```python
import numpy as np


def split_r_hat(samples):
    """
    We take in samples, without the warm-up iterations, expecting an array of m chains by n iterations by
    k estimands (parameters, variables, etc.). We then split each chain in the middle to arrive at 2m chains
    of n/2 iterations each, on which we begin computing quantities. This all follows the logic detailed in
    chapter 11 of Bayesian Data Analysis (Gelman et al., 2013).

    :param samples: A numpy array of samples, as detailed above, m chains by n iterations by k parameters
        (estimands)
    :return: The split $\hat{R}$ statistic, an estimation of how much further scale reduction there might
        be if the sampling was let to run with $n \to \infty$ - optimally $\hat{R} \approx 1$
    """
    marginal_posterior_variance, within_chain_variance = _marginal_posterior_variance(samples)
    return np.sqrt(marginal_posterior_variance / within_chain_variance)


def _marginal_posterior_variance(samples):
    """
    Compute the marginal posterior variance as detailed leading up to equation (11.3) in Gelman (2016)
    :param samples: A numpy array of samples, as detailed above, m chains by n iterations by k parameters
        (estimands)
    :return: The marginal posterior variance $\hat{var} (\phi \mid y)$, as estimated by the within-chain
        and between chain variances
    """
    m, n, k = samples.shape
    split_chain_samples = np.vstack((samples[:, :n // 2, :], samples[:, n // 2:, :]))
    m_eff, n_eff, _ = split_chain_samples.shape

    per_chain_mean = np.mean(split_chain_samples, axis=1)
    overall_mean = np.mean(per_chain_mean, axis=0)
    between_chain_variance = n_eff / (m_eff - 1) * np.sum(np.square(per_chain_mean - overall_mean), axis=0)
    per_chain_variance = 1 / (n_eff - 1) * np.sum(np.square(split_chain_samples - np.expand_dims(per_chain_mean, 1)),
                                                  axis=1)
    within_chain_variance = np.mean(per_chain_variance, axis=0)
    marginal_posterior_variance = ((n_eff - 1) * within_chain_variance + between_chain_variance) / n_eff
    return marginal_posterior_variance, within_chain_variance


def effective_sample_size(samples):
    """
    Estimate the effective sample sizes using the variograms (square difference at lag $t$) to estimate
    the autocorrelation at each lag $t$.

    Interestingly, the code in PyStan looks substantially different than the algorithm descriptions
    in the BDA textbook: https://github.com/stan-dev/pystan/blob/develop/pystan/_chains.pyx starting
    from the fact that their implementation is based on the autocivariance, rather than the autocorrelation

    I tried to implement the code from the book, but the description is wholly incomplete. It is also
    unclear how should one handle chains with different numbers of samples due to rejections.
    :param samples: A numpy array of samples, as detailed above, m chains by n iterations by k parameters
        (estimands)
    :return: An estimate of the effective sample size, ideally approaching the true number of samples taken.
    """
    marginal_posterior_variance, _ = _marginal_posterior_variance(samples)
    m, n, k = samples.shape
    split_chain_samples = np.vstack((samples[:, :n // 2, :], samples[:, n // 2:, :]))
    m_eff, n_eff, _ = split_chain_samples.shape
    # TODO: try centering the samples in each chain, inspired by the PyStan code
    # note: this does nothing, except perhaps introduce numeric stability
    split_chain_samples = split_chain_samples - np.expand_dims(np.mean(split_chain_samples, axis=1), 1)

    # the stopping point for different variables might be different, which we need to account for
    stopping_points = {}
    t = 1
    # TODO: unclear what to do if the correlation at lag one is negative - supposedly we would stop before it?
    lag_correlations = [_estimate_lag_correlation(t, split_chain_samples, marginal_posterior_variance)]
    while len(stopping_points) < k and t < (n_eff - 1):
        rho_t_plus_1 = _estimate_lag_correlation(t + 1, split_chain_samples, marginal_posterior_variance)
        rho_t_plus_2 = _estimate_lag_correlation(t + 2, split_chain_samples, marginal_posterior_variance)
        lag_correlations.extend((rho_t_plus_1, rho_t_plus_2))

        for i in range(k):
            lag_correlation_sum = rho_t_plus_1[i] + rho_t_plus_2[i]
            if i not in stopping_points and (lag_correlation_sum < 0):
                stopping_points[i] = t

        t += 2

    # verify we have a stopping point for every variable
    # TODO: this might be where I deviate from Gelman - it's unclear if they set it to be
    # TODO: the same for all variables, or independent for each one
    for i in range(k):
        if i not in stopping_points:
            stopping_points[i] = len(lag_correlations)

```

```
    lag_correlations_array = np.asarray(lag_correlations).T
    return np.asarray([m_eff * n_eff / (1 + 2 * np.sum(lag_correlations_array[i, :stopping_points[i]]))
                       for i in range(k)])


def _estimate_lag_correlation(t, split_chain_samples, marginal_posterior_variance):
    """
    Astimate the autocorrelation at lag t using the variogram
    :param t: The lag to estaimte in
    :param split_chain_samples: The samples for each chain, split in half, so double the actual
        sampled chains but each with half the number of samples
    :param marginal_posterior_variance: The marginal posterior variance estimated using the function
        implemented above
    :return: An estimate for the autocorrelation of the samples of each parameter (the third dimension
        of the split_chain_samples parameter) using the variogram.
    """
    variogram = np.mean(np.square(split_chain_samples[:, t:, :] - split_chain_samples[:, :-1 * t, :]), axis=(0, 1))
    return 1 - (variogram / (2 * marginal_posterior_variance))
```