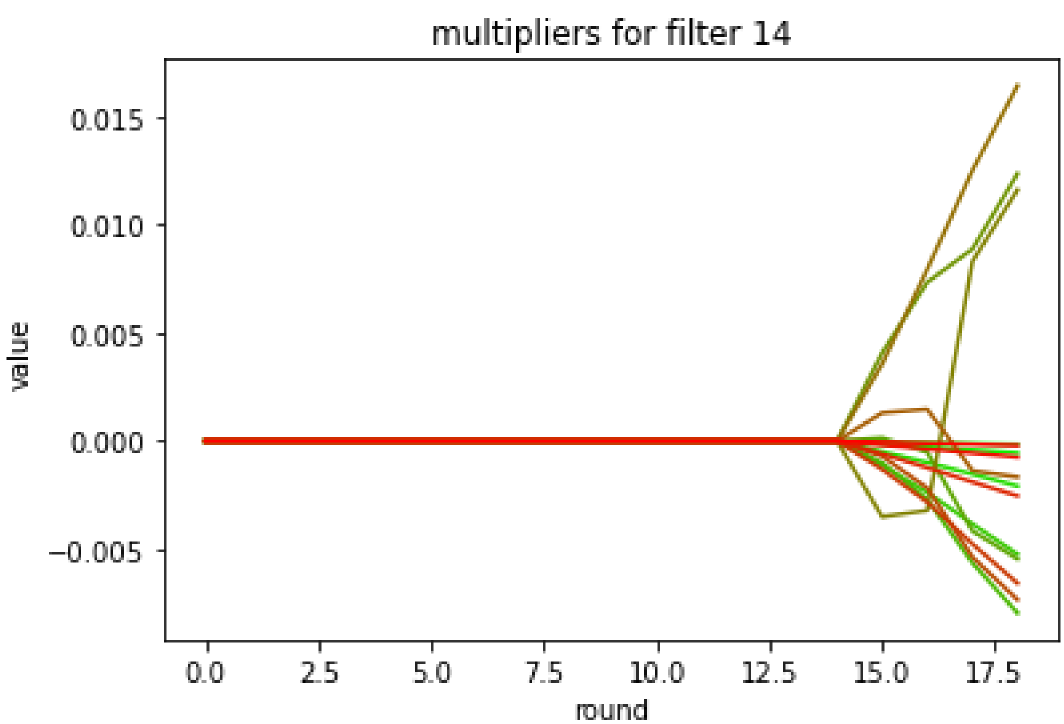

multipliers for filter 10
0.03
0.02
0.01
0.00
−0.01
−0.02
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 11
value
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

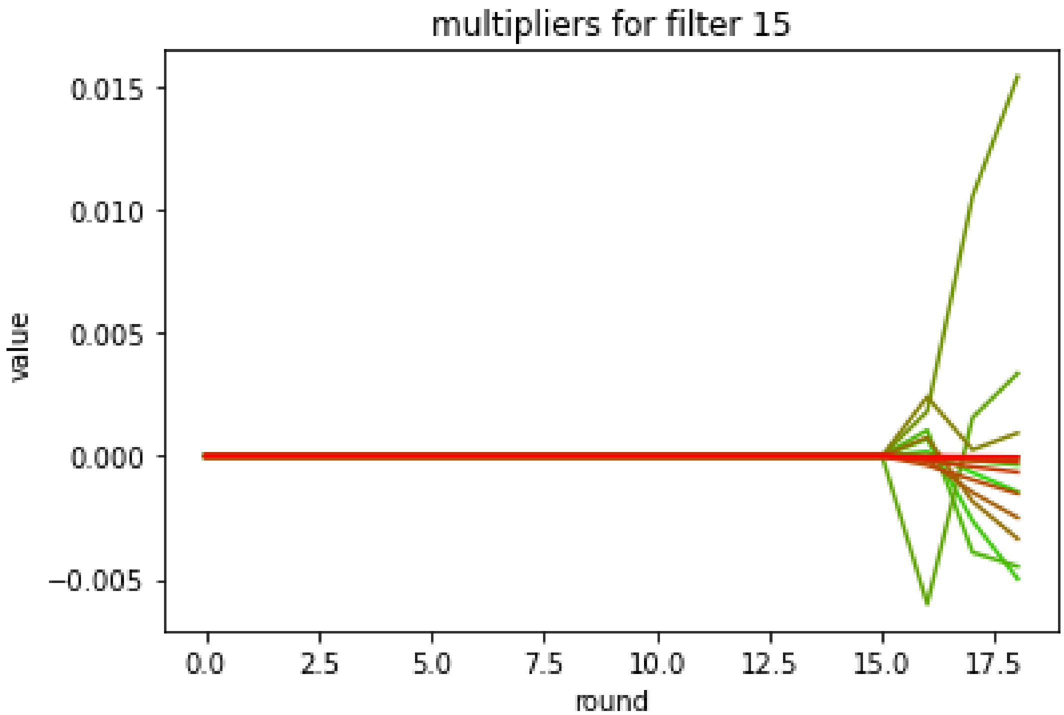

multipliers for filter 12
value
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

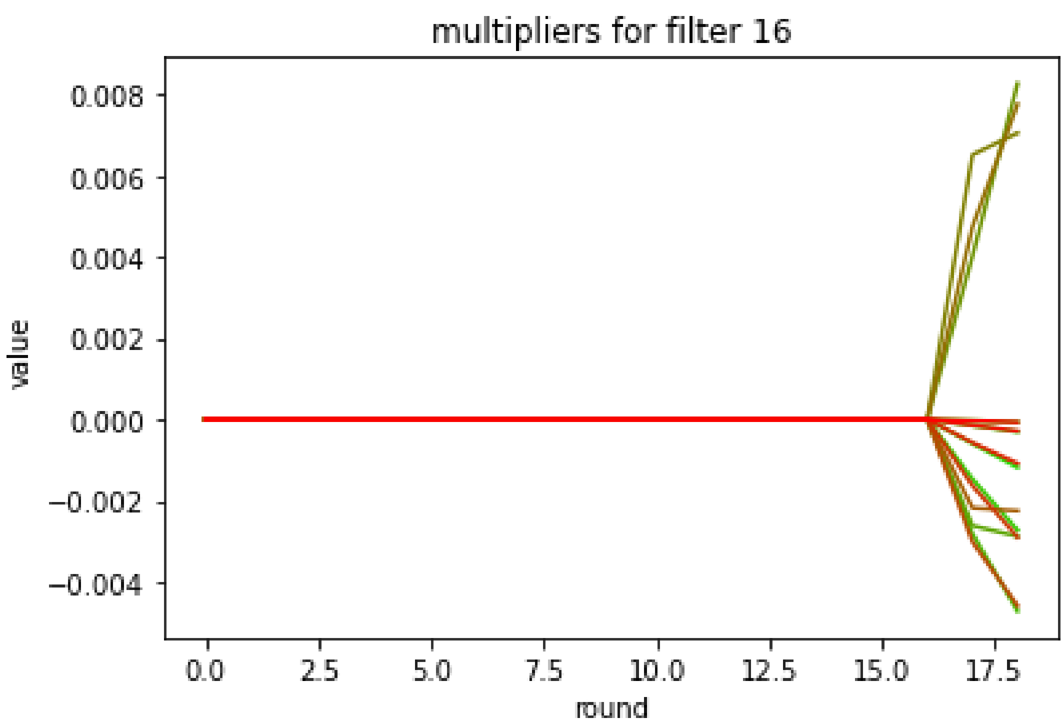


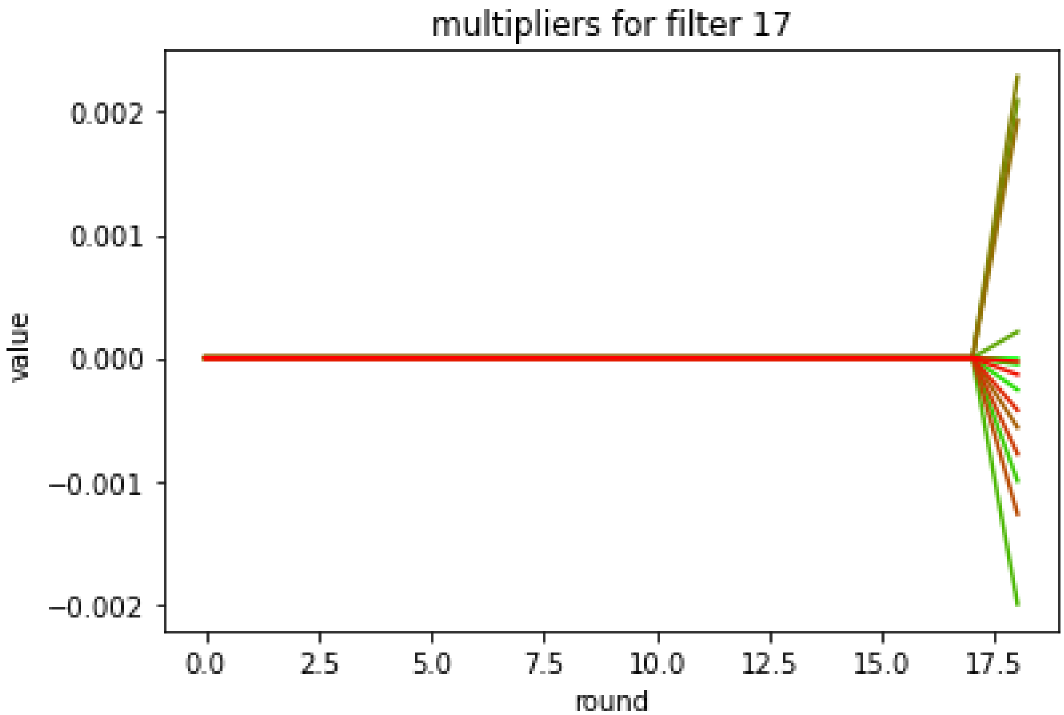
multipliers for filter 13
value
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

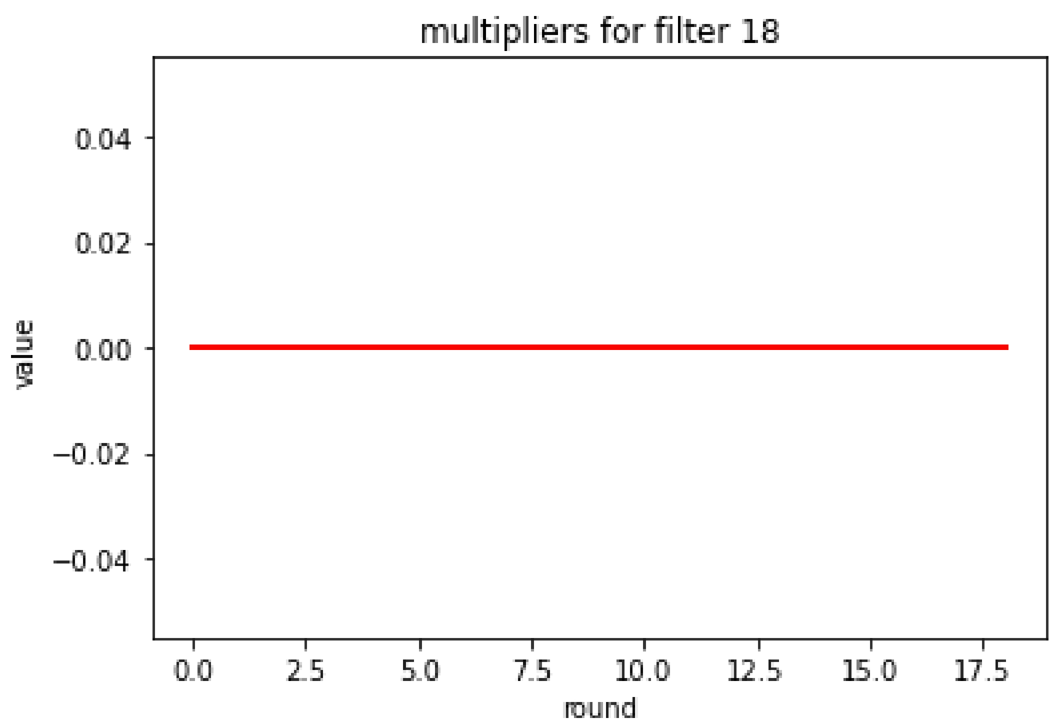


multipliers for filter 14
value
0.015
0.010
0.005
0.000
−0.005
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 15
0.015
0.010
0.005
0.000
−0.005
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

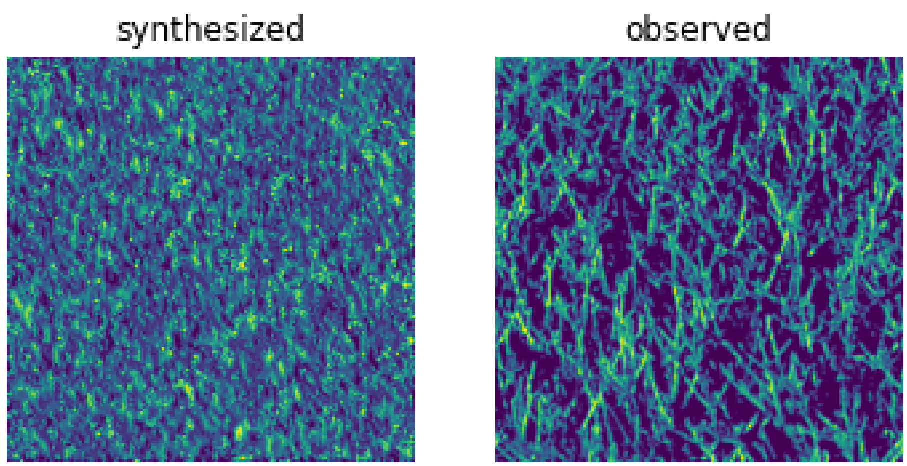

multipliers for filter 16
0.008
0.006
0.004
0.002
0.000
−0.002
−0.004
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 17
0.002
0.001
0.000
−0.001
−0.002
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round


multipliers for filter 18
0.04
0.02
0.00
−0.02
−0.04
value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
round

```
[ ]: print('Final result of FRAME on grass texture')
plt.subplot(1, 2, 1)
plt.title('synthesized')
plt.imshow(imgs_syn_grass_2[-1]/8)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('observed')
plt.imshow(example_grass/8)
plt.axis('off')
plt.show()
```

```
Final result of FRAME on grass texture
```



Following are some tools for sanity check of C extension:

```
[ ]: # test C extension, for debug
lib = ctypes.cdll.LoadLibrary('./lib_gibbs.so')
c_array = lambda a: (a.__array_interface__['data'][0] + np.arange(a.shape[0]) *␣
 ↪a.strides[0]).astype(np.uintp)
c_int32 = lambda x: x.astype(np.int32)
ndpointerpointer = lambda: ndpointer(dtype=np.uintp, ndim=1, flags='C')

# int 1D
test_int_1D = lib.test_int_1D
test_int_1D.restype = None
test_int_1D.argtypes = [ndpointer(ctypes.c_int), ctypes.c_int]

a = np.random.randn(8).astype(np.int32)
print(f'1D int in python: {a}')
test_int_1D(c_int32(a), 8)
```

```python
# double 1D
test_double_1D = lib.test_double_1D
test_double_1D.restype = None
test_double_1D.argtypes = [ndpointer(ctypes.c_double), ctypes.c_int]

b = np.random.randn(8).astype(np.float64)
print(f'1D double in python: {b}')
test_double_1D(b, 8)

# double 2D
test_double_2D = lib.test_double_2D
test_double_2D.restype = None
test_double_2D.argtypes = [ndpointerpointer(), ctypes.c_int, ctypes.c_int]

c = np.random.randn(24).reshape(2, 3, 4).astype(np.float64)
print(f'2D double in python: {c}')
test_double_2D(c_array(c.reshape(2, -1)), 2, 12)
```

```
1D int in python: [ 0  0  0  0  0  1 -1  0]
1D int in C: 0 0 0 0 0 1 -1 0
1D double in python: [-0.52572738 -0.23921949  0.74843493 -0.05246365
-1.61229869  0.39978037
 -0.41743485  0.4492817 ]
1D double in C: -0.525727 -0.239219 0.748435 -0.052464 -1.612299 0.399780
-0.417435 0.449282
2D double in python: [[[-0.70440335 -0.32157942  0.97433591 -0.4811643 ]
  [-1.64713841 -1.0085324  -1.15728378 -0.08546059]
  [-0.23865785  0.96578934 -0.32065502 -2.04889556]]

 [[ 0.82561461  0.41622182 -0.04382423  1.43162133]
  [ 1.59011811  1.04482892  2.31881768 -1.27087731]
  [ 0.77596773 -0.95883331 -0.17589816  0.64842775]]]
2D double in C: -0.704403 -0.321579 0.974336 -0.481164 -1.647138 -1.008532
-1.157284 -0.085461 -0.238658 0.965789 -0.320655 -2.048896
0.825615 0.416222 -0.043824 1.431621 1.590118 1.044829 2.318818 -1.270877
0.775968 -0.958833 -0.175898 0.648428
```