Project Documentation

File: `setup.py`

Functions

`load_text(path)` **(Line 5)**

No docstring

File: `benchmark_image_sizes.py`

Functions

`get_memory_usage()` **(Line 14)**

No docstring

`log_memory_usage(memory_usage, log_interval)` **(Line 20)**

No docstring

`build_solver(solver_name, A, Acsr, Acsc, Acoo, AL, b, atol, rtol)` **(Line 28)**

No docstring

`run_solver_single_image(solver_name, scale, index)` **(Line 102)**

No docstring

`run_solver(solver_name)` **(Line 194)**

No docstring

`main()` **(Line 214)**

No docstring

File: `calculate_laplacian_error.py`

Functions

`compute_alpha(image, trimap, laplacian_name, is_fg, is_bg, is_known)` **(Line 18)**

No docstring

`main()` **(Line 59)**

No docstring

File: `config.py`

Functions

`get_library_path(name)` **(Line 33)**

No docstring

File: `plot_laplacian_average_error.py`

File: `plot_laplacian_error_per_image.py`

File: `plot_results.py`

Functions

`make_fancy_bar_plot(filename, title, xlabel, xticks, bar_widths, bar_labels, xerr)` **(Line 19)**

No docstring

`plot_memory_usage()` **(Line 51)**

No docstring

`plot_time()` **(Line 95)**

No docstring

`plot_time_image_size()` **(Line 138)**

No docstring

File: `solve_amgcl.py`

Functions

`solve_amgcl_csr(csr_values, csr_indices, csr_indptr, b, x, atol, rtol, maxiter)` **(Line 26)**

No docstring

`main()` **(Line 79)**

No docstring

File: `solve_eigen.py`

Functions

`solve_eigen_icholt_coo(coo_data, row, col, b, rtol, initial_shift)` **(Line 37)**

No docstring

`solve_eigen_cholesky_coo(coo_data, row, col, b)` **(Line 79)**

No docstring

`main()` **(Line 112)**

No docstring

File: `solve_mumps.py`

Functions

`solve_mumps_coo(coo_values, i_inds, j_inds, b, x, is_symmetric, print_info)` **(Line 32)**

No docstring

`main()` **(Line 73)**

No docstring

File: `solve_petsc.py`

Functions

`solve_petsc_coo(coo_values, i_inds, j_inds, b, x, atol, rtol, gamg_threshold, maxiter)` **(Line 35)**

No docstring

`main()` **(Line 86)**

No docstring

File: `__about__.py`

File: `__init__.py`

File: `estimate_alpha_cf.py`

Functions

`estimate_alpha_cf(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` **(Line 8)**

Estimate alpha from an input image and an input trimap using Closed-Form Alpha Matting as proposed by :cite: `levin2007closed`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: ichol) laplacian*kwargs: dictionary Arguments passed to the :code: `cf_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver is_known: numpy.ndarray Binary mask of pixels for which to compute the laplacian matrix. Providing this parameter might improve performance if few pixels are unknown.

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*cf(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_knn.py`

Functions

`estimate_alpha_knn(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` **(Line 9)**

Estimate alpha from an input image and an input trimap using KNN Matting similar to :cite: `chen2013knn`. See `knn_laplacian` for more details.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: jacobi) laplacian*kwargs: dictionary Arguments passed to the :code: `knn_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*knn(... image, ... trimap, ... laplacian*kwargs={"n*neighbors": [15, 10]}, ... cg*kwargs={"maxiter":2000})

File: `estimate_alpha_lbdm.py`

Functions

`estimate_alpha_lbdm(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` (Line 9)

Estimate alpha from an input image and an input trimap using Learning Based Digital Matting as proposed by :cite: `zheng2009learning`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: ichol) laplacian*kwargs: dictionary Arguments passed to the :code: `lbdm_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*lbdm(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_lkm.py`

Functions

`estimate_alpha_lkm(image, trimap, laplacian_kwargs, cg_kwargs)` (Line 8)

Estimate alpha from an input image and an input trimap as described in Fast Matting Using Large Kernel Matting Laplacian Matrices by :cite: `he2010fast`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image laplacian*kwargs: dictionary Arguments passed to the :code: `lkm_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*lkm(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6, "radius": 15}, ... cg_kwargs={"maxiter":2000})

`A_matvec(x)` **(Line 54)**

No docstring

`jacobi(x)` **(Line 57)**

No docstring

File: `estimate_alpha_rw.py`

Functions

`estimate_alpha_rw(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` **(Line 9)**

Estimate alpha from an input image and an input trimap using Learning Based Digital Matting as proposed by :cite:`grady2005random`.

Parameters

image: numpy.ndarray Image with shape :math:`h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math:`h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: jacobi) laplacian*kwargs: dictionary Arguments passed to the :code:`rw_laplacian` function cg*kwargs: dictionary Arguments passed to the :code:`cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*rw(.... image, ... trimap, ... laplacian*kwargs={"sigma": 0.03}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_sm.py`

Functions

`estimate_alpha_sm(image, trimap, return_foreground_background, trimap_expansion_radius, trimap_expansion_threshold, sample_gathering_angles, sample_gathering_weights, sample_gathering_Np_radius, sample_refinement_radius, local_smoothing_radius1, local_smoothing_radius2, local_smoothing_radius3, local_smoothing_sigma_sq1, local_smoothing_sigma_sq2, local_smoothing_sigma_sq3)` **(Line 4)**

Estimate alpha from an input image and an input trimap using Shared Matting as proposed by :cite: `Gastal0liveira2010SharedMatting`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image return*foreground*background: numpy.ndarray Whether to return foreground and background estimate. They will be computed either way trimap*expansion*radius: int How much to expand trimap. trimap*expansion*threshold: float Which pixel colors are similar enough to expand trimap into sample*gathering*angles: int In how many directions to search for new samples. sample*gathering*weights: Tuple[float, float, float, float] Weights for various cost functions sample*gathering*Np*radius: int Radius of Np function sample*refinement*radius: int Search region for better neighboring samples local*smoothing*radius1: int Radius for foreground/background smoothing local*smoothing*radius2: int Radius for confidence computation local*smoothing*radius3: int Radius for low frequency alpha computation local*smoothing*sigma*sq1: float Squared sigma value for foreground/background smoothing Defaults to :code: `(2 * local_smoothing_radius1 + 1)**2 / (9 * pi)` if not given local*smoothing*sigma*sq2: float Squared sigma value for confidence computation local*smoothing*sigma*sq3: float Squared sigma value for low frequency alpha computation Defaults to :code: `(2 * local_smoothing_radius3 + 1)**2 / (9 * pi)` if not given

Returns

alpha: numpy.ndarray Estimated alpha matte foreground: numpy.ndarray Estimated foreground background: numpy.ndarray Estimated background

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha, foreground, background = estimate*alpha*sm(... image, ... trimap, ... return*foreground*background=True, ... sample*gathering_angles=4)

`estimate_alpha(I, F, B)` **(Line 168)**

No docstring

`inner(a, b)` **(Line 186)**

No docstring

`Mp2(I, F, B)` **(Line 193)**

No docstring

`Np(image, x, y, F, B, r)` **(Line 203)**

No docstring

`Ep(image, px, py, sx, sy)` **(Line 214)**

No docstring

`dist(a, b)` **(Line 260)**

No docstring

`length(a)` **(Line 267)**

No docstring

`expand_trimap(expanded_trimap, trimap, image, k_i, k_c)` **(Line 271)**

No docstring

`sample_gathering(gathering_F, gathering_B, gathering_alpha, image, trimap, num_angles, eN, eA, ef, eb, Np_radius)` **(Line 302)**

No docstring

`sample_refinement(refined_F, refined_B, refined_alpha, gathering_F, gathering_B, image, trimap, radius)` **(Line 433)**

No docstring

`local_smoothing(final_F, final_B, final_alpha, refined_F, refined_B, refined_alpha, image, trimap, radius1, radius2, radius3, sigma_sq1, sigma_sq2, sigma_sq3)` **(Line 488)**

No docstring

File: `__init__.py`

File: `__init__.py`

File: `cutout.py`

Functions

`cutout(image_path, trimap_path, cutout_path)` **(Line 6)**

Generate a cutout image from an input image and an input trimap. This method is using closed-form alpha matting as proposed by :cite: `levin2007closed` and multi-level foreground extraction :cite: `germer2020multilevel` .

Parameters

image*path: str Path of input image trimap*path: str Path of input trimap cutout_path: str Path of output cutout image

Example

cutout("../data/lemur.png", "../data/lemur*trimap.png", "lemur*cutout.png")

File: `__init__.py`

File: `estimate_foreground_cf.py`

Functions

`estimate_foreground_cf(image, alpha, regularization, rtol, neighbors, return_background, foreground_guess, background_guess, ichol_kwargs, cg_kwargs)` **(Line 8)**

Estimates the foreground of an image given alpha matte and image.

This method is based on the publication :cite: `levin2007closed`.

Parameters

image: numpy.ndarray Input image with shape :math: `h \times w \times d` alpha: numpy.ndarray Input alpha matte with shape :math: `h \times w` regularization: float Regularization strength :math: `\epsilon`, defaults to :math: `10^{-5}` neighbors: list of tuples of ints List of relative positions that define the neighborhood of a pixel return*background: bool Whether to return the estimated background in addition to the foreground foreground*guess: numpy.ndarray An initial guess for the foreground image in order to accelerate convergence. Using input image by default. background*guess: numpy.ndarray An initial guess for the background image. Using input image by default. ichol*kwargs: dictionary Keyword arguments for the incomplete Cholesky preconditioner cg_kwargs: dictionary Keyword arguments for the conjugate gradient descent solver

Returns

F: numpy.ndarray Extracted foreground B: numpy.ndarray Extracted background (not returned by default)

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") alpha = load*image("data/lemur/lemur*alpha.png", "GRAY") F = estimate*foreground*cf(image, alpha, return*background=False) F, B = estimate*foreground*cf(image, alpha, return_background=True)

See Also

stack_images: This function can be used to place the foreground on a new background.

File: `estimate_foreground_ml.py`

Functions

`_resize_nearest_multichannel(dst, src)` (Line 6)

Internal method.

Resize image src to dst using nearest neighbors filtering. Images must have multiple color channels, i.e. :code: `len(shape) == 3`.

Parameters

dst: numpy.ndarray of type np.float32 output image src: numpy.ndarray of type np.float32 input image

`_resize_nearest(dst, src)` (Line 33)

Internal method.

Resize image src to dst using nearest neighbors filtering. Images must be grayscale, i.e. :code: `len(shape) == 3`.

Parameters

dst: numpy.ndarray of type np.float32 output image src: numpy.ndarray of type np.float32 input image

`_estimate_fb_ml(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, gradient_weight)` (Line 62)

No docstring

`estimate_foreground_ml(image, alpha, regularization, n_small_iterations, n_big_iterations, small_size, return_background, gradient_weight)` (Line 186)

Estimates the foreground of an image given its alpha matte.

See :cite: `germer2020multilevel` for reference.

Parameters

image: numpy.ndarray Input image with shape :math: `h \times w \times d` alpha: numpy.ndarray Input alpha matte shape :math: `h \times w` regularization: float Regularization strength :math: `\epsilon`, defaults to :math: `10^{-5}`. Higher regularization results in smoother colors. n*small*iterations: int Number of iterations performed on small scale, defaults to :math: `10` n*big*iterations: int Number of iterations performed on large scale, defaults to :math: `2` small*size: int Threshold that determines at which size `n_small_iterations` should be used return*background: bool Whether to return the estimated background in addition to the foreground gradient_weight: float Larger values enforce smoother foregrounds, defaults to :math: `1`

Returns

F: numpy.ndarray Extracted foreground B: numpy.ndarray Extracted background

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") alpha = load*image("data/lemur/lemur*alpha.png", "GRAY") F = estimate*foreground*ml(image, alpha, return*background=False) F, B = estimate*foreground*ml(image, alpha, return_background=True)

See Also

stack_images: This function can be used to place the foreground on a new background.

File: `estimate_foreground_ml_cupy.py`

Functions

`estimate_foreground_ml_cupy(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, block_size, return_background, to_numpy)` **(Line 110)**

See the :code: `estimate_foreground` method for documentation.

`resize_nearest(dst, src, w_src, h_src, w_dst, h_dst, depth)` **(Line 152)**

No docstring

File: `estimate_foreground_ml_pyopencl.py`

Functions

`estimate_foreground_ml_pyopencl(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, return_background)` **(Line 104)**

See the :code: `estimate_foreground` method for documentation.

`upload(array)` **(Line 115)**

No docstring

`alloc()` **(Line 123)**

No docstring

`download(device_buf, shape)` **(Line 127)**

No docstring

`resize_nearest(dst, src, w_src, h_src, w_dst, h_dst, depth)` **(Line 154)**

No docstring

File: `__init__.py`

File: `cf_laplacian.py`

Functions

`_cf_laplacian(image, epsilon, r, values, indices, indptr, is_known)` **(Line 6)**

No docstring

`cf_laplacian(image, epsilon, radius, is_known)` **(Line 132)**

This function implements the alpha estimator for closed-form alpha matting as proposed by :cite:`levin2007closed`.

Parameters

image: numpy.ndarray Image with shape :math:`h\times w \times 3` epsilon: float Regularization strength, defaults to :math:`10^{-7}`. Strong regularization improves convergence but results in smoother alpha mattes. radius: int Radius of local window size, defaults to :math:`1`, i.e. only adjacent pixels are considered. The size of the local window is given as :math:`(2 r + 1)^2`, where :math:`r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. is_known: numpy.ndarray Binary mask of pixels for which to compute the laplacian matrix. Laplacian entries for known pixels will have undefined values.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `knn_laplacian.py`

Functions

`knn_laplacian(image, n_neighbors, distance_weights, kernel)` **(Line 7)**

This function calculates the KNN matting Laplacian matrix similar to :cite:`chen2013knn`. We use a kernel of 1 instead of a soft kernel by default since the former is faster to compute and both produce almost identical results in all our experiments, which is to be expected as the soft kernel is very close to 1 in most cases.

Parameters

image: numpy.ndarray Image with shape :math:`h\times w \times 3` n*neighbors: list of ints Number of neighbors to consider. If :code: `len(n_neighbors)>1` multiple nearest neighbor calculations are done and merged, defaults to `[20, 10]`, i.e. first 20 neighbors are considered and in the second run :math: `10` neighbors. The pixel distances are then weighted by the :code: `distance_weights`. distance*weights: list of floats Weight of distance in feature vector, defaults to `[2.0, 0.1]`. kernel: str Must be either "binary" or "soft". Default is "binary".

Returns

L: scipy.sparse.spmatrix Matting Laplacian matrix

File: `laplacian.py`

Functions

`make_linear_system(L, trimap, lambda_value, return_c)` (Line 5)

This function constructs a linear system from a matting Laplacian by constraining the foreground and background pixels with a diagonal matrix `C` to values in the right-hand-side vector `b`. The constraints are weighted by a factor :math:`\lambda`. The linear system is given as

.. math::

A = L + \lambda C,

where :math:`C=\mathop{Diag}(c)` having :math:`c_i = 1` if pixel i is known and :math:`c_i = 0` otherwise. The right-hand-side :math:`b` is a vector with entries :math:`b_i = 1` is pixel is is a foreground pixel and :math:`b_i = 0` otherwise.

Parameters

L: scipy.sparse.spmatrix Laplacian matrix, e.g. calculated with :code:`lbdm_laplacian` function trimap: numpy.ndarray Trimap with shape :math:`h\times w` lambda*value: float Constraint penalty, defaults to 100 return*c: bool Whether to return the constraint matrix `C`, defaults to False

Returns

A: scipy.sparse.spmatrix Matrix describing the system of linear equations b: numpy.ndarray Vector describing the right-hand side of the system C: numpy.ndarray Vector describing the diagonal entries of the matrix `C`, only returned if `return_c` is set to True

File: `lbdm_laplacian.py`

Functions

`calculate_kernel_matrix(X, v)` (Line 6)

No docstring

`_lbdm_laplacian(image, epsilon, r)` (Line 16)

No docstring

`lbdm_laplacian(image, epsilon, radius)` (Line 64)

Calculate a Laplacian matrix based on :cite:`zheng2009learning`.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` epsilon: float Regularization strength, defaults to :math: `10^{-7}`. Strong regularization improves convergence but results in smoother alpha mattes. radius: int Radius of local window size, defaults to :math: `1`, i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2`, where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image.

Returns

L: scipy.sparse.csr_matrix Matting Laplacian

File: `lkm_laplacian.py`

Functions

`lkm_laplacian(image, epsilon, radius, return_diagonal)` **(Line 6)**

Calculates the Laplacian for large kernel matting :cite: `he2010fast`

Parameters

image: numpy.ndarray Image of shape :math: `h\times w \times 3` epsilons: float Regularization strength, defaults to :math: `10^{-7}` radius: int Radius of local window size, defaults to :math: `10`, i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2`, where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. return_diagonal: bool Whether to also return the diagonal of the laplacian, defaults to True

Returns

L*matvec: function Function that applies the Laplacian matrix to a vector diag*L: numpy.ndarray Diagonal entries of the matting Laplacian, only returns if `return_diagonal` is True

`L_matvec(p)` **(Line 51)**

No docstring

File: `rw_laplacian.py`

Functions

`_rw_laplacian(image, sigma, r)` **(Line 7)**

No docstring

`rw_laplacian(image, sigma, radius, regularization)` **(Line 47)**

This function implements the alpha estimator for random walk alpha matting as described in :cite: `grady2005random`.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` sigma: float Sigma used to calculate the weights (see Equation 4 in :cite: `grady2005random`), defaults to :math: `0.033` radius: int Radius of local window size, defaults to :math: `1` , i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2` , where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. regularization: float Regularization strength, defaults to :math: `10^{-8}` . Strong regularization improves convergence but results in smoother alpha matte.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `uniform_laplacian.py`

Functions

`uniform_laplacian(image, radius)` **(Line 9)**

This function returns a Laplacian matrix with all weights equal to one.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` radius: int Radius of local window size, defaults to 1, i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2` , where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `__init__.py`

File: `ichol.py`

Classes

`CholeskyDecomposition` **(Line 148)**

Cholesky Decomposition

Calling this object applies the preconditioner to a vector by forward and back substitution.

Parameters

Ltuple: tuple of numpy.ndarrays Tuple of array describing values, row indices and row pointers for Cholesky factor in the compressed sparse column format (csc)

Methods: - `__init__(self, Ltuple)`

Line 159: No docstring

- `L(self)`
  *Line 163: Returns the Cholesky factor

Returns

L: scipy.sparse.csc_matrix Cholesky factor*

- `__call__(self, b)`
  Line 175: No docstring

File: `jacobi.py`

Functions

`jacobi(A)` **(Line 1)**

Compute the Jacobi preconditioner function for the matrix A.

Parameters

A: np.array Input matrix to compute the Jacobi preconditioner for.

Returns

precondition_matvec: function Function which applies the Jacobi preconditioner to a vector

Example

> from pymatting import * import numpy as np A = np.array([[2, 3], [3, 5]]) preconditioner = jacobi(A) preconditioner(np.array([1, 2])) array([0.5, 0.4])

`precondition_matvec(x)` **(Line 28)**

No docstring

File: `vcycle.py`

Functions

`make_P(shape)` **(Line 6)**

No docstring

`jacobi_step(A, A_diag, b, x, num_iter, omega)` **(Line 32)**

No docstring

`_vcycle_step(A, b, shape, cache, num_pre_iter, num_post_iter, omega,` `direct_solve_size)` **(Line 46)**

No docstring

`vcycle(A, shape, num_pre_iter, num_post_iter, omega, direct_solve_size, cache)` **(Line 103)**

Implements the V-Cycle preconditioner. The V-Cycle solver was recommended by :cite: `lee2014scalable` to solve the alpha matting problem.

Parameters

A: numpy.ndarray Input matrix shape: tuple of ints Describing the height and width of the image num*pre*iter: int Number of Jacobi iterations before each V-Cycle, defaults to 1 num*post*iter: int Number of Jacobi iterations after each V-Cycle, defaults to 1 omega: float Weight parameter for the Jacobi method. If method fails to converge, try different values.

Returns

precondition: function Function which applies the V-Cycle preconditioner to a vector

Example

> from pymatting import * import numpy as np from scipy.sparse import csc_matrix A = np.array([[2, 3], [3, 5]]) preconditioner = vcycle(A, (2, 2)) preconditioner(np.array([1, 2])) array([-1., 1.])

`precondition(r)` **(Line 148)**

No docstring

File: `__init__.py`

File: `callback.py`

Classes

`CounterCallback` **(Line 1)**

Callback to count number of iterations of iterative solvers.

Methods: - `__init__(self)`
Line 4: No docstring

- `__call__(self, A, x, b, norm_b, r, norm_r)`
  Line 7: No docstring

`ProgressCallback` **(Line 11)**

Callback to count number of iterations of iterative solvers. Also prints residual error.

Methods: - `__init__(self)`
Line 17: No docstring

- `__call__(self, A, x, b, norm_b, r, norm_r)`
  Line 20: No docstring

File: `cg.py`

Functions

`cg(A, b, x0, atol, rtol, maxiter, callback, M, reorthogonalize)` (Line 4)

Solves a system of linear equations :math: `Ax=b` using conjugate gradient descent :cite: `hestenes1952methods`

Parameters

A: scipy.sparse.csr*matrix Square matrix b: numpy.ndarray Vector describing the right-hand side of the system x0: numpy.ndarray Initialization, if `None` then :code: `x=np.zeros_like(b)` atol: float Absolute tolerance. The loop terminates if the :math: `||r||` is smaller than `atol`, where :math: `r` denotes the residual of the current iterate. rtol: float Relative tolerance. The loop terminates if :math: `{||r||}/{||b||}` is smaller than `rtol`, where :math: `r` denotes the residual of the current iterate. callback: function Function :code: `callback(A, x, b, norm_b, r, norm_r)` called after each iteration, defaults to `None` M: function or scipy.sparse.csr*matrix Function that applies the preconditioner to a vector. Alternatively, `M` can be a matrix describing the precondioner. reorthogonalize: boolean Whether to apply reorthogonalization of the residuals after each update, defaults to `False`

Returns

x: numpy.ndarray Solution of the system

Example

> from pymatting import * import numpy as np A = np.array([[3.0, 1.0], [1.0, 2.0]]) M = jacobi(A) b = np.array([4.0, 3.0]) cg(A, b, M=M) array([1., 1.])

`precondition(x)` (Line 54)

No docstring

`precondition(x)` (Line 61)

No docstring

File: `__init__.py`

File: `boxfilter.py`

Functions

`boxfilter_rows_valid(src, r)` **(Line 7)**

No docstring

`boxfilter_rows_same(src, r)` **(Line 32)**

No docstring

`boxfilter_rows_full(src, r)` **(Line 61)**

No docstring

`boxfilter(src, radius, mode)` **(Line 90)**

Computes the boxfilter (uniform blur, i.e. blur with kernel :code: `np.ones(radius, radius)`) of an input image.

Depending on the mode, the input image of size :math: `(h, w)` is either of shape

- :math: `(h - 2 r, w - 2 r)` in case of 'valid' mode
- :math: `(h, w)` in case of 'same' mode
- :math: `(h + 2 r, w + 2 r)` in case of 'full' mode

.. image:: figures/padding.png

Parameters

src: numpy.ndarray Input image having either shape :math: `h \times w \times d` or :math: `h \times w` radius: int Radius of boxfilter, defaults to :math: `3` mode: str One of 'valid', 'same' or 'full', defaults to 'same'

Returns

dst: numpy.ndarray Blurred image

Example

> from pymatting import * import numpy as np boxfilter(np.eye(5), radius=2, mode="valid") array([[5.]]) boxfilter(np.eye(5), radius=2, mode="same") array([[3., 3., 3., 2., 1.], [3., 4., 4., 3., 2.], [3., 4., 5., 4., 3.], [2., 3., 4., 4., 3.], [1., 2., 3., 3., 3.]]) boxfilter(np.eye(5), radius=2, mode="full") array([[1., 1., 1., 1., 1., 0., 0., 0., 0.], [1., 2., 2., 2., 2., 1., 0., 0., 0.], [1., 2., 3., 3., 3., 2., 1., 0., 0.], [1., 2., 3., 4., 4., 3., 2., 1., 0.], [1., 2., 3., 4., 5., 4., 3., 2., 1.], [0., 1., 2., 3., 4., 4., 3., 2., 1.], [0., 0., 1., 2., 3., 3., 3., 2., 1.], [0., 0., 0., 1., 2., 2., 2., 2., 1.], [0., 0., 0., 0., 1., 1., 1., 1., 1.]])

File: `distance.py`

Functions

`_propagate_1d_first_pass(d)` **(Line 6)**

No docstring

`_propagate_1d(d, v, z, f)` **(Line 18)**

No docstring

`_propagate_distance(distance)` **(Line 62)**

No docstring

`distance_transform(mask)` **(Line 76)**

For every non-zero value, compute the distance to the closest zero value. Based on :cite:`felzenszwalb2012distance`.

Parameters

mask: numpy.ndarray 2D matrix of zero and nonzero values.

Returns

distance: numpy.ndarray Distance to closest zero-valued pixel.

Example

> from pymatting import * import numpy as np mask = np.random.rand(10, 20) < 0.9 distance = distance_transform(mask)

File: `kdtree.py`

Classes

`KDTree` **(Line 236)**

KDTree implementation

Methods: - `__init__(self, data_points, min_leaf_size)`
*Line 239: Constructs a KDTree for given data points. The implementation currently only supports data type `np.float32`.

Parameters

data*points: numpy.ndarray (of type `np.float32`) Dataset with shape :math:`n \times d`, where :math:`n` is the number of data points in the data set and :math:`d` is the dimension of each data point min*leaf_size: int Minimum number of nodes in a leaf, defaults to 8

Example

> from pymatting import * import numpy as np data*set = np.random.randn(100, 2) tree = KDTree(data*set.astype(np.float32))*

- `query(self, query_points, k)`
  *Line 285: Query the tree

Parameters

query_points: numpy.ndarray (of type `np.float32`) Data points for which the next neighbours should be calculated k: int Number of neighbors to find

Returns

distances: numpy.ndarray Distances to the neighbors indices: numpy.ndarray Indices of the k nearest neighbors in original data array

Example

> from pymatting import * import numpy as np data*set = np.random.randn(100, 2) tree = KDTree(data*set.astype(np.float32)) tree.query(np.array([[0.5,0.5]], dtype=np.float32), k=3) (array([[0.14234178, 0.15879704, 0.26760164]], dtype=float32), array([[29, 21, 20]]))*

File: `timer.py`

Classes

`Timer` (Line 4)

Timer for benchmarking

Methods: - `__init__(self)`
Line 7: Starts a timer

- `stop(self, message)`
  *Line 12: Return and print time since last stop-call or initialization. Also print elapsed time if message is provided.

Parameters

message: str Message to print in front of passed seconds

Example

> from pymatting import * t = Timer() t.stop() 2.6157200919999966 t = Timer() t.stop('Test') Test - 11.654551 seconds 11.654551381000001*

File: `util.py`

Functions

`apply_to_channels(single_channel_func)` (Line 9)

Creates a new function which operates on each channel

Parameters

single*channel*func: function Function that acts on a single color channel

Returns

channel_func: function The same function that operates on all color channels

Example

> from pymatting import * import numpy as np from scipy.signal import convolve2d single*channel*fun = lambda x: convolve2d(x, np.ones((3, 3)), 'valid') multi*channel*fun = apply*to*channels(single*channel*fun) I = np.random.rand(480, 320, 3) multi*channel*fun(I).shape (478, 318, 3)

`vec_vec_dot(a, b)` **(Line 55)**

Computes the dot product of two vectors.

Parameters

a: numpy.ndarray First vector (if np.ndim(a) > 1 the function calculates the product for the two last axes) b: numpy.ndarray Second vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: scalar Dot product of `a` and `b`

Example

> import numpy as np from pymatting import * a = np.ones(2) b = np.ones(2) vec*vec*dot(a,b) 2.0

`mat_vec_dot(A, b)` **(Line 82)**

Calculates the matrix vector product for two arrays.

Parameters

A: numpy.ndarray Matrix (if np.ndim(A) > 2 the function calculates the product for the two last axes) b: numpy.ndarray Vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: numpy.ndarray Matrix vector product of both arrays

Example

import numpy as np from pymatting import * A = np.eye(2) b = np.ones(2)
mat*vec*dot(A,b) array([1., 1.])

`vec_vec_outer(a, b)` **(Line 109)**

Computes the outer product of two vectors

a: numpy.ndarray First vector (if np.ndim(b) > 1 the function calculates the product for the two last axes) b: numpy.ndarray Second vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: numpy.ndarray Outer product of `a` and `b` as numpy.ndarray

Example

import numpy as np from pymatting import * a = np.arange(1,3) b = np.arange(1,3)
vec*vec*outer(a,b) array([[1, 2], [2, 4]])

`fix_trimap(trimap, lower_threshold, upper_threshold)` **(Line 135)**

Fixes broken trimap :math: `T` by thresholding the values

.. math:: T^{\text{fixed}}*{ij}= \begin{cases} 0,&\text{if } T*{ij}<\text{lower_threshold}\ 1,&\text{if }T_{ij}>\text{upper_threshold}\ 0.5, &\text{otherwise}.\ \end{cases}

Parameters

trimap: numpy.ndarray Possibly broken trimap lower*threshold: float Threshold used to determine background pixels, defaults to 0.1 upper*threshold: float Threshold used to determine foreground pixels, defaults to 0.9

Returns

fixed_trimap: numpy.ndarray Trimap having values in :math: `\{0, 0.5, 1\}`

Example

from pymatting import * import numpy as np trimap = np.array([0,0.1, 0.4, 0.9, 1])
fix_trimap(trimap, 0.2, 0.8) array([0. , 0. , 0.5, 1. , 1.])

`isiterable(obj)` **(Line 186)**

Checks if an object is iterable

Parameters

obj: object Object to check

Returns

is_iterable: bool Boolean variable indicating whether the object is iterable

Example

from pymatting import * l = [] isiterable(l) True

`_resize_pil_image(image, size, resample)` **(Line 213)**

No docstring

`load_image(path, mode, size, resample)` **(Line 232)**

This function can be used to load an image from a file.

Parameters

path: str Path of image to load. mode: str Can be "GRAY", "RGB" or something else (see PIL.convert())

Returns

image: numpy.ndarray Loaded image

`save_image(path, image, make_directory)` **(Line 263)**

Given a path, save an image there.

Parameters

path: str Where to save the image. image: numpy.ndarray, dtype in [np.uint8, np.float32, np.float64] Image to save. Images of float dtypes should be in range [0, 1]. Images of uint8 dtype should be in range [0, 255] make_directory: bool Whether to create the directories needed for the image path.

`to_rgb8(image)` **(Line 290)**

Convertes an image to rgb8 color space

Parameters

image: numpy.ndarray Image to convert

Returns

image: numpy.ndarray Converted image with same height and width as input image but with three color channels

Example

from pymatting import * import numpy as np I = np.eye(2) to_rgb8(I) array([[[255, 255, 255], [0, 0, 0]], [[0, 0, 0], [255, 255, 255]]], dtype=uint8) `make_grid(images, nx, ny, dtype)` **(Line 334)**

Plots a grid of images.

Parameters

images : list of numpy.ndarray List of images to plot nx: int Number of rows ny: int Number of columns dtype: type Data type of output array

Returns

grid: numpy.ndarray Grid of images with datatype `dtype`

`show_images(images)` **(Line 421)**

Plot grid of images.

Parameters

images : list of numpy.ndarray List of images to plot height : int, matrix Height in pixels the output grid, defaults to 512

`trimap_split(trimap, flatten, bg_threshold, fg_threshold)` **(Line 439)**

This function splits the trimap into foreground pixels, background pixels, and unknown pixels.

Foreground pixels are pixels where the trimap has values larger than or equal to `fg_threshold` (default: 0.9). Background pixels are pixels where the trimap has values smaller than or equal to `bg_threshold` (default: 0.1). Pixels with other values are assumed to be unknown.

Parameters

trimap: numpy.ndarray Trimap with shape :math: `h \times w` flatten: bool If true np.flatten is called on the trimap

Returns

is*fg: numpy.ndarray Boolean array indicating which pixel belongs to the foreground is*bg: numpy.ndarray Boolean array indicating which pixel belongs to the background is*known: numpy.ndarray Boolean array indicating which pixel is known is*unknown: numpy.ndarray Boolean array indicating which pixel is unknown bg*threshold: float Pixels with smaller trimap values will be considered background. fg*threshold: float Pixels with larger trimap values will be considered foreground.

Example

> import numpy as np from pymatting import * trimap = np.array([[1,0],[0.5,0.2]]) is*fg, is*bg, is*known, is*unknown = trimap*split(trimap) is*fg array([True, False, False, False]) is*bg array([False, True, False, False]) is*known array([True, True, False, False]) is_unknown array([False, False, True, True])

`sanity_check_image(image)` **(Line 528)**

Performs a sanity check for input images. Image values should be in the range [0, 1], the `dtype` should be `np.float32` or `np.float64` and the image shape should be `(?, ?, 3)`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times 3`

Example

> import numpy as np from pymatting import check*image image = (np.random.randn(64, 64, 2) * 255).astype(np.int32) sanity*check_image(image) **main**:1: UserWarning: Expected RGB image of shape (?, ?, 3), but image.shape is (64, 64, 2). **main**:1: UserWarning: Image values should be in [0, 1], but image.min() is -933. **main**:1: UserWarning: Image values should be in [0, 1], but image.max() is 999. **main**:1: UserWarning: Unexpected image.dtype int32. Are you sure that you do not want to use np.float32 or np.float64 instead?

`blend(foreground, background, alpha)` (Line 581)

This function composes a new image for given foreground image, background image and alpha matte.

This is done by applying the composition equation

.. math:: I = \alpha F + (1-\alpha)B.

Parameters

foreground: numpy.ndarray Foreground image background: numpy.ndarray Background image alpha: numpy.ndarray Alpha matte

Returns

image: numpy.ndarray Composed image as numpy.ndarray

Example

> from pymatting import * foreground = load*image("data/lemur/lemur*foreground.png", "RGB") background = load*image("data/lemur/beach.png", "RGB") alpha = load*image("data/lemur/lemur_alpha.png", "GRAY") I = blend(foreground, background, alpha)

`stack_images()` (Line 617)

This function stacks images along the third axis. This is useful for combining e.g. rgb color channels or color and alpha channels.

Parameters

*images: numpy.ndarray Images to be stacked.

Returns

image: numpy.ndarray Stacked images as numpy.ndarray

Example

> from pymatting.util.util import stack*images import numpy as np I = stack*images(np.random.rand(4,5,3), np.random.rand(4,5,3)) I.shape (4, 5, 6)

`row_sum(A)` **(Line 646)**

Calculate the sum of each row of a matrix

Parameters

A: np.ndarray or scipy.sparse.spmatrix Matrix to sum rows of

Returns

row_sums: np.ndarray Vector of summed rows

Example

> from pymatting import * import numpy as np A = np.random.rand(2,2) A array([[0.62750946, 0.12917617], [0.8599449 , 0.5777254]]) row_sum(A) array([0.75668563, 1.4376703])

`normalize_rows(A, threshold)` **(Line 675)**

Normalize the rows of a matrix

Rows with sum below threshold are left as-is.

Parameters

A: scipy.sparse.spmatrix Matrix to normalize threshold: float Threshold to avoid division by zero

Returns

A: scipy.sparse.spmatrix Matrix with normalized rows

Example

> from pymatting import * import numpy as np A = np.arange(4).reshape(2,2) normalize_rows(A) array([[0. , 1.], [0.4, 0.6]])

`grid_coordinates(width, height, flatten)` **(Line 715)**

Calculates image pixel coordinates for an image with a specified shape

Parameters

width: int Width of the input image height: int Height of the input image flatten: bool Whether the array containing the coordinates should be flattened or not, defaults to False

Returns

x: numpy.ndarray x coordinates y: numpy.ndarray y coordinates

Example

> from pymatting import * x, y = grid_coordinates(2,2) x array([[0, 1], [0, 1]]) y array([[0, 0], [1, 1]])

`sparse_conv_matrix_with_offsets(width, height, kernel, dx, dy)` **(Line 757)**

Calculates a convolution matrix that can be applied to a vectorized image

Additionally, this function allows to specify which pixels should be used for the convoltion, i.e.

.. math:: \left(I * K\right)*{ij} = \sum*k K*k I*{i+{\Delta*y*}k,j+{\Delta*y*}k},

where :math: `K` is the flattened convolution kernel.

Parameters

width: int Width of the input image height: int Height of the input image kernel: numpy.ndarray Convolutional kernel dx: numpy.ndarray Offset in x direction dy: nunpy.ndarray Offset in y direction

Returns

M: scipy.sparse.csr_matrix Convolution matrix

`sparse_conv_matrix(width, height, kernel)` **(Line 807)**

Calculates a convolution matrix that can be applied to a vectorized image

Parameters

width: int Width of the input image height: int Height of the input image kernel: numpy.ndarray Convolutional kernel

Returns

M: scipy.sparse.csr_matrix Convolution matrix

Example

> from pymatting import * import numpy as np sparse*conv*matrix(3,3,np.ones((3,3))) <9x9 sparse matrix of type '' with 49 stored elements in Compressed Sparse Row format>

`weights_to_laplacian(W, normalize, regularization)` **(Line 840)**

Calculates the random walk normalized Laplacian matrix from the weight matrix

Parameters

W: numpy.ndarray Array of weights normalize: bool Whether the rows of W should be normalized to 1, defaults to True regularization: float Regularization strength, defaults to 0, i.e. no regularizaion

Returns

L: scipy.sparse.spmatrix Laplacian matrix

Example

> from pymatting import * import numpy as np weights*to*laplacian(np.ones((4,4))) matrix([[0.75, -0.25, -0.25, -0.25], [-0.25, 0.75, -0.25, -0.25], [-0.25, -0.25, 0.75, -0.25], [-0.25, -0.25, -0.25, 0.75]])

`normalize(values)` **(Line 878)**

Normalizes an array such that all values are between 0 and 1

Parameters

values: numpy.ndarray Array to normalize

Returns

result: numpy.ndarray Normalized array

Example

> from pymatting import * import numpy as np normalize(np.array([0, 1, 3, 10])) array([0. , 0.1, 0.3, 1.])

`div_round_up(x, n)` **(Line 904)**

Divides a number x by another integer n and rounds up the result

Parameters

x: int Numerator n: int Denominator

Returns

result: int Result

Example

from pymatting import * div*round*up(3,2) 2

`remove_background_bicolor(image, fg_color, bg_color)` **(Line 928)**

Remove background from image with at most two colors. Might not work if image has more than two colors.

Parameters

image: numpy.ndarray RGB input image fg*color: numpy.ndarray RGB Foreground color bg*color: numpy.ndarray RGB Background color

Returns

output: numpy.ndarray RGBA output image

Example

from pymatting import * import numpy as np image = np.random.rand(480, 320, 3) fg*color = np.random.rand(3) bg*color = np.random.rand(3) output = remove*background*bicolor(image, fg*color, bg*color) print(output.shape) (480, 320, 4)

`multi_channel_func(image)` **(Line 35)**

No docstring

File: `__init__.py`

File: `build.py`

Functions

`generate_html(node, references, html)` **(Line 7)**

No docstring

`main()` **(Line 157)**

No docstring

`write_website(html_path, title, content)` **(Line 226)**

No docstring

File: `highlight.py`

Functions

`group(x)` **(Line 3)**

No docstring

`non_capturing_group(x)` **(Line 6)**

No docstring

`named_group(name, x)` **(Line 9)**

No docstring

`opt(x)` **(Line 12)**

No docstring

`any_of()` **(Line 15)**

No docstring

`escape(x)` **(Line 18)**

No docstring

`indentation(line)` **(Line 66)**

No docstring

`remove_too_much_identation(code)` **(Line 69)**

No docstring

`highlight(code, output)` **(Line 75)**

No docstring

`highlight_inline(code)` **(Line 83)**

No docstring

`highlight_block(code)` **(Line 90)**

No docstring

File: `parse_bib.py`

Functions

`parse_bib(text)` **(Line 4)**

No docstring

`main()` **(Line 156)**

No docstring

`replace(match)` **(Line 71)**

No docstring

File: `parse_markdown.py`

Classes

`Stream` **(Line 4)**

No docstring

Methods: - `__init__(self, text)`
Line 5: No docstring

- `peek(self, n)`
  Line 9: No docstring

- `consume(self, n)`
  Line 12: No docstring

- `available(self)`
  Line 17: No docstring

- `skip(self, n)`
  Line 20: No docstring

- `__bool__(self)`
  Line 24: No docstring

- `match(self, pattern, flags)`
  Line 27: No docstring

- `match_consume(self, pattern, flags)`
  Line 31: No docstring

File: `util.py`

Classes

`HTML` **(Line 22)**

No docstring

Methods: - `__init__(self, value)`
Line 23: No docstring

- `__str__(self)`
  Line 26: No docstring

File: `make_frames.py`

Classes

`FrameWriterCallback` **(Line 10)**

No docstring

Methods: - `__init__(self)`
Line 11: No docstring

- `__call__(self, A, x, b, norm_b, r, norm_r)`
  Line 14: No docstring

File: `advanced_example.py`

File: `expert_example.py`

File: `lemur_at_the_beach.py`

File: `simple_example.py`

File: `__about__.py`

File: `__init__.py`

File: `estimate_alpha_cf.py`

Functions

`estimate_alpha_cf(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` **(Line 8)**

Estimate alpha from an input image and an input trimap using Closed-Form Alpha Matting as proposed by :cite: `levin2007closed`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: ichol) laplacian*kwargs: dictionary Arguments passed to the :code: `cf_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver is_known: numpy.ndarray Binary mask of pixels for which to compute the laplacian matrix. Providing this parameter might improve performance if few pixels are unknown.

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alphacf(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_knn.py`

Functions

`estimate_alpha_knn(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` (Line 9)

Estimate alpha from an input image and an input trimap using KNN Matting similar to :cite: `chen2013knn`. See `knn_laplacian` for more details.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: jacobi) laplacian*kwargs: dictionary Arguments passed to the :code: `knn_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*knn(... image, ... trimap, ... laplacian*kwargs={"n*neighbors": [15, 10]}, ... cg*kwargs={"maxiter":2000})

File: `estimate_alpha_lbdm.py`

Functions

`estimate_alpha_lbdm(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` (Line 9)

Estimate alpha from an input image and an input trimap using Learning Based Digital Matting as proposed by :cite: `zheng2009learning`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: ichol) laplacian*kwargs: dictionary Arguments passed to the :code:* `lbdm_laplacian` *function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*lbdm(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_lkm.py`

Functions

`estimate_alpha_lkm(image, trimap, laplacian_kwargs, cg_kwargs)` (Line 8)

Estimate alpha from an input image and an input trimap as described in Fast Matting Using Large Kernel Matting Laplacian Matrices by :cite: `he2010fast` .

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image laplacian*kwargs: dictionary Arguments passed to the :code:* `lkm_laplacian` *function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*lkm(... image, ... trimap, ... laplacian*kwargs={"epsilon": 1e-6, "radius": 15}, ... cg_kwargs={"maxiter":2000})

`A_matvec(x)` (Line 54)

No docstring

`jacobi(x)` (Line 57)

No docstring

File: `estimate_alpha_rw.py`

Functions

`estimate_alpha_rw(image, trimap, preconditioner, laplacian_kwargs, cg_kwargs)` (Line 9)

Estimate alpha from an input image and an input trimap using Learning Based Digital Matting as proposed by :cite: `grady2005random`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image preconditioner: function or scipy.sparse.linalg.LinearOperator Function or sparse matrix that applies the preconditioner to a vector (default: jacobi) laplacian*kwargs: dictionary Arguments passed to the :code: `rw_laplacian` function cg*kwargs: dictionary Arguments passed to the :code: `cg` solver

Returns

alpha: numpy.ndarray Estimated alpha matte

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha = estimate*alpha*rw(.... image, ... trimap, ... laplacian*kwargs={"sigma": 0.03}, ... cg_kwargs={"maxiter":2000})

File: `estimate_alpha_sm.py`

Functions

`estimate_alpha_sm(image, trimap, return_foreground_background, trimap_expansion_radius, trimap_expansion_threshold, sample_gathering_angles, sample_gathering_weights, sample_gathering_Np_radius, sample_refinement_radius, local_smoothing_radius1, local_smoothing_radius2, local_smoothing_radius3, local_smoothing_sigma_sq1, local_smoothing_sigma_sq2, local_smoothing_sigma_sq3)` (Line 4)

Estimate alpha from an input image and an input trimap using Shared Matting as proposed by :cite: `GastalOliveira2010SharedMatting`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times d` for which the alpha matte should be estimated trimap: numpy.ndarray Trimap with shape :math: `h \times w` of the image return*foreground*background: numpy.ndarray Whether to return foreground and background estimate. They will be computed either way trimap*expansion*radius: int How much to expand trimap. trimap*expansion*threshold: float Which pixel colors are similar enough to expand trimap into sample*gathering*angles: int In how many directions to search for new samples.

sample*gathering*weights: Tuple[float, float, float, float] Weights for various cost functions sample*gathering*N*pradius: int Radius of Np function sample*refinement*radius: int Search region for better neighboring samples local*smoothing*radius1: int Radius for foreground/background smoothing local*smoothing*radius2: int Radius for confidence computation local*smoothing*radius3: int Radius for low frequency alpha computation local*smoothing*sigma*sq1: float Squared sigma value for foreground/background smoothing Defaults to :code: `(2 * local_smoothing_radius1 + 1)**2 / (9 * pi)` if not given local*smoothing*sigmasq2: float Squared sigma value for confidence computation local*smoothing*sigma*sq3: float Squared sigma value for low frequency alpha computation Defaults to :code: `(2 * local_smoothing_radius3 + 1)**2 / (9 * pi)` if not given

Returns

alpha: numpy.ndarray Estimated alpha matte foreground: numpy.ndarray Estimated foreground background: numpy.ndarray Estimated background

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") trimap = load*image("data/lemur/lemur*trimap.png", "GRAY") alpha, foreground, background = estimate*alpha*sm(... image, ... trimap, ... return*foreground*background=True, ... sample*gathering_angles=4)

`estimate_alpha(I, F, B)` **(Line 168)**

No docstring

`inner(a, b)` **(Line 186)**

No docstring

`Mp2(I, F, B)` **(Line 193)**

No docstring

`Np(image, x, y, F, B, r)` **(Line 203)**

No docstring

`Ep(image, px, py, sx, sy)` **(Line 214)**

No docstring

`dist(a, b)` **(Line 260)**

No docstring

`length(a)` **(Line 267)**

No docstring

`expand_trimap(expanded_trimap, trimap, image, k_i, k_c)` **(Line 271)**

No docstring

`sample_gathering(gathering_F, gathering_B, gathering_alpha, image, trimap, num_angles, eN, eA, ef, eb, Np_radius)` **(Line 302)**

No docstring

`sample_refinement(refined_F, refined_B, refined_alpha, gathering_F, gathering_B, image, trimap, radius)` **(Line 433)**

No docstring

`local_smoothing(final_F, final_B, final_alpha, refined_F, refined_B, refined_alpha, image, trimap, radius1, radius2, radius3, sigma_sq1, sigma_sq2, sigma_sq3)` **(Line 488)**

No docstring

File: `__init__.py`

File: `__init__.py`

File: `cutout.py`

Functions

`cutout(image_path, trimap_path, cutout_path)` **(Line 6)**

Generate a cutout image from an input image and an input trimap. This method is using closed-form alpha matting as proposed by :cite: `levin2007closed` and multi-level foreground extraction :cite: `germer2020multilevel` .

Parameters

image*path: str Path of input image trimap*path: str Path of input trimap cutout_path: str Path of output cutout image

Example

cutout("../data/lemur.png", "../data/lemur*trimap.png", "lemur*cutout.png")

File: `__init__.py`

File: `estimate_foreground_cf.py`

Functions

`estimate_foreground_cf(image, alpha, regularization, rtol, neighbors, return_background, foreground_guess, background_guess, ichol_kwargs, cg_kwargs)` **(Line 8)**

Estimates the foreground of an image given alpha matte and image.

This method is based on the publication :cite: `levin2007closed`.

Parameters

image: numpy.ndarray Input image with shape :math: `h \times w \times d` alpha: numpy.ndarray Input alpha matte with shape :math: `h \times w` regularization: float Regularization strength :math: `\epsilon`, defaults to :math: `10^{-5}` neighbors: list of tuples of ints List of relative positions that define the neighborhood of a pixel return*background: bool Whether to return the estimated background in addition to the foreground foreground*guess: numpy.ndarray An initial guess for the foreground image in order to accelerate convergence. Using input image by default. background*guess: numpy.ndarray An initial guess for the background image. Using input image by default. ichol*kwargs: dictionary Keyword arguments for the incomplete Cholesky preconditioner cg_kwargs: dictionary Keyword arguments for the conjugate gradient descent solver

Returns

F: numpy.ndarray Extracted foreground B: numpy.ndarray Extracted background (not returned by default)

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") alpha = load*image("data/lemur/lemur*alpha.png", "GRAY") F = estimate*foreground*cf(image, alpha, return*background=False) F, B = estimate*foreground*cf(image, alpha, return_background=True)

See Also

stack_images: This function can be used to place the foreground on a new background.

File: `estimate_foreground_ml.py`

Functions

`_resize_nearest_multichannel(dst, src)` **(Line 6)**

Internal method.

Resize image src to dst using nearest neighbors filtering. Images must have multiple color channels, i.e. :code: `len(shape) == 3`.

Parameters

dst: numpy.ndarray of type np.float32 output image src: numpy.ndarray of type np.float32 input image

`_resize_nearest(dst, src)` **(Line 33)**

Internal method.

Resize image src to dst using nearest neighbors filtering. Images must be grayscale, i.e. :code: `len(shape) == 3`.

Parameters

dst: numpy.ndarray of type np.float32 output image src: numpy.ndarray of type np.float32 input image

`_estimate_fb_ml(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, gradient_weight)` (Line 62)

No docstring

`estimate_foreground_ml(image, alpha, regularization, n_small_iterations, n_big_iterations, small_size, return_background, gradient_weight)` (Line 186)

Estimates the foreground of an image given its alpha matte.

See :cite: `germer2020multilevel` for reference.

Parameters

image: numpy.ndarray Input image with shape :math: `h \times w \times d` alpha: numpy.ndarray Input alpha matte shape :math: `h \times w` regularization: float Regularization strength :math: `\epsilon`, defaults to :math: `10^{-5}`. Higher regularization results in smoother colors. n*small*iterations: int Number of iterations performed on small scale, defaults to :math: `10` n*big*iterations: int Number of iterations performed on large scale, defaults to :math: `2` small*size: int Threshold that determines at which size `n_small_iterations` should be used return*background: bool Whether to return the estimated background in addition to the foreground gradient_weight: float Larger values enforce smoother foregrounds, defaults to :math: `1`

Returns

F: numpy.ndarray Extracted foreground B: numpy.ndarray Extracted background

Example

> from pymatting import * image = load*image("data/lemur/lemur.png", "RGB") alpha = load*image("data/lemur/lemur*alpha.png", "GRAY") F = estimate*foreground*ml(image, alpha, return*background=False) F, B = estimate*foreground*ml(image, alpha, return_background=True)

See Also

stack_images: This function can be used to place the foreground on a new background.

File: `estimate_foreground_ml_cupy.py`

Functions

`estimate_foreground_ml_cupy(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, block_size, return_background, to_numpy)` (Line 110)

See the :code: `estimate_foreground` method for documentation.

`resize_nearest(dst, src, w_src, h_src, w_dst, h_dst, depth)` **(Line 152)**

No docstring

File: `estimate_foreground_ml_pyopencl.py`

Functions

`estimate_foreground_ml_pyopencl(input_image, input_alpha, regularization, n_small_iterations, n_big_iterations, small_size, return_background)` **(Line 104)**

See the :code: `estimate_foreground` method for documentation.

`upload(array)` **(Line 115)**

No docstring

`alloc()` **(Line 123)**

No docstring

`download(device_buf, shape)` **(Line 127)**

No docstring

`resize_nearest(dst, src, w_src, h_src, w_dst, h_dst, depth)` **(Line 154)**

No docstring

File: `__init__.py`

File: `cf_laplacian.py`

Functions

`_cf_laplacian(image, epsilon, r, values, indices, indptr, is_known)` **(Line 6)**

No docstring

`cf_laplacian(image, epsilon, radius, is_known)` **(Line 132)**

This function implements the alpha estimator for closed-form alpha matting as proposed by :cite: `levin2007closed`.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` epsilon: float Regularization strength, defaults to :math: `10^{-7}`. Strong regularization improves convergence but results in smoother alpha mattes. radius: int Radius of local window size, defaults to :math: `1`, i.e. only adjacent pixels are considered. The size of the local window is given as

:math: `(2 r + 1)^2`, where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. is_known: numpy.ndarray Binary mask of pixels for which to compute the laplacian matrix. Laplacian entries for known pixels will have undefined values.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `knn_laplacian.py`

Functions

`knn_laplacian(image, n_neighbors, distance_weights, kernel)` **(Line 7)**

This function calculates the KNN matting Laplacian matrix similar to :cite: `chen2013knn`. We use a kernel of 1 instead of a soft kernel by default since the former is faster to compute and both produce almost identical results in all our experiments, which is to be expected as the soft kernel is very close to 1 in most cases.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` n*neighbors: list of ints Number of neighbors to consider. If :code: `len(n_neighbors)>1` multiple nearest neighbor calculations are done and merged, defaults to `[20, 10]`, i.e. first 20 neighbors are considered and in the second run :math: `10` neighbors. The pixel distances are then weighted by the :code: `distance_weights`. distance*weights: list of floats Weight of distance in feature vector, defaults to `[2.0, 0.1]`. kernel: str Must be either "binary" or "soft". Default is "binary".

Returns

L: scipy.sparse.spmatrix Matting Laplacian matrix

File: `laplacian.py`

Functions

`make_linear_system(L, trimap, lambda_value, return_c)` **(Line 5)**

This function constructs a linear system from a matting Laplacian by constraining the foreground and background pixels with a diagonal matrix `C` to values in the right-hand-side vector `b`. The constraints are weighted by a factor :math: `\lambda`. The linear system is given as

.. math::

A = L + \lambda C,

where :math: `C=\mathop{Diag}(c)` having :math: `c_i = 1` if pixel i is known and :math: `c_i = 0` otherwise. The right-hand-side :math: `b` is a vector with entries :math: `b_i = 1` is pixel is is a foreground pixel and :math: `b_i = 0` otherwise.

Parameters

L: scipy.sparse.spmatrix Laplacian matrix, e.g. calculated with :code: `lbdm_laplacian` function trimap: numpy.ndarray Trimap with shape :math: `h\times w` lambda*value: float Constraint penalty, defaults to 100 return*c: bool Whether to return the constraint matrix `C`, defaults to False

Returns

A: scipy.sparse.spmatrix Matrix describing the system of linear equations b: numpy.ndarray Vector describing the right-hand side of the system C: numpy.ndarray Vector describing the diagonal entries of the matrix `C`, only returned if `return_c` is set to True

File: `lbdm_laplacian.py`

Functions

`calculate_kernel_matrix(X, v)` (Line 6)

No docstring

`_lbdm_laplacian(image, epsilon, r)` (Line 16)

No docstring

`lbdm_laplacian(image, epsilon, radius)` (Line 64)

Calculate a Laplacian matrix based on :cite: `zheng2009learning`.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` epsilon: float Regularization strength, defaults to :math: `10^{-7}`. Strong regularization improves convergence but results in smoother alpha mattes. radius: int Radius of local window size, defaults to :math: `1`, i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2`, where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image.

Returns

L: scipy.sparse.csr_matrix Matting Laplacian

File: `lkm_laplacian.py`

Functions

`lkm_laplacian(image, epsilon, radius, return_diagonal)` (Line 6)

Calculates the Laplacian for large kernel matting :cite: `he2010fast`

Parameters

image: numpy.ndarray Image of shape :math: `h\times w \times 3` epsilons: float Regularization strength, defaults to :math: `10^{-7}` radius: int Radius of local window size, defaults to :math: `10` , i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2` , where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. return_diagonal: bool Whether to also return the diagonal of the laplacian, defaults to True

Returns

L*matvec: function Function that applies the Laplacian matrix to a vector diag*L: numpy.ndarray Diagonal entries of the matting Laplacian, only returns if `return_diagonal` is True

`L_matvec(p)` (Line 51)

No docstring

File: `rw_laplacian.py`

Functions

`_rw_laplacian(image, sigma, r)` (Line 7)

No docstring

`rw_laplacian(image, sigma, radius, regularization)` (Line 47)

This function implements the alpha estimator for random walk alpha matting as described in :cite: `grady2005random` .

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` sigma: float Sigma used to calculate the weights (see Equation 4 in :cite: `grady2005random`), defaults to :math: `0.033` radius: int Radius of local window size, defaults to :math: `1` , i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2` , where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image. regularization: float Regularization strength, defaults to :math: `10^{-8}` . Strong regularization improves convergence but results in smoother alpha matte.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `uniform_laplacian.py`

Functions

`uniform_laplacian(image, radius)` (Line 9)

This function returns a Laplacian matrix with all weights equal to one.

Parameters

image: numpy.ndarray Image with shape :math: `h\times w \times 3` radius: int Radius of local window size, defaults to 1, i.e. only adjacent pixels are considered. The size of the local window is given as :math: `(2 r + 1)^2`, where :math: `r` denotes the radius. A larger radius might lead to violated color line constraints, but also favors further propagation of information within the image.

Returns

L: scipy.sparse.spmatrix Matting Laplacian

File: `__init__.py`

File: `ichol.py`

Classes

`CholeskyDecomposition` (Line 148)

Cholesky Decomposition

Calling this object applies the preconditioner to a vector by forward and back substitution.

Parameters

Ltuple: tuple of numpy.ndarrays Tuple of array describing values, row indices and row pointers for Cholesky factor in the compressed sparse column format (csc)

Methods: - `__init__(self, Ltuple)`
Line 159: No docstring

- `L(self)`
  *Line 163: Returns the Cholesky factor

Returns

L: scipy.sparse.csc_matrix Cholesky factor*

- `__call__(self, b)`
  Line 175: No docstring

File: `jacobi.py`

Functions

`jacobi(A)` (Line 1)

Compute the Jacobi preconditioner function for the matrix A.

Parameters

A: np.array Input matrix to compute the Jacobi preconditioner for.

Returns

precondition_matvec: function Function which applies the Jacobi preconditioner to a vector

Example

> from pymatting import * import numpy as np A = np.array([[2, 3], [3, 5]]) preconditioner = jacobi(A) preconditioner(np.array([1, 2])) array([0.5, 0.4])

`precondition_matvec(x)` **(Line 28)**

No docstring

File: `vcycle.py`

Functions

`make_P(shape)` **(Line 6)**

No docstring

`jacobi_step(A, A_diag, b, x, num_iter, omega)` **(Line 32)**

No docstring

`_vcycle_step(A, b, shape, cache, num_pre_iter, num_post_iter, omega, direct_solve_size)` **(Line 46)**

No docstring

`vcycle(A, shape, num_pre_iter, num_post_iter, omega, direct_solve_size, cache)` **(Line 103)**

Implements the V-Cycle preconditioner. The V-Cycle solver was recommended by :cite: `lee2014scalable` to solve the alpha matting problem.

Parameters

A: numpy.ndarray Input matrix shape: tuple of ints Describing the height and width of the image num*pre*iter: int Number of Jacobi iterations before each V-Cycle, defaults to 1 num*post*iter: int Number of Jacobi iterations after each V-Cycle, defaults to 1 omega: float Weight parameter for the Jacobi method. If method fails to converge, try different values.

Returns

precondition: function Function which applies the V-Cycle preconditioner to a vector

Example

from pymatting import * import numpy as np from scipy.sparse import csc_matrix A = np.array([[2, 3], [3, 5]]) preconditioner = vcycle(A, (2, 2)) preconditioner(np.array([1, 2])) array([-1., 1.])

`precondition(r)` **(Line 148)**

No docstring

File: `__init__.py`

File: `callback.py`

Classes

`CounterCallback` **(Line 1)**

Callback to count number of iterations of iterative solvers.

Methods: - `__init__(self)`
Line 4: No docstring

- `__call__(self, A, x, b, norm_b, r, norm_r)`
  Line 7: No docstring

`ProgressCallback` **(Line 11)**

Callback to count number of iterations of iterative solvers. Also prints residual error.

Methods: - `__init__(self)`
Line 17: No docstring

- `__call__(self, A, x, b, norm_b, r, norm_r)`
  Line 20: No docstring

File: `cg.py`

Functions

`cg(A, b, x0, atol, rtol, maxiter, callback, M, reorthogonalize)` **(Line 4)**

Solves a system of linear equations :math: `Ax=b` using conjugate gradient descent :cite: `hestenes1952methods`

Parameters

A: scipy.sparse.csr*matrix Square matrix b: numpy.ndarray Vector describing the right-hand side of the system x0: numpy.ndarray Initialization, if* `None` *then :code:* `x=np.zeros_like(b)` *atol: float Absolute tolerance. The loop terminates if the :math:* `||r||` *is smaller than* `atol`*, where :math:* `r` *denotes the residual of the current iterate. rtol: float Relative tolerance. The loop terminates if :math:* `{||r||}/{||b||}` *is smaller than* `rtol`*, where :math:* `r` *denotes the* *residual of the current iterate. callback: function Function :code:* `callback(A, x, b, norm_b, r, norm_r)` *called after each iteration, defaults to* `None` *M: function or scipy.sparse.csr*matrix Function that applies the preconditioner to a vector. Alternatively, `M` can be a matrix describing the precondioner. reorthogonalize: boolean Whether to apply reorthogonalization of the residuals after each update, defaults to `False`

Returns

x: numpy.ndarray Solution of the system

Example

> from pymatting import * import numpy as np A = np.array([[3.0, 1.0], [1.0, 2.0]]) M = jacobi(A) b = np.array([4.0, 3.0]) cg(A, b, M=M) array([1., 1.])

`precondition(x)` **(Line 54)**

No docstring

`precondition(x)` **(Line 61)**

No docstring

File: `__init__.py`

File: `boxfilter.py`

Functions

`boxfilter_rows_valid(src, r)` **(Line 7)**

No docstring

`boxfilter_rows_same(src, r)` **(Line 32)**

No docstring

`boxfilter_rows_full(src, r)` **(Line 61)**

No docstring

`boxfilter(src, radius, mode)` **(Line 90)**

Computes the boxfilter (uniform blur, i.e. blur with kernel :code:`np.ones(radius, radius)`) of an input image.

Depending on the mode, the input image of size :math: `(h, w)` is either of shape

- :math: `(h - 2 r, w - 2 r)` in case of 'valid' mode
- :math: `(h, w)` in case of 'same' mode
- :math: `(h + 2 r, w + 2 r)` in case of 'full' mode

.. image:: figures/padding.png

Parameters

src: numpy.ndarray Input image having either shape :math: `h \times w \times d` or :math: `h \times w` radius: int Radius of boxfilter, defaults to :math: `3` mode: str One of 'valid', 'same' or 'full', defaults to 'same'

Returns

dst: numpy.ndarray Blurred image

Example

> from pymatting import * import numpy as np boxfilter(np.eye(5), radius=2, mode="valid") array([[5.]]) boxfilter(np.eye(5), radius=2, mode="same") array([[3., 3., 3., 2., 1.], [3., 4., 4., 3., 2.], [3., 4., 5., 4., 3.], [2., 3., 4., 4., 3.], [1., 2., 3., 3., 3.]]) boxfilter(np.eye(5), radius=2, mode="full") array([[1., 1., 1., 1., 1., 0., 0., 0., 0.], [1., 2., 2., 2., 2., 1., 0., 0., 0.], [1., 2., 3., 3., 3., 2., 1., 0., 0.], [1., 2., 3., 4., 4., 3., 2., 1., 0.], [1., 2., 3., 4., 5., 4., 3., 2., 1.], [0., 1., 2., 3., 4., 4., 3., 2., 1.], [0., 0., 1., 2., 3., 3., 3., 2., 1.], [0., 0., 0., 1., 2., 2., 2., 2., 1.], [0., 0., 0., 0., 1., 1., 1., 1., 1.]])

File: `distance.py`

Functions

`_propagate_1d_first_pass(d)` **(Line 6)**

No docstring

`_propagate_1d(d, v, z, f)` **(Line 18)**

No docstring

`_propagate_distance(distance)` **(Line 62)**

No docstring

`distance_transform(mask)` **(Line 76)**

For every non-zero value, compute the distance to the closest zero value. Based on :cite: `felzenszwalb2012distance`.

Parameters

mask: numpy.ndarray 2D matrix of zero and nonzero values.

Returns

distance: numpy.ndarray Distance to closest zero-valued pixel.

Example

> from pymatting import * import numpy as np mask = np.random.rand(10, 20) < 0.9

distance = distance_transform(mask)

File: `kdtree.py`

Classes

`KDTree` (Line 236)

KDTree implementation

Methods: - `__init__(self, data_points, min_leaf_size)`
*Line 239: Constructs a KDTree for given data points. The implementation currently only supports data type `np.float32`.

Parameters

data*points: numpy.ndarray (of type `np.float32`) Dataset with shape :math: `n \times d`, where :math: `n` is the number of data points in the data set and :math: `d` is the dimension of each data point min*leaf_size: int Minimum number of nodes in a leaf, defaults to 8

Example

from pymatting import * import numpy as np data*set = np.random.randn(100, 2) tree = KDTree(data*set.astype(np.float32))*

- `query(self, query_points, k)`
  *Line 285: Query the tree

Parameters

query_points: numpy.ndarray (of type `np.float32`) Data points for which the next neighbours should be calculated k: int Number of neighbors to find

Returns

distances: numpy.ndarray Distances to the neighbors indices: numpy.ndarray Indices of the k nearest neighbors in original data array

Example

from pymatting import * import numpy as np data*set = np.random.randn(100, 2) tree = KDTree(data*set.astype(np.float32)) tree.query(np.array([[0.5,0.5]], dtype=np.float32), k=3) (array([[0.14234178, 0.15879704, 0.26760164]], dtype=float32), array([[29, 21, 20]]))*

File: `timer.py`

Classes

`Timer` **(Line 4)**

Timer for benchmarking

Methods: - `__init__(self)`
Line 7: Starts a timer

- `stop(self, message)`
  *Line 12: Return and print time since last stop-call or initialization. Also print elapsed time if message is provided.

Parameters

message: str Message to print in front of passed seconds

Example

> from pymatting import * t = Timer() t.stop() 2.6157200919999966 t = Timer() t.stop('Test') Test - 11.654551 seconds 11.654551381000001*

File: `util.py`

Functions

`apply_to_channels(single_channel_func)` **(Line 9)**

Creates a new function which operates on each channel

Parameters

single*channel*func: function Function that acts on a single color channel

Returns

channel_func: function The same function that operates on all color channels

Example

> from pymatting import * import numpy as np from scipy.signal import convolve2d single*channel*fun = lambda x: convolve2d(x, np.ones((3, 3)), 'valid') multi*channel*fun = apply*to*channels(single*channel*fun) I = np.random.rand(480, 320, 3) multi*channel*fun(I).shape (478, 318, 3)

`vec_vec_dot(a, b)` **(Line 55)**

Computes the dot product of two vectors.

Parameters

a: numpy.ndarray First vector (if np.ndim(a) > 1 the function calculates the product for the two last axes) b: numpy.ndarray

Second vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: scalar Dot product of `a` and `b`

Example

> import numpy as np from pymatting import * a = np.ones(2) b = np.ones(2)
> vec*vec*dot(a,b) 2.0

`mat_vec_dot(A, b)` **(Line 82)**

Calculates the matrix vector product for two arrays.

Parameters

A: numpy.ndarray Matrix (if np.ndim(A) > 2 the function calculates the product for the two last axes) b: numpy.ndarray Vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: numpy.ndarray Matrix vector product of both arrays

Example

> import numpy as np from pymatting import * A = np.eye(2) b = np.ones(2)
> mat*vec*dot(A,b) array([1., 1.])

`vec_vec_outer(a, b)` **(Line 109)**

Computes the outer product of two vectors

a: numpy.ndarray First vector (if np.ndim(b) > 1 the function calculates the product for the two last axes) b: numpy.ndarray Second vector (if np.ndim(b) > 1 the function calculates the product for the two last axes)

Returns

product: numpy.ndarray Outer product of `a` and `b` as numpy.ndarray

Example

> import numpy as np from pymatting import * a = np.arange(1,3) b = np.arange(1,3)
> vec*vec*outer(a,b) array([[1, 2], [2, 4]])

`fix_trimap(trimap, lower_threshold, upper_threshold)` **(Line 135)**

Fixes broken trimap :math: `T` by thresholding the values

.. math:: T^{\text{fixed}}*{ij}= \begin{cases} 0,&\text{if } T*{ij}<\text{lower_threshold}\ 1,&\text{if }T_{ij}>\text{upper_threshold}\ 0.5, &\text{otherwise}.\ \end{cases}

Parameters

trimap: numpy.ndarray Possibly broken trimap lower*threshold: float Threshold used to determine background pixels, defaults to 0.1 upper*threshold: float Threshold used to determine foreground pixels, defaults to 0.9

Returns

fixed_trimap: numpy.ndarray Trimap having values in :math:`\{0, 0.5, 1\}`

Example

> from pymatting import * import numpy as np trimap = np.array([0,0.1, 0.4, 0.9, 1]) fix_trimap(trimap, 0.2, 0.8) array([0. , 0. , 0.5, 1. , 1.])

`isiterable(obj)` **(Line 186)**

Checks if an object is iterable

Parameters

obj: object Object to check

Returns

is_iterable: bool Boolean variable indicating whether the object is iterable

Example

> from pymatting import * l = [] isiterable(l) True

`_resize_pil_image(image, size, resample)` **(Line 213)**

No docstring

`load_image(path, mode, size, resample)` **(Line 232)**

This function can be used to load an image from a file.

Parameters

path: str Path of image to load. mode: str Can be "GRAY", "RGB" or something else (see PIL.convert())

Returns

image: numpy.ndarray Loaded image

`save_image(path, image, make_directory)` **(Line 263)**

Given a path, save an image there.

Parameters

path: str Where to save the image. image: numpy.ndarray, dtype in [np.uint8, np.float32, np.float64] Image to save. Images of float dtypes should be in range [0, 1]. Images of uint8 dtype should be in range [0, 255] make_directory: bool Whether to create the directories needed for the image path.

`to_rgb8(image)` **(Line 290)**

Convertes an image to rgb8 color space

Parameters

image: numpy.ndarray Image to convert

Returns

image: numpy.ndarray Converted image with same height and width as input image but with three color channels

Example

> from pymatting import * import numpy as np I = np.eye(2) to_rgb8(I) array([[[255, 255, 255], [0, 0, 0]], [[0, 0, 0], [255, 255, 255]]], dtype=uint8)

`make_grid(images, nx, ny, dtype)` **(Line 334)**

Plots a grid of images.

Parameters

images : list of numpy.ndarray List of images to plot nx: int Number of rows ny: int Number of columns dtype: type Data type of output array

Returns

grid: numpy.ndarray Grid of images with datatype `dtype`

`show_images(images)` **(Line 421)**

Plot grid of images.

Parameters

images : list of numpy.ndarray List of images to plot height : int, matrix Height in pixels the output grid, defaults to 512

`trimap_split(trimap, flatten, bg_threshold, fg_threshold)` **(Line 439)**

This function splits the trimap into foreground pixels, background pixels, and unknown pixels.

Foreground pixels are pixels where the trimap has values larger than or equal to `fg_threshold` (default: 0.9). Background pixels are pixels where the trimap has values smaller than or equal to `bg_threshold` (default: 0.1). Pixels with other values are assumed to be unknown.

Parameters

trimap: numpy.ndarray Trimap with shape :math: `h \times w` flatten: bool If true np.flatten is called on the trimap

Returns

is*fg: numpy.ndarray Boolean array indicating which pixel belongs to the foreground is*bg: numpy.ndarray Boolean array indicating which pixel belongs to the background is*known: numpy.ndarray Boolean array indicating which pixel is known is*unknown: numpy.ndarray Boolean array indicating which pixel is unknown bg*threshold: float Pixels with smaller trimap values will be considered background. fg*threshold: float Pixels with larger trimap values will be considered foreground.

Example

> import numpy as np from pymatting import * trimap = np.array([[1,0],[0.5,0.2]]) is*fg, is*bg, is*known, is*unknown = trimap*split(trimap) is*fg array([True, False, False, False]) is*bg array([False, True, False, False]) is*known array([True, True, False, False]) is_unknown array([False, False, True, True])

`sanity_check_image(image)` **(Line 528)**

Performs a sanity check for input images. Image values should be in the range [0, 1], the `dtype` should be `np.float32` or `np.float64` and the image shape should be `(?, ?, 3)`.

Parameters

image: numpy.ndarray Image with shape :math: `h \times w \times 3`

Example

> import numpy as np from pymatting import check*image image = (np.random.randn(64, 64, 2) * 255).astype(np.int32) sanity*check_image(image) **main**:1: UserWarning: Expected RGB image of shape (?, ?, 3), but image.shape is (64, 64, 2). **main**:1: UserWarning: Image values should be in [0, 1], but image.min() is -933. **main**:1: UserWarning: Image values should be in [0, 1], but image.max() is 999. **main**:1: UserWarning: Unexpected image.dtype int32. Are you sure that you do not want to use np.float32 or np.float64 instead?

`blend(foreground, background, alpha)` **(Line 581)**

This function composes a new image for given foreground image, background image and alpha matte.

This is done by applying the composition equation

.. math:: I = \alpha F + (1-\alpha)B.

Parameters

foreground: numpy.ndarray Foreground image background: numpy.ndarray Background image alpha: numpy.ndarray Alpha matte

Returns

image: numpy.ndarray Composed image as numpy.ndarray

Example

> from pymatting import * foreground = load*image("data/lemur/lemur*foreground.png", "RGB") background = load*image("data/lemur/beach.png", "RGB") alpha = load*image("data/lemur/lemur_alpha.png", "GRAY") I = blend(foreground, background, alpha)

`stack_images()` **(Line 617)**

This function stacks images along the third axis. This is useful for combining e.g. rgb color channels or color and alpha channels.

Parameters

*images: numpy.ndarray Images to be stacked.

Returns

image: numpy.ndarray Stacked images as numpy.ndarray

Example

> from pymatting.util.util import stack*images import numpy as np I = stack*images(np.random.rand(4,5,3), np.random.rand(4,5,3)) I.shape (4, 5, 6)

`row_sum(A)` **(Line 646)**

Calculate the sum of each row of a matrix

Parameters

A: np.ndarray or scipy.sparse.spmatrix Matrix to sum rows of

Returns

row_sums: np.ndarray Vector of summed rows

Example

from pymatting import * import numpy as np A = np.random.rand(2,2) A array([[0.62750946, 0.12917617], [0.8599449 , 0.5777254]]) row_sum(A) array([0.75668563, 1.4376703])

`normalize_rows(A, threshold)` **(Line 675)**

Normalize the rows of a matrix

Rows with sum below threshold are left as-is.

Parameters

A: scipy.sparse.spmatrix Matrix to normalize threshold: float Threshold to avoid division by zero

Returns

A: scipy.sparse.spmatrix Matrix with normalized rows

Example

from pymatting import * import numpy as np A = np.arange(4).reshape(2,2) normalize_rows(A) array([[0. , 1.], [0.4, 0.6]])

`grid_coordinates(width, height, flatten)` **(Line 715)**

Calculates image pixel coordinates for an image with a specified shape

Parameters

width: int Width of the input image height: int Height of the input image flatten: bool Whether the array containing the coordinates should be flattened or not, defaults to False

Returns

x: numpy.ndarray x coordinates y: numpy.ndarray y coordinates

Example

from pymatting import * x, y = grid_coordinates(2,2) x array([[0, 1], [0, 1]]) y array([[0, 0], [1, 1]])

`sparse_conv_matrix_with_offsets(width, height, kernel, dx, dy)` **(Line 757)**

Calculates a convolution matrix that can be applied to a vectorized image

Additionally, this function allows to specify which pixels should be used for the convoltion, i.e.

.. math:: \left(I * K\right)*{ij} = \sum*k Kk *I*{i+{\Delta*y}k,j+{\Delta*y}k},

where :math:`K` is the flattened convolution kernel.

Parameters

width: int Width of the input image height: int Height of the input image kernel: numpy.ndarray Convolutional kernel dx: numpy.ndarray Offset in x direction dy: nunpy.ndarray Offset in y direction

Returns

M: scipy.sparse.csr_matrix Convolution matrix

`sparse_conv_matrix(width, height, kernel)` **(Line 807)**

Calculates a convolution matrix that can be applied to a vectorized image

Parameters

width: int Width of the input image height: int Height of the input image kernel: numpy.ndarray Convolutional kernel

Returns

M: scipy.sparse.csr_matrix Convolution matrix

Example

> from pymatting import * import numpy as np sparse*conv*matrix(3,3,np.ones((3,3))) <9x9 sparse matrix of type '' with 49 stored elements in Compressed Sparse Row format>

`weights_to_laplacian(W, normalize, regularization)` **(Line 840)**

Calculates the random walk normalized Laplacian matrix from the weight matrix

Parameters

W: numpy.ndarray Array of weights normalize: bool Whether the rows of W should be normalized to 1, defaults to True regularization: float Regularization strength, defaults to 0, i.e. no regularizaion

Returns

L: scipy.sparse.spmatrix Laplacian matrix

Example

> from pymatting import * import numpy as np weights*to*laplacian(np.ones((4,4))) matrix([[0.75, -0.25, -0.25, -0.25], [-0.25, 0.75, -0.25, -0.25], [-0.25, -0.25, 0.75, -0.25], [-0.25, -0.25, -0.25, 0.75]])

`normalize(values)` **(Line 878)**

Normalizes an array such that all values are between 0 and 1

Parameters

values: numpy.ndarray Array to normalize

Returns

result: numpy.ndarray Normalized array

Example

from pymatting import * import numpy as np normalize(np.array([0, 1, 3, 10])) array([0. , 0.1, 0.3, 1.])

`div_round_up(x, n)` **(Line 904)**

Divides a number x by another integer n and rounds up the result

Parameters

x: int Numerator n: int Denominator

Returns

result: int Result

Example

from pymatting import * div*round*up(3,2) 2

`remove_background_bicolor(image, fg_color, bg_color)` **(Line 928)**

Remove background from image with at most two colors. Might not work if image has more than two colors.

Parameters

image: numpy.ndarray RGB input image fg*color: numpy.ndarray RGB Foreground color bg*color: numpy.ndarray RGB Background color

Returns

output: numpy.ndarray RGBA output image

Example

from pymatting import * import numpy as np image = np.random.rand(480, 320, 3) fg*color = np.random.rand(3) bg*color = np.random.rand(3) output = remove*background*bicolor(image, fg*color, bg*color) print(output.shape) (480, 320, 4) `multi_channel_func(image)` **(Line 35)**

No docstring

File: `__init__.py`

File: `download_images.py`

Functions

`is_pymatting_root()` **(Line 7)**

No docstring

`download_files()` **(Line 37)**

No docstring

`extract_files()` **(Line 89)**

No docstring

`main()` **(Line 103)**

No docstring

File: `test_boxfilter.py`

Functions

`run_boxfilter(m, n, r, mode, n_runs)` **(Line 7)**

No docstring

`test_boxfilter()` **(Line 32)**

No docstring

File: `test_cg.py`

Functions

`test_cg()` **(Line 5)**

No docstring

`precondition(x)` **(Line 21)**

No docstring

File: `test_distance.py`

Functions

`distance_transform_naive(mask)` **(Line 7)**

No docstring

`distance_transform_naive_vectorized(mask)` **(Line 33)**

No docstring

`test_distance()` **(Line 45)**

No docstring

File: `test_estimate_alpha.py`

Functions

`test_alpha()` **(Line 5)**

No docstring

File: `test_foreground.py`

Functions

`test_foreground()` **(Line 11)**

No docstring

File: `test_ichol.py`

Functions

`test_ichol()` **(Line 6)**

No docstring

File: `test_kdtree.py`

Functions

`run_kdtree()` **(Line 7)**

No docstring

`test_kdtree()` **(Line 44)**

No docstring

File: `test_laplacians.py`

Functions

`test_laplacians()` **(Line 11)**

No docstring

File: `test_lkm.py`

Functions

`test_lkm()` **(Line 13)**

No docstring

`A_lkm(x)` **(Line 30)**

No docstring

`jacobi_lkm(r)` **(Line 35)**

No docstring

File: `test_preconditioners.py`

Functions

`test_preconditioners()` **(Line 14)**

No docstring

File: `test_remove_background_bicolor.py`

Functions

`test_remove_background_bicolor()` **(Line 5)**

No docstring

File: `test_simple_api.py`

Functions

`test_cutout()` **(Line 6)**

No docstring

File: `test_util.py`

Functions

`test_util()` **(Line 5)**

No docstring

`conv(image, kernel)` **(Line 7)**

No docstring