Contents

Chapter 1

CMakeLists.txt

```
project (fast_bilateral_space_stereo)

cmake_minimum_required(VERSION 3.0)
cmake_policy(VERSION 3.0)

set(CMAKE_CONFIGURATION_TYPES Debug Release CACHE TYPE INTERNAL FORCE )

list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

find_package(Eigen)
find_package(OpenCV 3)
find_package(Ceres)

# Source files
SET(FAST_BIL_SOURCES
    src/main.cpp
    src/bilateral_grid_simplified.cpp
    src/bilateral_grid_simplified.h
    src/fast_bilateral_solver.cpp
    src/fast_bilateral_solver.h
    src/stereo_matcher_birchfield_tomasi.cpp
    src/stereo_matcher_birchfield_tomasi.h
)

include_directories(${EIGEN_INCLUDE_DIR} ${OpenCV_INCLUDE_DIRS} ${GLOG_INCLUDE_DIR} ${
    CERES_INCLUDE_DIRS})

# Create the executable
add_executable(fast_bilateral_space_stereo ${FAST_BIL_SOURCES})
if(MSVC)
    set(LINK_LIBRARY ${CERES_LIBRARIES} ${OpenCV_LIBS} ${GLOG_LIBRARIES} shlwapi.lib)
else()
    set(LINK_LIBRARY ${CERES_LIBRARIES} ${OpenCV_LIBS} ${GLOG_LIBRARIES})
endif()
target_link_libraries(fast_bilateral_space_stereo ${LINK_LIBRARY})
```

Chapter 2

main.cpp

```
#define ENABLE_DOMAIN_TRANSFORM_FILTER // you need OpenCV extra modules for this, you can just
    comment this otherwise.

#include <string>

#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#ifdef ENABLE_DOMAIN_TRANSFORM_FILTER
#include <opencv2/ximgproc.hpp>
#endif

#include "bilateral_grid_simplified.h"
#include "stereo_matcher_birchfield_tomasi.h"
#include "fast_bilateral_solver.h"

#define USE_EXAMPLE_1
//#define USE_EXAMPLE_2

int main(int argc, char** argv)
{
  /// EXAMPLE 1
  ///
  ///
#ifdef USE_EXAMPLE_1
  const std::string image_pair_filename = "../data/middlebury_scenes2006_midd1.jpg";

  // bilateral grid properties
  const int property_grid_sigma_spatial = 16;
  const int property_grid_sigma_luma = 16;
  const int property_grid_sigma_chroma = 16;

  // stereo matching properties
  const int property_disparity_min = -50;
  const int property_disparity_max = 50;
  const stereo_matcher_birchfield_tomasi::block_filter_size property_stereo_block_filter =
      stereo_matcher_birchfield_tomasi::block_filter_size::size_5x5;

  // solver properties
  const int property_solver_nb_iterations = 500;
  const float property_solver_lambda = 0.2f;
  const int property_solver_keep_nb_of_intermediate_images = 0; // you can ignore this one,
      for debugging
                                          // post-process domain transform properties
  const float property_dt_sigmaSpatial = 40.0f;
  const float property_dt_sigmaColor = 220.0f;
  const int property_dt_numIters = 3;
#endif
  ///
  ///
  ///

  /// EXAMPLE 2
```

```
///
///
#ifdef USE_EXAMPLE_2
const std::string image_pair_filename = "http://urixblog.com/p/2012/2012.09.13me/picture-7.
    jpg";

// bilateral grid properties
const int property_grid_sigma_spatial = 18;
const int property_grid_sigma_luma = 16;
const int property_grid_sigma_chroma = 24;

// stereo matching properties
const int property_disparity_min = -18;
const int property_disparity_max = 18;
const stereo_matcher_birchfield_tomasi::block_filter_size property_stereo_block_filter =
    stereo_matcher_birchfield_tomasi::block_filter_size::size_5x5;

// solver properties
const int property_solver_nb_iterations = 50;
const float property_solver_lambda = 0.4f;
const int property_solver_keep_nb_of_intermediate_images = 0; // you can ignore this one,
    for debugging

// post-process domain transform properties
const float property_dt_sigmaSpatial = 40.0f;
const float property_dt_sigmaColor = 220.0f;
const int property_dt_numIters = 3;
#endif
///
///
///

// load stereo pair
cv::Mat stereo_images[2];
cv::Mat pair_image;
pair_image = cv::imread(image_pair_filename, CV_LOAD_IMAGE_COLOR);

if (pair_image.cols > 0)
{
  stereo_images[0] = pair_image(cv::Rect(0, 0, pair_image.cols >> 1, pair_image.rows)).clone
      ();
  stereo_images[1] = pair_image(cv::Rect(pair_image.cols >> 1, 0, pair_image.cols >> 1,
      pair_image.rows)).clone();
}
else
{
  std::cout << "failed to load " << image_pair_filename << std::endl;
  return -1;
}

cv::imshow("stereo image pair", pair_image);
cv::waitKey(16);

// convert to gray scale
cv::Mat stereo_images_gray[2];
cv::cvtColor(stereo_images[0], stereo_images_gray[0], CV_BGR2GRAY);
cv::cvtColor(stereo_images[1], stereo_images_gray[1], CV_BGR2GRAY);

// grid
bilateral_grid_simplified grid;
grid.init(stereo_images[0], property_grid_sigma_spatial, property_grid_sigma_luma,
    property_grid_sigma_chroma);

// stereo matching
stereo_matcher_birchfield_tomasi stereo_matcher;
stereo_matcher.get_parameters().disparity_min = property_disparity_min;
stereo_matcher.get_parameters().disparity_max = property_disparity_max;
stereo_matcher.get_parameters().filter_size = property_stereo_block_filter;
stereo_matcher.stereo_match(stereo_images_gray);
```

```
    // loss function
    std::vector<int> lookup;
    stereo_matcher.generate_data_loss_table(grid, lookup);

    ///// bilateral solver
    fast_bilateral_solver solver;

    // let's work from "0 --> disparity range" instead of "disparity min --> disparty max"
    // and let's use the minimum disparity image as a starting point.
    cv::Mat input_x = stereo_matcher.get_output().min_disp_image - stereo_matcher.get_parameters().disparity_min;
    cv::Mat input_x_fl;
    input_x.convertTo(input_x_fl, CV_32FC1);
    cv::Mat input_confidence_fl;
    stereo_matcher.get_output().conf_disp_image.convertTo(input_confidence_fl, CV_32FC1, 1.0f / 255.0f);

    // for initialization, let's apply a weighted bilateral filter!
    //   filtered image = blur(image x confidence) / blur(confidence)
    //   the confidence image is an image where a 1 means we have a match with the stereo matcher. 0 if there was no match.
    cv::Mat tc_im;
    cv::multiply(input_x_fl, input_confidence_fl, tc_im);
    cv::Mat tc = grid.filter(tc_im);
    cv::Mat c = grid.filter(input_confidence_fl);
    cv::Mat start_point_image;
    cv::divide(tc, c, start_point_image);

    //// decomment if you want to start with 0...
    ////start_point_image = cv::Scalar(0.f);

    cv::Mat final_disparty_image = solver.solve(start_point_image, grid, lookup,
      (stereo_matcher.get_parameters().disparity_max - stereo_matcher.get_parameters().disparity_min) + 1, property_solver_lambda, property_solver_nb_iterations, property_solver_keep_nb_of_intermediate_images);
    final_disparty_image += (float)stereo_matcher.get_parameters().disparity_min;

    // display disparity image
    cv::Mat adjmap_final;
    final_disparty_image.convertTo(adjmap_final, CV_8UC1,
      255.0 / (stereo_matcher.get_parameters().disparity_max - stereo_matcher.get_parameters().disparity_min),
      -stereo_matcher.get_parameters().disparity_min * 255.0f / (stereo_matcher.get_parameters().disparity_max - stereo_matcher.get_parameters().disparity_min));
    cv::imshow("disparity image", adjmap_final);

    // optional: apply domain transform to smoothen the disparity image
#ifdef ENABLE_DOMAIN_TRANSFORM_FILTER
    cv::Mat final_disparty_dtfiltered_image;
    cv::ximgproc::dtFilter(stereo_images[0],
      final_disparty_image, final_disparty_dtfiltered_image,
      property_dt_sigmaSpatial, property_dt_sigmaColor,
      cv::ximgproc::DTF_RF,
      property_dt_numIters);

    // display disparity image
    cv::Mat adjmap_dt;
    final_disparty_dtfiltered_image.convertTo(adjmap_dt, CV_8UC1,
      255.0 / (stereo_matcher.get_parameters().disparity_max - stereo_matcher.get_parameters().disparity_min),
      -stereo_matcher.get_parameters().disparity_min * 255.0f / (stereo_matcher.get_parameters().disparity_max - stereo_matcher.get_parameters().disparity_min));
    cv::imshow("disparity image + domain transform", adjmap_dt);
#endif

    cv::waitKey(0);
    return 0;
}
```

Chapter 3

bilateral_grid_simplified.h

```cpp
#pragma once

#include <Eigen/Sparse>

#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>

#include <string>
#include <chrono>
#include <unordered_map>

// A nave implementation of a simplified bilateral grid.
// There are many ways you can optimize this class.
class bilateral_grid_simplified
{
public:
  std::int32_t get_nb_vertices() const { return nb_vertices; }
  const Eigen::SparseMatrix<float, Eigen::RowMajor>& get_splat_matrix() const { return mat_splat; }
  const Eigen::SparseMatrix<float, Eigen::RowMajor>& get_slice_matrix() const { return mat_slice; }
  const Eigen::SparseMatrix<float, Eigen::RowMajor>& get_sblur_matrix() const { return mat_blur; }
  const Eigen::MatrixXf& get_normalizer_matrix() const { return mat_normalizer; }

  std::int32_t get_reference_nb_pixels() const { return nb_vertices; }
  std::int32_t get_reference_width() const { return nb_vertices; }
  std::int32_t get_reference_height() const { return nb_vertices; }
public:
  bilateral_grid_simplified();

  // Initialize the simplified grid.
  void init(const cv::Mat reference_bgr, const int sigma_spatial = 32, const int sigma_luma = 32, const int sigma_chroma = 32);

  // Apply a bilateral filter to an image
  // [in] input_image: expects a single channel floating-point image
  // returns a smoothed image
  cv::Mat filter(cv::Mat input_image);

  // Apply the splat and splice operation on an image
  // [in] input_image: expects a single channel floating-point image
  // returns a splat and spliced image
  cv::Mat splat_slice(cv::Mat input_image);

  // Apply the blur matrix
  // - [in] in: a matrix in bilateral space
  // returns a smoothed matrix
  Eigen::MatrixXf blur(Eigen::MatrixXf& in) const;
  Eigen::SparseMatrix<float, Eigen::RowMajor> blur(const Eigen::SparseMatrix<float, Eigen::RowMajor>& in) const;

```

```
private:
  std::int32_t nb_vertices;
  std::int32_t nb_reference_pixels;
  std::int32_t reference_width;
  std::int32_t reference_height;
  Eigen::SparseMatrix<float, Eigen::RowMajor> mat_splat;
  Eigen::SparseMatrix<float, Eigen::RowMajor> mat_slice;
  Eigen::SparseMatrix<float, Eigen::RowMajor> mat_blur;
  Eigen::MatrixXf mat_normalizer;
};
```

Chapter 4

bilateral_grid_simplified.cpp

```
#include "bilateral_grid_simplified.h"

bilateral_grid_simplified::bilateral_grid_simplified() :
  nb_vertices(0),
  nb_reference_pixels(0),
  reference_width(0),
  reference_height(0)
{

}

void bilateral_grid_simplified::init(const cv::Mat reference_bgr, const int sigma_spatial,
    const int sigma_luma, const int sigma_chroma)
{
  // let's work in yuv space
  cv::Mat reference_yuv;
  cv::cvtColor(reference_bgr, reference_yuv, CV_BGR2YUV);

  std::chrono::steady_clock::time_point begin_grid_construction = std::chrono::steady_clock::
      now();

  const int w = reference_yuv.cols;
  const int h = reference_yuv.rows;

  reference_width = w;
  reference_height = h;
  nb_reference_pixels = w * h;

  int max_coord[5];
  max_coord[0] = w / sigma_spatial;
  max_coord[1] = h / sigma_spatial;
  max_coord[2] = 255 / sigma_luma;
  max_coord[3] = 255 / sigma_chroma;
  max_coord[4] = 255 / sigma_chroma;

  // with this hash function we can convert each 5 dimensional coordinate to 1 unique number
  std::int64_t hash_vec[5];
  for (int i = 0; i < 5; ++i)
    hash_vec[i] = static_cast<std::int64_t>(std::pow(255, i));

  std::unordered_map<std::int64_t /* hash */, int /* vert id */> hashed_coords;
  hashed_coords.reserve(w*h);

  const unsigned char* pref = (const unsigned char*)reference_yuv.data;
  int vert_idx = 0;
  int pix_idx = 0;

  typedef Eigen::Triplet<float> T;
  std::vector<T> tripletList;
  tripletList.reserve(w * h);

  // loop through each pixel of the image
```

```
for (int y = 0; y < h; ++y)
{
  for (int x = 0; x < w; ++x)
  {
    std::int64_t coord[5];
    coord[0] = x / sigma_spatial;
    coord[1] = y / sigma_spatial;
    coord[2] = pref[0] / sigma_luma;
    coord[3] = pref[1] / sigma_chroma;
    coord[4] = pref[2] / sigma_chroma;

    // convert the coordinate to a hash value
    std::int64_t hash_coord = 0;
    for (int i = 0; i < 5; ++i)
      hash_coord += coord[i] * hash_vec[i];

    // pixels whom are alike will have the same hash value.
    // We only want to keep a unique list of hash values, therefore make sure we only insert
    // unique hash values.
    auto it = hashed_coords.find(hash_coord);
    if (it == hashed_coords.end())
    {
      hashed_coords.insert(std::pair<std::int64_t, int>(hash_coord, vert_idx));
      tripletList.push_back(T(vert_idx, pix_idx, 1.0f));
      ++vert_idx;
    }
    else
    {
      tripletList.push_back(T(it->second, pix_idx, 1.0f));

    }

    pref += 3; // skip 3 bytes (y u v)
    ++pix_idx;
  }
}

// construct our splat and splice matrices
mat_splat = Eigen::SparseMatrix<float, Eigen::RowMajor>(hashed_coords.size(), tripletList.
    size());
mat_splat.setFromTriplets(tripletList.begin(), tripletList.end());
mat_slice = mat_splat.transpose();

nb_vertices = static_cast<std::int32_t>(hashed_coords.size());
std::chrono::steady_clock::time_point end_grid_construction = std::chrono::steady_clock::now
    ();
std::cout << "grid construction:" << std::chrono::duration_cast<std::chrono::milliseconds>(
    end_grid_construction - begin_grid_construction).count() << "ms" << std::endl;


std::chrono::steady_clock::time_point begin_blur_construction = std::chrono::steady_clock::
    now();

// Blur matrices
Eigen::SparseMatrix<float, Eigen::RowMajor> mat_b_left(hashed_coords.size(), hashed_coords.
    size());
Eigen::SparseMatrix<float, Eigen::RowMajor> mat_b_right(hashed_coords.size(), hashed_coords.
    size());
mat_blur = Eigen::SparseMatrix<float, Eigen::RowMajor>(hashed_coords.size(), hashed_coords.
    size());
for (int i = 0; i < 5; ++i)
{
  std::int64_t offset_hash_coord = -1 * hash_vec[i];

  tripletList.clear();
  for (auto it = hashed_coords.begin(); it != hashed_coords.end(); ++it)
  {
    std::int64_t neighb_coord = it->first + offset_hash_coord;
    auto it_neighb = hashed_coords.find(neighb_coord);
```

```
      if (it_neighb != hashed_coords.end())
      {
        tripletList.push_back(T(it->second, it_neighb->second, 1.0f));
      }

    }
    mat_b_left.setZero();
    mat_b_left.setFromTriplets(tripletList.begin(), tripletList.end());


    offset_hash_coord = 1 * hash_vec[i];

    tripletList.clear();
    for (auto it = hashed_coords.begin(); it != hashed_coords.end(); ++it)
    {
      std::int64_t neighb_coord = it->first + offset_hash_coord;
      auto it_neighb = hashed_coords.find(neighb_coord);
      if (it_neighb != hashed_coords.end())
      {
        tripletList.push_back(T(it->second, it_neighb->second, 1.0f));
      }

    }
    mat_b_right.setZero();
    mat_b_right.setFromTriplets(tripletList.begin(), tripletList.end());

    mat_blur += mat_b_left;
    mat_blur += mat_b_right;
  }

  std::chrono::steady_clock::time_point end_blur_construction = std::chrono::steady_clock::now
      ();
  std::cout << "blur construction: " << std::chrono::duration_cast<std::chrono::milliseconds>(
      end_blur_construction - begin_blur_construction).count() << "ms" << std::endl;

  mat_slice.finalize();
  mat_splat.finalize();
  mat_blur.finalize();

  // normalization matrix (the splat matrix is not normalized)
  Eigen::MatrixXf mat_ones(w*h, 1);
  mat_ones.setOnes();
  Eigen::MatrixXf mat_ones_splatted = mat_splat * mat_ones;
  mat_normalizer = mat_slice * mat_ones_splatted;

}

Eigen::MatrixXf bilateral_grid_simplified::blur(Eigen::MatrixXf& in) const
{
  return  mat_blur * in + (in *(5.0f * 2.0f));
}

Eigen::SparseMatrix<float, Eigen::RowMajor> bilateral_grid_simplified::blur(const Eigen::
    SparseMatrix<float, Eigen::RowMajor>& in) const
{
  Eigen::SparseMatrix<float, Eigen::RowMajor> a = (in *(5.0f * 2.0f));
  Eigen::SparseMatrix<float, Eigen::RowMajor> b = mat_blur * in;
  return b + a;
}

cv::Mat bilateral_grid_simplified::filter(cv::Mat input_image)
{
  assert(input_image.type() == CV_32FC1);

  std::chrono::steady_clock::time_point start_blur = std::chrono::steady_clock::now();

  // splat & blur & slice
  Eigen::Map<Eigen::MatrixXf> eig_input_image(reinterpret_cast<float*>(input_image.data),
      input_image.cols * input_image.rows, 1);
```

```
  Eigen::MatrixXf mat_splatted = mat_splat * eig_input_image;
  Eigen::MatrixXf mat_splatted_and_blurred = blur(mat_splatted);
  Eigen::MatrixXf mat_blurred = mat_slice * mat_splatted_and_blurred;

  // normalize
  Eigen::MatrixXf mat_blurred_result = mat_blurred.cwiseQuotient(mat_normalizer);

  // convert to opencv
  cv::Mat cv_blurred_result(input_image.rows, input_image.cols, CV_32FC1);
  memcpy(cv_blurred_result.data, mat_blurred_result.data(), input_image.rows * input_image.
      cols * sizeof(float));

  std::chrono::steady_clock::time_point end_blur = std::chrono::steady_clock::now();
  std::cout << "blur: " << std::chrono::duration_cast<std::chrono::milliseconds>(end_blur -
      start_blur).count() << "ms" << std::endl;

  return cv_blurred_result;
}

cv::Mat bilateral_grid_simplified::splat_slice(cv::Mat input_image)
{
  assert(input_image.type() == CV_32FC1);

  // splat & slice
  Eigen::Map<Eigen::MatrixXf> eig_input_image(reinterpret_cast<float*>(input_image.data),
      input_image.cols * input_image.rows, 1);
  Eigen::MatrixXf mat_splatted = mat_splat * eig_input_image;
  Eigen::MatrixXf mat_sliced = mat_slice * mat_splatted;

  // normalize
  Eigen::MatrixXf result = mat_sliced.cwiseQuotient(mat_normalizer);

  cv::Mat cv_result(input_image.rows, input_image.cols, CV_32FC1);
  memcpy(cv_result.data, result.data(), input_image.rows * input_image.cols * 4);

  return cv_result;
}

//void serialize(std::string directory)
//{
//  //Serialize(directory + "\\mat_splat.dat", mat_splat);
//  //Serialize(directory + "\\mat_slice.dat", mat_slice);
//  //Serialize(directory + "\\mat_blur.dat", mat_blur);
//}
//
//void deserialize(std::string directory)
//{
//  /*  std::chrono::steady_clock::time_point begin_deserialize = std::chrono::steady_clock::
    now();
//
//  Deserialize(directory + "\\mat_splat.dat", mat_splat);
//  Deserialize(directory + "\\mat_slice.dat", mat_slice);
//  Deserialize(directory + "\\mat_blur.dat", mat_blur);
//  nb_vertices = static_cast<int>(mat_splat.outerSize());
//
//  std::chrono::steady_clock::time_point end_deserialize = std::chrono::steady_clock::now();
//  std::cout << "bilateral_grid::deserialize: " << std::chrono::duration_cast<std::chrono::
    milliseconds>(end_deserialize - begin_deserialize).count() << "ms" << std::endl;
//  */
//}
```

Chapter 5

stereo_matcher_birchfield_tomasi.h

```cpp
#pragma once

#include "bilateral_grid_simplified.h"

#include <Eigen/Sparse>

#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>

#include <string>
#include <chrono>

// Class for the stereo matching term
// Please paragraph '4. An Efficient Stereo Data Term' in the
// 'Fast Bilateral Space Stereo for Synthetic Defocus" paper.
class stereo_matcher_birchfield_tomasi
{
public:
  enum class block_filter_size
  {
    size_5x5,
    size_15x15,
    size_25x25
  };

  struct parameters
  {
    parameters():
      disparity_min(0),
      disparity_max(16),
      noise_epsilon(4),
      filter_size(block_filter_size::size_25x25)
    {

    }

    int disparity_min;
    int disparity_max;
    int noise_epsilon;
    block_filter_size filter_size;
  };
  parameters& get_parameters() { return current_parameters; }

  struct output
  {
    cv::Mat min_disp_image;
    cv::Mat max_disp_image;
    cv::Mat conf_disp_image;

    // Debugging images
    cv::Mat block_match_image;
    cv::Mat block_match_image1_x;
```

```
    cv::Mat block_match_image2_x;
    cv::Mat block_match_image1_y;
    cv::Mat block_match_image_final;
  };
  const output& get_output() { return current_output; }

public:
  stereo_matcher_birchfield_tomasi();
  ~stereo_matcher_birchfield_tomasi();

  // This function generates a min and max disparity image
  // - [in] stereo_images: uint8 grayscale stereo images
  // output: see get_output()
  void stereo_match(cv::Mat stereo_images[2]);

  // Generates a lookup table explained in the paper
  // - [in] grid: the bilateral grid
  // - [out] out_lookup: the output, a lookup table, meaning a cost value per disparity per
      vertex
  void generate_data_loss_table(const bilateral_grid_simplified& grid, std::vector<int>&
      out_lookup);

private:
  void block_filter_horz_21012(const cv::Mat& in, cv::Mat& out);
  void block_filter_horz_1050510(const cv::Mat& in, cv::Mat& out);

  void block_filter_vert_21012(const cv::Mat& in, cv::Mat& out);
  void block_filter_vert_1050510(const cv::Mat& in, cv::Mat& out);

  void block_filter_vert_505(const cv::Mat& in, cv::Mat& out);
  void block_filter_horz_505(const cv::Mat& in, cv::Mat& out);

  output current_output;
  parameters current_parameters;
};
```

Chapter 6

stereo_matcher_birchfield_tomasi.cpp

```cpp
#include "stereo_matcher_birchfield_tomasi.h"

stereo_matcher_birchfield_tomasi::stereo_matcher_birchfield_tomasi()
{

}

stereo_matcher_birchfield_tomasi::~stereo_matcher_birchfield_tomasi()
{

}

void stereo_matcher_birchfield_tomasi::stereo_match(cv::Mat stereo_images[2])
{
  assert(stereo_images[0].type() == CV_8UC1); // it should be grayscale
  assert(stereo_images[1].type() == CV_8UC1);

  const int disparity_min = current_parameters.disparity_min;
  const int disparity_max = current_parameters.disparity_max;
  const int noise_epsilon = current_parameters.noise_epsilon;

  std::chrono::steady_clock::time_point begin_stereo_match_construction = std::chrono::
      steady_clock::now();

  // a small blur
  cv::Mat stereo_filt_images[2];
  cv::boxFilter(stereo_images[0], stereo_filt_images[0], -1, cv::Size(2, 2));
  cv::boxFilter(stereo_images[1], stereo_filt_images[1], -1, cv::Size(2, 2));

  // min-max kernel
  cv::Mat stereo_images_upper[2];
  cv::Mat stereo_images_lower[2];
  cv::Mat minmax_kernel = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(2, 2));
  for (int i = 0; i < 2; ++i)
  {
    cv::erode(stereo_filt_images[i], stereo_images_lower[i], minmax_kernel);
    stereo_images_lower[i] -= noise_epsilon;
    cv::dilate(stereo_filt_images[i], stereo_images_upper[i], minmax_kernel);
    stereo_images_upper[i] += noise_epsilon;
  }

  const int width = stereo_images[0].cols;
  const int height = stereo_images[0].rows;

  // a lot of temporary images, makes debugging more easy :)
  cv::Mat& block_match_image = current_output.block_match_image;
  cv::Mat& block_match_image1_x = current_output.block_match_image1_x;
  cv::Mat& block_match_image2_x = current_output.block_match_image2_x;
  cv::Mat& block_match_image1_y = current_output.block_match_image1_y;
  cv::Mat& block_match_image_final = current_output.block_match_image_final;

  block_match_image.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);
```

```
block_match_image1_x.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);
block_match_image2_x.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);
block_match_image1_y.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);
block_match_image_final.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);

cv::Mat& min_disp_image = current_output.min_disp_image;
cv::Mat& max_disp_image = current_output.max_disp_image;
min_disp_image.create(stereo_images[0].rows, stereo_images[0].cols, CV_16SC1);
max_disp_image.create(stereo_images[0].rows, stereo_images[0].cols, CV_16SC1);
min_disp_image = cv::Scalar(std::numeric_limits<int16_t>::max());
max_disp_image = cv::Scalar(-std::numeric_limits<int16_t>::max());

for (int d = disparity_min; d <= disparity_max; d += 1)
{
  // check upper and lower bounds in order to see if we have a match
  // at this certain disparity
  if (d < 0) // negative disparity
  {
    for (int y = 0; y < height; ++y)
    {
      // we cannot calculate the cost function here, because we will go out of the image
      // so let's say we allow this disparity...
      int idx = y * width;
      for (int x = 0; x < -d; ++x, ++idx)
      {
        block_match_image.data[idx] = 1;
      }

      for (int x = -d; x < width; ++x, ++idx)
      {
        block_match_image.data[idx] =
          (stereo_images_upper[0].data[idx] >= stereo_images_lower[1].data[idx + d])
          &&
          (stereo_images_lower[0].data[idx] <= stereo_images_upper[1].data[idx + d]);
      }
    }
  }
  else // positive disparity
  {
    for (int y = 0; y < height; ++y)
    {
      int idx = y * width;
      for (int x = 0; x < width - d; ++x, ++idx)
      {
        block_match_image.data[idx] =
          (stereo_images_upper[0].data[idx] >= stereo_images_lower[1].data[idx + d])
          &&
          (stereo_images_lower[0].data[idx] <= stereo_images_upper[1].data[idx + d]);
      }

      // we cannot calculate the cost function here, because we will go out of the image
      // so let's say we allow this disparity...
      for (int x = width - d; x < width; ++x, ++idx)
      {
        block_match_image.data[idx] = 1;
      }
    }
  }

  // execute an erosion filter in order to supress the noise
  if (current_parameters.filter_size == block_filter_size::size_5x5)
  {
    block_filter_horz_21012(block_match_image, block_match_image1_x);
    block_filter_vert_21012(block_match_image1_x, block_match_image_final);
  }
  else if (current_parameters.filter_size == block_filter_size::size_15x15)
  {
    block_filter_horz_21012(block_match_image, block_match_image1_x);
    block_filter_horz_505(block_match_image1_x, block_match_image2_x);
```

```
      block_filter_vert_21012(block_match_image2_x, block_match_image1_y);
      block_filter_vert_505(block_match_image1_y, block_match_image_final);
    }
    else if (current_parameters.filter_size == block_filter_size::size_25x25)
    {
      block_filter_horz_21012(block_match_image, block_match_image1_x);
      block_filter_horz_1050510(block_match_image1_x, block_match_image2_x);

      block_filter_vert_21012(block_match_image2_x, block_match_image1_y);
      block_filter_vert_1050510(block_match_image1_y, block_match_image_final);
    }


    for (int i = 0; i < width * height; ++i)
    {
      if (block_match_image_final.data[i])
      {
        if (min_disp_image.at<int16_t>(i) == std::numeric_limits<int16_t>::max())
        {
          min_disp_image.at<int16_t>(i) = std::min(min_disp_image.at<int16_t>(i), (int16_t)d);
        }

        max_disp_image.at<int16_t>(i) = std::max(max_disp_image.at<int16_t>(i), (int16_t)d);
      }
    }


  }

  cv::Mat& conf_disp_image = current_output.conf_disp_image;
  conf_disp_image.create(stereo_images[0].rows, stereo_images[0].cols, CV_8UC1);
  for (int i = 0; i < width * height; ++i)
  {
    if (min_disp_image.at<int16_t>(i) == std::numeric_limits<int16_t>::max())
    {
      min_disp_image.at<int16_t>(i) = disparity_min;
      max_disp_image.at<int16_t>(i) = disparity_max;
      //  min_disp_image.at<int16_t>(i) = std::numeric_limits<int16_t>::max();
      //  max_disp_image.at<int16_t>(i) = std::numeric_limits<int16_t>::max();

      conf_disp_image.data[i] = 0;
    }
    else
    {
      conf_disp_image.data[i] = 255;
    }
  }

  std::chrono::steady_clock::time_point end_stereo_match_construction = std::chrono::
      steady_clock::now();
  std::cout << "stereo match: " << std::chrono::duration_cast<std::chrono::milliseconds>(
      end_stereo_match_construction - begin_stereo_match_construction).count() << "ms" << std::
      endl;
}

void stereo_matcher_birchfield_tomasi::generate_data_loss_table(const
    bilateral_grid_simplified& grid, std::vector<int>& lookup)
{
  const int nb_vertices = grid.get_nb_vertices();
  const int disparity_min = static_cast<int>(current_parameters.disparity_min);
  const int disparity_max = static_cast<int>(current_parameters.disparity_max);
  const int noise_epsilon = static_cast<int>(current_parameters.noise_epsilon);
  const int disparity_range = static_cast<int>((disparity_max - disparity_min) + 1);
  const cv::Mat& max_disp_image = current_output.max_disp_image;
  const cv::Mat& min_disp_image = current_output.min_disp_image;

  std::chrono::steady_clock::time_point begin_data_loss = std::chrono::steady_clock::now();
```

```
  lookup.clear();
  lookup.resize(nb_vertices * disparity_range, 0);

  for (int vertex_id = 0, vertex_id_end = nb_vertices; vertex_id < vertex_id_end; ++vertex_id)
  {
    int counter = 0;
    int* plookup = &lookup[vertex_id * disparity_range];
    for (Eigen::SparseMatrix<float, Eigen::RowMajor>::InnerIterator it(grid.get_splat_matrix()
        , vertex_id); it; ++it) // loop through pixels of that vertex
    {
      const int pixel_id = it.index();
      const int pixel_weight = static_cast<int>(it.value());

      int gj = 0;
      for (int j = max_disp_image.at<int16_t>(pixel_id) + 1 - disparity_min; j <
          disparity_range; ++j)
      {
        gj += pixel_weight;
        plookup[j] += gj;
      }

      gj = 0;
      for (int j = (int)min_disp_image.at<int16_t>(pixel_id) - 1 - disparity_min; j >= 0; --j)
      {
        gj += pixel_weight;
        plookup[j] += gj;
      }

    }

  }


  std::chrono::steady_clock::time_point end_data_loss = std::chrono::steady_clock::now();
  std::cout << "data loss: " << std::chrono::duration_cast<std::chrono::milliseconds>(
      end_data_loss - begin_data_loss).count() << "ms" << std::endl;

  // debug
  // cv::Mat debug_lookup_image(nb_vertices, matcher.disparty_max + 1, CV_32SC1);
  // memcpy(debug_lookup_image.data, lookup.data(), lookup.size() * sizeof(int));

}

void stereo_matcher_birchfield_tomasi::block_filter_horz_21012(const cv::Mat& in, cv::Mat& out
    )
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;

  int idx = 0;
  for (int y = 0; y < height; ++y)
  {
    out.data[idx] = in.data[idx] &&
      in.data[idx + 1] && in.data[idx + 2];
    ++idx;

    out.data[idx] = in.data[idx] && in.data[idx - 1] &&
      in.data[idx + 1] && in.data[idx + 2];
    ++idx;

    for (int x = 2; x < width - 2; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 1] && in.data[idx - 2] &&
        in.data[idx + 1] && in.data[idx + 2];
      ++idx;
    }

```

```
    out.data[idx] = in.data[idx] && in.data[idx - 1] &&
      in.data[idx - 2] && in.data[idx + 1];
    ++idx;

    out.data[idx] = in.data[idx] && in.data[idx - 1] && in.data[idx - 2];
    ++idx;
  }
}

void stereo_matcher_birchfield_tomasi::block_filter_horz_1050510(const cv::Mat& in, cv::Mat&
    out)
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;

  int idx = 0;
  for (int y = 0; y < height; ++y)
  {
    for (int x = 0; x < 5; ++x)
    {
      out.data[idx] = in.data[idx] &&
        in.data[idx + 5] && in.data[idx + 10];
      ++idx;
    }

    for (int x = 5; x < 10; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5] &&
        in.data[idx + 5] && in.data[idx + 10];
      ++idx;
    }

    for (int x = 10; x < width - 10; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5] && in.data[idx - 10] &&
        in.data[idx + 5] && in.data[idx + 10];
      ++idx;
    }

    for (int x = width - 10; x < width - 5; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5] && in.data[idx - 10] &&
        in.data[idx + 5];
      ++idx;
    }

    for (int x = width - 5; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5] && in.data[idx - 10];
      ++idx;
    }
  }
}

void stereo_matcher_birchfield_tomasi::block_filter_horz_505(const cv::Mat& in, cv::Mat& out)
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;

  int idx = 0;
  for (int y = 0; y < height; ++y)
  {
    for (int x = 0; x < 5; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx + 5];
```

```
      ++idx;
    }

    for (int x = 5; x < width - 5; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5] && in.data[idx + 5];
      ++idx;
    }

    for (int x = width - 5; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - 5];
      ++idx;
    }
  }
}

void stereo_matcher_birchfield_tomasi::block_filter_vert_21012(const cv::Mat& in, cv::Mat& out
    )
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;
  const int stride_1 = 1 * width;
  const int stride_2 = 2 * width;

  int idx = 0;
  for (int x = 0; x < width; ++x)
  {
    out.data[idx] = in.data[idx] &&
      in.data[idx + stride_1] && in.data[idx + stride_2];
    ++idx;
  }

  for (int x = 0; x < width; ++x)
  {
    out.data[idx] = in.data[idx] && in.data[idx - stride_1] &&
      in.data[idx + stride_1] && in.data[idx + stride_2];
    ++idx;
  }

  for (int y = 2; y < height - 2; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      int idx = x + y * width;

      out.data[idx] = in.data[idx] && in.data[idx - stride_1] && in.data[idx - stride_2] &&
        in.data[idx + stride_1] && in.data[idx + stride_2];
    }
  }

  for (int x = 0; x < width; ++x)
  {
    out.data[idx] = in.data[idx] && in.data[idx - stride_1] && in.data[idx - stride_2] &&
      in.data[idx + stride_1];
    ++idx;
  }

  for (int x = 0; x < width; ++x)
  {
    out.data[idx] = in.data[idx] && in.data[idx - stride_1] && in.data[idx - stride_2];
    ++idx;
  }

}

void stereo_matcher_birchfield_tomasi::stereo_matcher_birchfield_tomasi::
```

```
    block_filter_vert_1050510(const cv::Mat& in, cv::Mat& out)
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;
  const int stride_5 = 5 * width;
  const int stride_10 = 10 * width;

  int idx = 0;
  for (int y = 0; y < 5; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] &&
        in.data[idx + stride_5] && in.data[idx + stride_10];
      ++idx;
    }
  }

  for (int y = 5; y < 10; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - stride_5] &&
        in.data[idx + stride_5] && in.data[idx + stride_10];
      ++idx;
    }
  }

  for (int y = 10; y < height - 10; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      int idx = x + y * width;

      out.data[idx] = in.data[idx] && in.data[idx - stride_5] && in.data[idx - stride_10] &&
        in.data[idx + stride_5] && in.data[idx + stride_10];
    }
  }

  for (int y = height - 10; y < height - 5; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - stride_5] && in.data[idx - stride_10] &&
        in.data[idx + stride_5];
      ++idx;
    }
  }

  for (int y = height - 5; y < height; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - stride_5] && in.data[idx - stride_10];
      ++idx;
    }
  }
}

void stereo_matcher_birchfield_tomasi::stereo_matcher_birchfield_tomasi::block_filter_vert_505
    (const cv::Mat& in, cv::Mat& out)
{
  assert(in.type() == out.type());

  const int width = in.cols;
  const int height = in.rows;
  const int stride_5 = 5 * width;
```

```

  int idx = 0;
  for (int y = 0; y < 5; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] &&
        in.data[idx + stride_5];
      ++idx;
    }
  }

  for (int y = 5; y < height - 5; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      int idx = x + y * width;

      out.data[idx] = in.data[idx] && in.data[idx - stride_5] && in.data[idx + stride_5];
    }
  }

  for (int y = height - 5; y < height; ++y)
  {
    for (int x = 0; x < width; ++x)
    {
      out.data[idx] = in.data[idx] && in.data[idx - stride_5];
      ++idx;
    }
  }
}

//void serialize(std::string directory)
//{
//  cv::imwrite(directory + "\\min_disp_image.png", min_disp_image);
//  cv::imwrite(directory + "\\max_disp_image.png", max_disp_image);
//  cv::imwrite(directory + "\\conf_disp_image.png", conf_disp_image);
//}

//void deserialize(std::string directory)
//{
//  std::chrono::steady_clock::time_point begin_deserialize = std::chrono::steady_clock::now()
    ;

//  min_disp_image = cv::imread(directory + "\\min_disp_image.png", CV_LOAD_IMAGE_GRAYSCALE);
//  max_disp_image = cv::imread(directory + "\\max_disp_image.png", CV_LOAD_IMAGE_GRAYSCALE);
//  conf_disp_image = cv::imread(directory + "\\conf_disp_image.png", CV_LOAD_IMAGE_GRAYSCALE)
    ;

//  make_float_images();

//  std::chrono::steady_clock::time_point end_deserialize = std::chrono::steady_clock::now();
//  std::cout << "stereo_matcher::deserialize: " << std::chrono::duration_cast<std::chrono::
    milliseconds>(end_deserialize - begin_deserialize).count() << "ms" << std::endl;

//}
```

Chapter 7

fast_bilateral_solver.h

```cpp
#pragma once

#include <Eigen/Sparse>

#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>

#include <glog/logging.h>
#include <ceres/ceres.h>

#include <string>
#include <chrono>
#include <unordered_map>
#include <cassert>

#include "bilateral_grid_simplified.h"

// A nave implementation of the bilateral solver.
// There are many ways you can optimize this class.
class fast_bilateral_solver
{
  class fast_bilateral_problem : public ceres::FirstOrderFunction
  {
  public:
    fast_bilateral_problem(
      const Eigen::SparseMatrix<float, Eigen::RowMajor>& mat_C_CBC,
      const bilateral_grid_simplified& grid,
      const std::vector<int>& lookup,
      int disparty_range,
      float lambda,
      int keep_nb_of_intermediate_images);

    const Eigen::SparseMatrix<float, Eigen::RowMajor>& mat_C_CBC;
    const bilateral_grid_simplified& grid;
    const std::vector<int>& lookup;
    const int disparty_range;
    const float lambda;
    const int nb_vertices;
    const float border_cost_value = 500.0f; // used if the disparity is beyond the lookup table range
    const int keep_nb_of_intermediate_images;

    mutable double intermediate_prev_energy;
    mutable std::vector<Eigen::VectorXf> intermediate_results;

    virtual int NumParameters() const override { return nb_vertices; }

    bool Evaluate(const double* const parameters,
      double* cost,
      double* gradient) const override;
  };
```

```
public:
  fast_bilateral_solver();

  // Bistochastize function
  // - See page 4 of "Fast Bilateral Space Stereo for Synthetic Defocus - Supplemental
      Material"
  void bistochastize(const bilateral_grid_simplified& in_grid, Eigen::SparseMatrix<float,
      Eigen::RowMajor>& out_Dn, Eigen::SparseMatrix<float, Eigen::RowMajor>& out_Dm, const int
      nb_iterations = 16);

  // Runs the solver
  // - [in] initial_image_float: initial values for the solver
  // - [in] bilateral_grid_simplified:
  // - [in] data_cost_lookup: data cost lookup table, in this case the stereo disparity cost
      function
  // - [in] disparity_range: we work here from 0 ---> disparty_range
  // - [in] lambad: how much smoothing versus data term do you want?
  // - [in] max_num_iterations: number of iterations
  // - [in] keep_nb_of_intermediate_images: store the result of each iteration of the solver,
      one can set the maximum number of intermediate images it needs to store
  // returns the resolved image
  cv::Mat solve(cv::Mat initial_image_float, const bilateral_grid_simplified& grid, const std
      ::vector<int>& data_cost_lookup, const int disparity_range, const float lambda, const
      int max_num_iterations = 25, const int keep_nb_of_intermediate_images = 0);

  // the result of each iteration (see keep_nb_of_intermediate_images in the solve function)
  const std::vector<cv::Mat>& get_intermediate_result_images() { return intermediate_results;
      }
private:
  // Splats and image in pixel space in to bilateral space
  void splat_initial_image(const bilateral_grid_simplified& grid, const cv::Mat
      initial_image_float, double* parameters);

  std::vector<cv::Mat> intermediate_results;
};
```

Chapter 8

fast_bilateral_solver.cpp

```cpp
#include "fast_bilateral_solver.h"

fast_bilateral_solver::fast_bilateral_problem::fast_bilateral_problem(
  const Eigen::SparseMatrix<float, Eigen::RowMajor>& mat_C_CBC,
  const bilateral_grid_simplified& grid,
  const std::vector<int>& lookup,
  int disparty_range,
  float lambda,
  int keep_nb_of_intermediate_images) :
  mat_C_CBC(mat_C_CBC),
  grid(grid),
  lookup(lookup),
  disparty_range(disparty_range),
  lambda(lambda),
  nb_vertices(grid.get_nb_vertices()),
  keep_nb_of_intermediate_images(keep_nb_of_intermediate_images)
{
  intermediate_prev_energy = std::numeric_limits<double>::max();
}

bool fast_bilateral_solver::fast_bilateral_problem::Evaluate(const double* const parameters,
  double* cost,
  double* gradient) const
{
  // the solver works with doubles, but our functions uses float, not very efficient :/

  Eigen::VectorXf x_eig = Eigen::VectorXf(nb_vertices);
  Eigen::VectorXf grad_eig = Eigen::VectorXf(nb_vertices);

  // convert to an eigen vector (not very efficient in this way)
  for (int i = 0; i < nb_vertices; ++i)
  {
    x_eig(i) = static_cast<float>(parameters[i]);
  }

  /// Smoothing term!

  // smoothing energy loss
  float loss_smoothing_term = x_eig.transpose() * mat_C_CBC * x_eig;

  // smoothing gradient
  grad_eig = (mat_C_CBC * x_eig) * 2.0f;
  for (int i = 0; i < nb_vertices; ++i)
  {
    gradient[i] = grad_eig(i);
  }

  /// Data term
  const float fdisparty_max = (float)disparty_range;
  float loss_data_term = 0.f;
  for (int j = 0; j < nb_vertices; ++j)
  {
```

```
    const float vj = static_cast<float>(parameters[j]);

    float grad_data = 0.f;
    // is the disparity value in range of the lookup table?
    if (vj < 0.f)
    {
      // no
      grad_data = -(border_cost_value - 0.f);
      loss_data_term += 500.0f;
    }
    else if (vj >= fdisparty_max)
    {
      // no
      grad_data = -(0.f - border_cost_value);
      loss_data_term += 500.0f;
    }
    else
    {
      // yes!

      // a linear interpolation between lookup values
      const int vj_id = static_cast<int>(std::floor(vj));
      const int lookup_idx = j * disparity_range + vj_id;
      const float gj_floor = static_cast<float>(lookup[lookup_idx]);
      const float gj_ceil = static_cast<float>(lookup[lookup_idx + 1]);

      grad_data = -(gj_floor - gj_ceil);

      const float v1 = ((float)(vj_id + 1) - vj) * gj_floor;
      const float v2 = (vj - (float)vj_id) * gj_ceil;
      loss_data_term += v1 + v2;
    }

    gradient[j] += grad_data *lambda;
  }

  // final energy term (combine data and smoothing term)
  float loss_total = loss_smoothing_term + lambda * loss_data_term;
  cost[0] = loss_total;

  // keep result of each iteration if needed
  if ((loss_total < intermediate_prev_energy) && (intermediate_results.size() <
      keep_nb_of_intermediate_images))
  {
    intermediate_results.push_back(x_eig);
    intermediate_prev_energy = loss_total;
  }


  return true;
}

fast_bilateral_solver::fast_bilateral_solver()
{
  //google::InitGoogleLogging(nullptr);
}

void fast_bilateral_solver::bistochastize(const bilateral_grid_simplified& grid, Eigen::
    SparseMatrix<float, Eigen::RowMajor>& Dn, Eigen::SparseMatrix<float, Eigen::RowMajor>& Dm,
    const int nb_iterations)
{
  Eigen::MatrixXf mat_pixs(grid.get_splat_matrix().innerSize(), 1);
  mat_pixs.setOnes();
  Eigen::MatrixXf mat_m = grid.get_splat_matrix() * mat_pixs;

  Eigen::MatrixXf mat_n(grid.get_nb_vertices(), 1);
  mat_n.setOnes();

  // bistochastize method
```

```
  Eigen::MatrixXf mat_n_new;
  for (int i = 0; i < nb_iterations; ++i)
  {

    Eigen::MatrixXf ress = (mat_n.cwiseProduct(mat_m)).cwiseQuotient(grid.blur(mat_n));

    mat_n_new = ress.cwiseSqrt();

    Eigen::MatrixXf diff = mat_n_new - mat_n;
    float fdelta = diff.sum();
    std::cout << "bistochastize delta: " <<  fdelta << std::endl;
    mat_n = mat_n_new;
  }
  mat_m = mat_n.cwiseProduct(grid.blur(mat_n));

  // Convert result to diagonal matrices (could not find correct function in eigen :/)
  Dm = Eigen::SparseMatrix<float, Eigen::RowMajor>(grid.get_nb_vertices(), grid.
      get_nb_vertices());
  {
    typedef Eigen::Triplet<float> T;
    std::vector<T> tripletList;
    for (int i = 0; i < grid.get_nb_vertices(); ++i)
    {
      tripletList.push_back(T(i, i, mat_m(i, 0)));

    }
    Dm.setFromTriplets(tripletList.begin(), tripletList.end());
  }

  Dn = Eigen::SparseMatrix<float, Eigen::RowMajor>(grid.get_nb_vertices(), grid.
      get_nb_vertices());
  {
    typedef Eigen::Triplet<float> T;
    std::vector<T> tripletList;
    for (int i = 0; i < grid.get_nb_vertices(); ++i)
    {
      tripletList.push_back(T(i, i, mat_n.coeffRef(i, 0)));

    }
    Dn.setFromTriplets(tripletList.begin(), tripletList.end());
  }
}

void fast_bilateral_solver::splat_initial_image(const bilateral_grid_simplified& grid, const
    cv::Mat initial_image_float, double* parameters)
{
  assert(initial_image_float.type() == CV_32FC1);
  const int nb_pixels = initial_image_float.cols * initial_image_float.rows;
  Eigen::VectorXf in_vec(nb_pixels);
  const float *pff = reinterpret_cast<const float*>(initial_image_float.data);
  for (int i = 0; i < nb_pixels; ++i)
  {
    in_vec(i) = pff[i];
  }

  // splat input image
  Eigen::VectorXf splat_in_vec = grid.get_splat_matrix() * in_vec;

  // the grid is not normalized, so splat a bunch of 'ones' so we can normalize the result
  Eigen::MatrixXf normalization_vec(nb_pixels, 1);
  normalization_vec.setOnes();
  Eigen::MatrixXf normalization_weights = grid.get_splat_matrix() * normalization_vec;
  splat_in_vec = splat_in_vec.cwiseQuotient(normalization_weights);

  for (int i = 0; i < grid.get_nb_vertices(); ++i)
  {
    parameters[i] = splat_in_vec(i);
  }
}
```

```cpp

cv::Mat fast_bilateral_solver::solve(cv::Mat initial_image_float, const
    bilateral_grid_simplified& grid, const std::vector<int>& data_cost_lookup, const int
    disparity_range, const float lambda, const int max_num_iterations, const int
    keep_nb_of_intermediate_images)
{
  assert(initial_image_float.type() == CV_32FC1);

  const int nb_pixels = initial_image_float.cols * initial_image_float.rows;

  // fill in initial parameters/image
  double* parameters = new double[grid.get_nb_vertices()];
  splat_initial_image(grid, initial_image_float, parameters);

  // create smoothing matrix
  Eigen::SparseMatrix<float, Eigen::RowMajor> Dn, Dm;
  bistochastize(grid, Dn, Dm);
  const Eigen::SparseMatrix<float, Eigen::RowMajor> mat_CBC = Dn * grid.blur(Dn);
  const Eigen::SparseMatrix<float, Eigen::RowMajor> mat_C_CBC = Dm - mat_CBC;

  // setup solver & solve
  ceres::GradientProblemSolver::Options options;
  options.minimizer_progress_to_stdout = true;
  options.max_num_iterations = max_num_iterations;
  ceres::GradientProblemSolver::Summary summary;
  fast_bilateral_problem* pr = new fast_bilateral_problem(mat_C_CBC, grid, data_cost_lookup,
      disparity_range, lambda, keep_nb_of_intermediate_images);
  ceres::GradientProblem problem(pr);
  ceres::Solve(options, problem, parameters, &summary);

  std::cout << summary.FullReport() << "\n";

  // copy result in to an eigen vector
  Eigen::VectorXf eig_parameters(grid.get_nb_vertices());
  for (int i = 0; i < grid.get_nb_vertices(); ++i)
  {
    eig_parameters(i) = static_cast<float>(parameters[i]);
  }

  // slice the result!
  Eigen::MatrixXf eig_result = grid.get_slice_matrix() * eig_parameters;

  // to and opencv image
  cv::Mat result_image(initial_image_float.rows, initial_image_float.cols, CV_32FC1);
  memcpy(result_image.data, eig_result.data(), sizeof(float) * nb_pixels);

  // process intermediate results for debugging
  {
    intermediate_results.clear();
    intermediate_results.resize(pr->intermediate_results.size());
    Eigen::MatrixXf eig_result2;
    for (size_t i = 0; i < pr->intermediate_results.size(); ++i)
    {
      eig_result2 = grid.get_slice_matrix() * pr->intermediate_results[i];
      auto& img = intermediate_results[i];
      img.create(initial_image_float.rows, initial_image_float.cols, CV_32FC1);
      memcpy(img.data, eig_result2.data(), sizeof(float) * nb_pixels);
    }

  }


  delete[] parameters;

  return result_image;
}
```