Single Image Haze Removal Using Dark Channel Prior

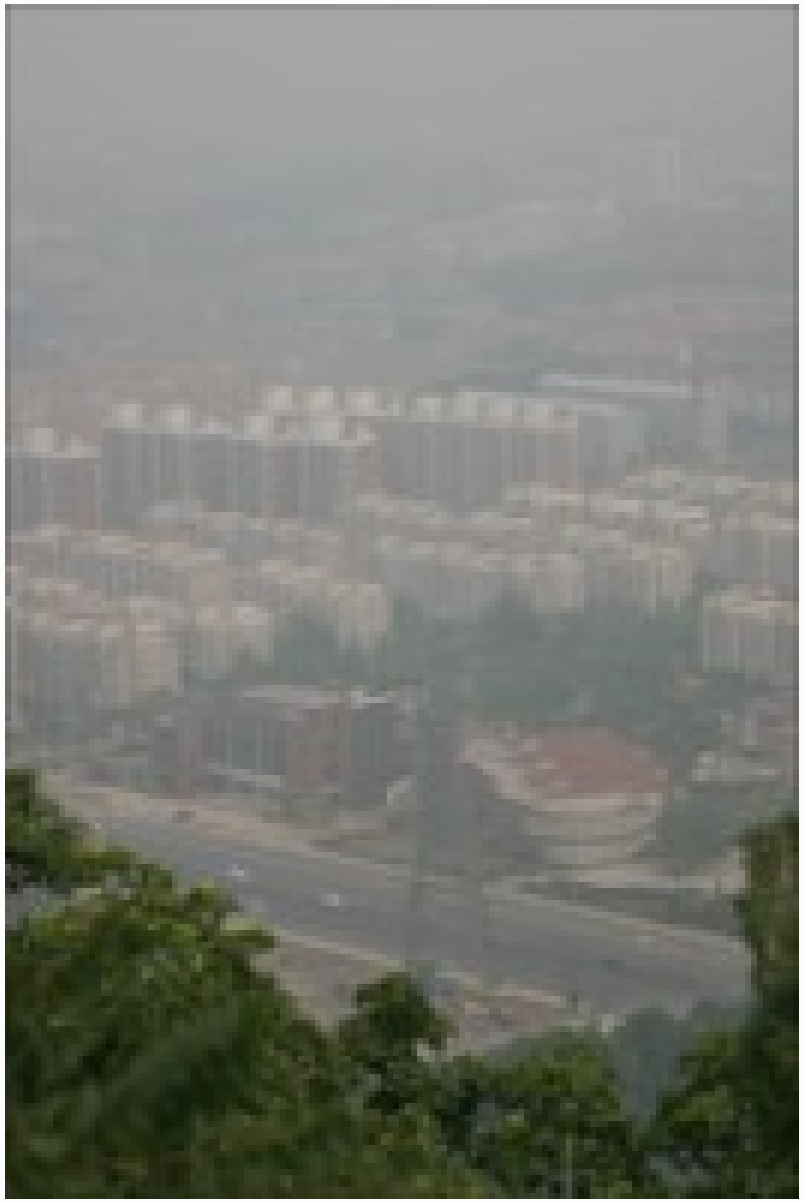
Implementação da técnica em C++

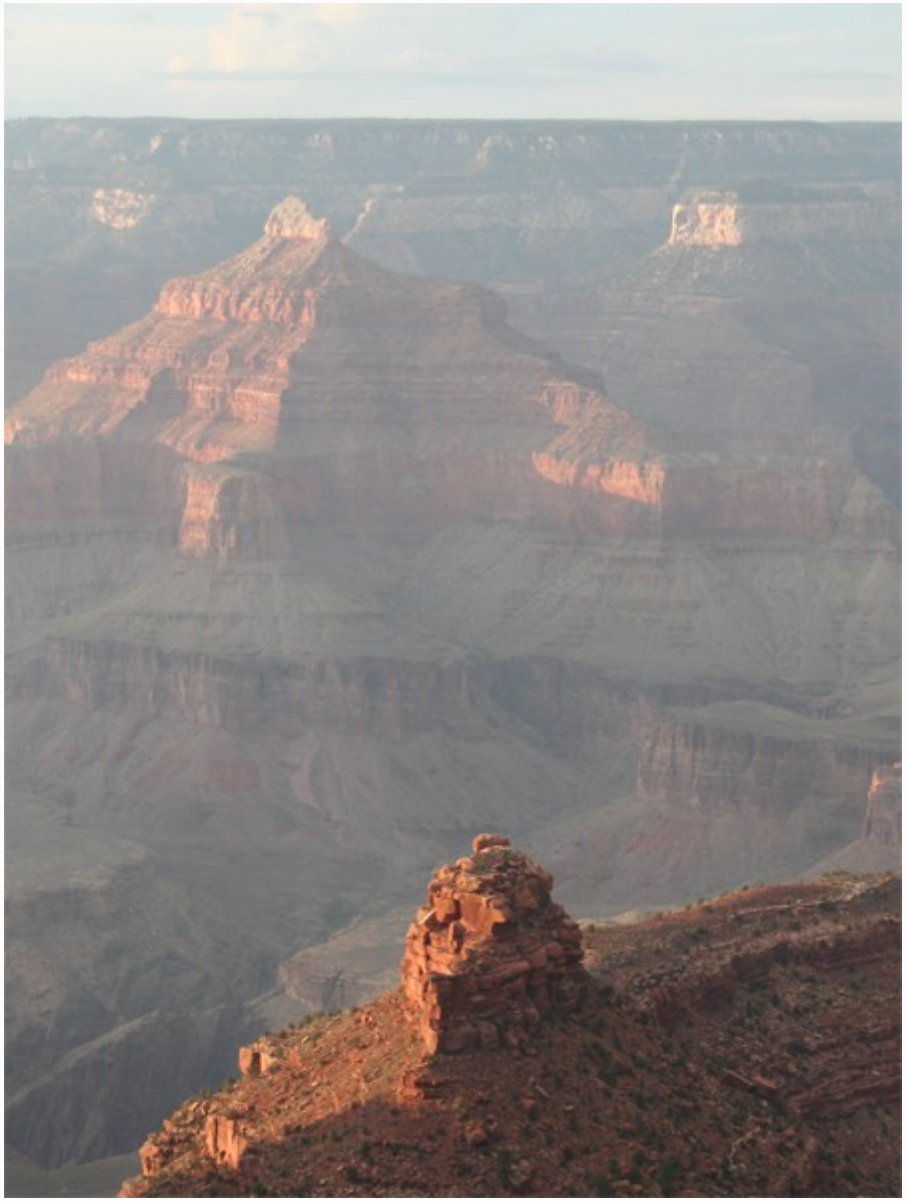
Vinicius Pavanelli Vianna
Eng. Eletricista com ênfase em Telecomunicações (EPUSP)

O que são imagens com “haze”?

- Haze pode ser traduzido como nevoeiro, bruma, são imagens esfumaçadas.
- O efeito é causado por partículas em suspensão no ar, que difrata a luz solar tornando a imagem esbranquiçada.

O que são imagens com “haze”?





Sobre o artigo

Single Image Haze Removal Using Dark Channel Prior

- Autores:
    - Kaiming He (Department of Information Engineering - The Chinese University of Hong Kong)
    - Jian Sun (Microsoft Research Asia)
    - Xiaoou Tang (Shenzhen Institute of Advanced Technology - Chinese Academy of Sciences)
- Best paper award - IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2009
- Publicado em IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: 33, Issue: 12, Dec. 2011)
- Utiliza uma técnica de “Image Matting” desenvolvida em A. Levin D. Lischinski and Y. Weiss. A Closed Form Solution to Natural Image Matting. (IEEE Trans. Pattern Analysis and Machine Intelligence, Feb 2008)

Sobre o artigo

Single Image Haze Removal Using Dark Channel Prior

- Possui muito material de consulta na página do autor principal:
  - http://kaiminghe.com/
  - http://kaiminghe.com/cvpr09/index.html

- Como a técnica inicialmente usada necessita de resolver um sistema linear muito grande os três autores desenvolveram uma outra técnica chamada “Guided Filtering”, que é muita utilizada em várias áreas e foi incluída no MATLAB 2014 e OpenCV 3.0

- Usando esta técnica de Guided Filtering eles conseguiram uma performance muito maior, segundo os dados dele em um i7 3.0GHz com 8GB de RAM e Windows 7:

  - A implementação deles demora 0,2s por cada Mp (megapixel).

  - Uma implementação utilizando uma abordagem “coarse-to-fine” consegue processar 32Mp em 0,5s

Canal Escuro (Dark Channel)

- O canal escuro é obtido utilizando o menor valor dos pixels dentro de uma janela.
  - 1. Criamos uma imagem onde cada pixel é o valor mínimo entre os canais R, G e B.
  - 2. A imagem do dark channel é o valor mínimo da imagem anterior dentro de uma janela.

$$J^{dark}(x) = \min_{y \in \Omega(x)} \left(\min_{c \in (r,g,b)} J^{c}(y) \right)$$

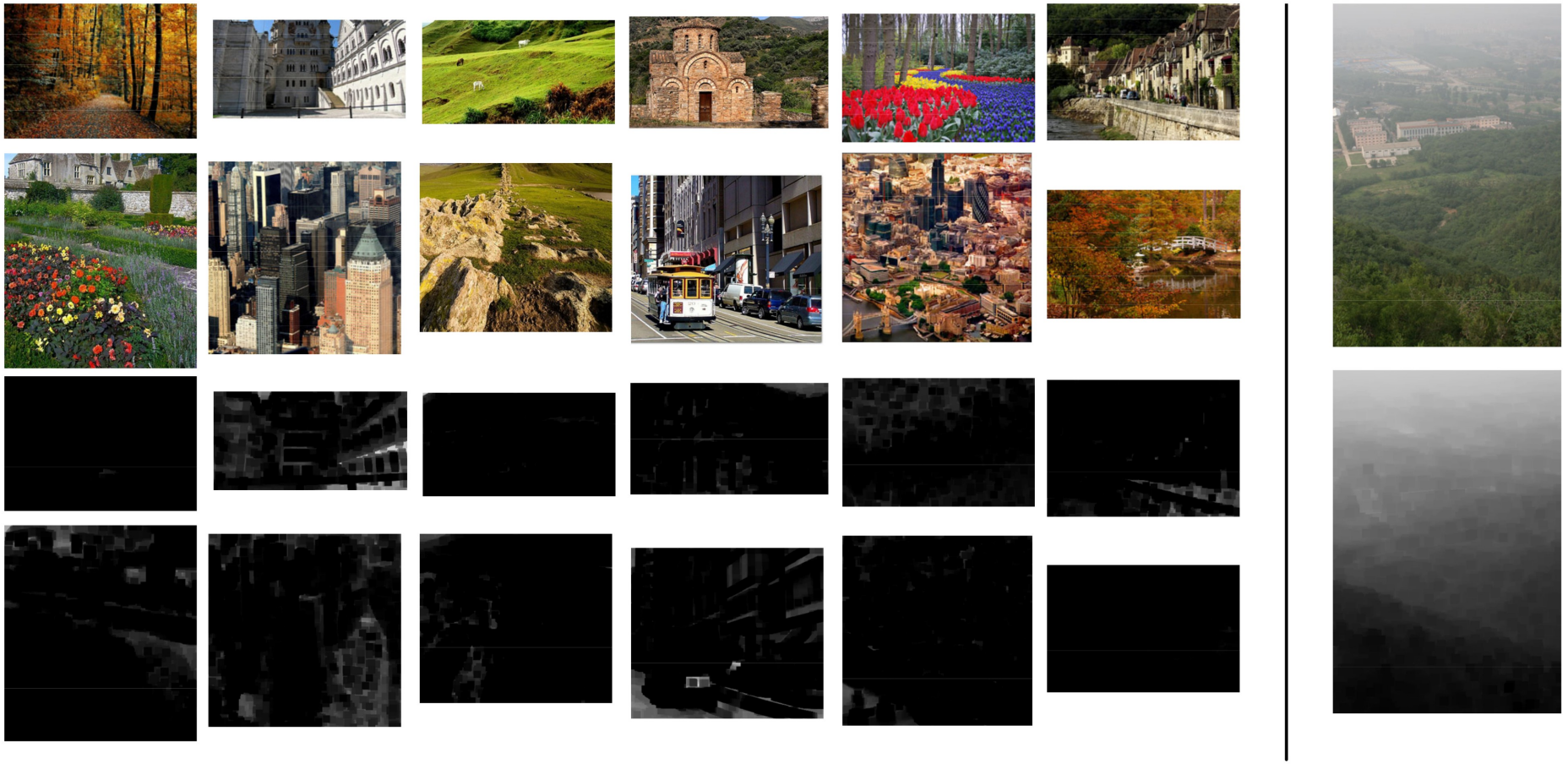
Canal Escuro (Dark Channel)

$$J$$

$$J^{min}(x) = \min_{c \in (r,g,b)} J^c(y)$$

$$J^{dark}(x) = \min_{y \in \Omega(x)} \left(\min_{c \in (r,g,b)} J^c(y) \right)$$

Canal Escuro (Dark Channel)

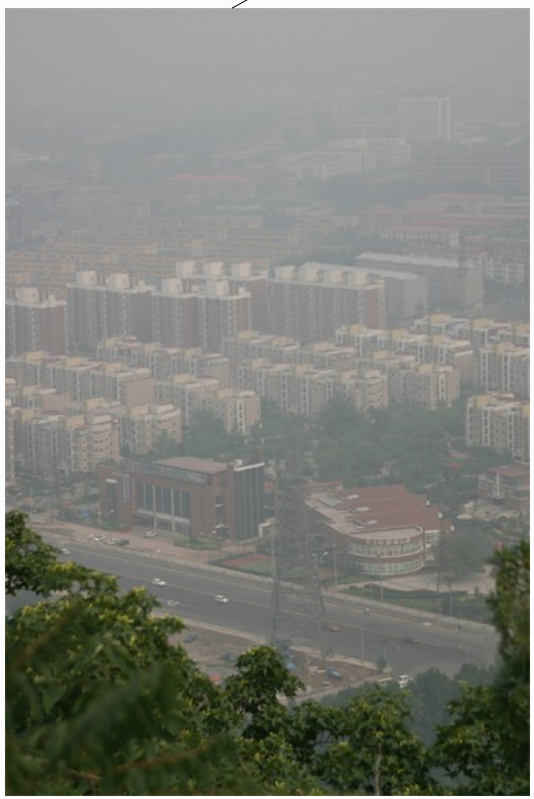


São escuros em imagens normais, e claros em imagens com névoas!

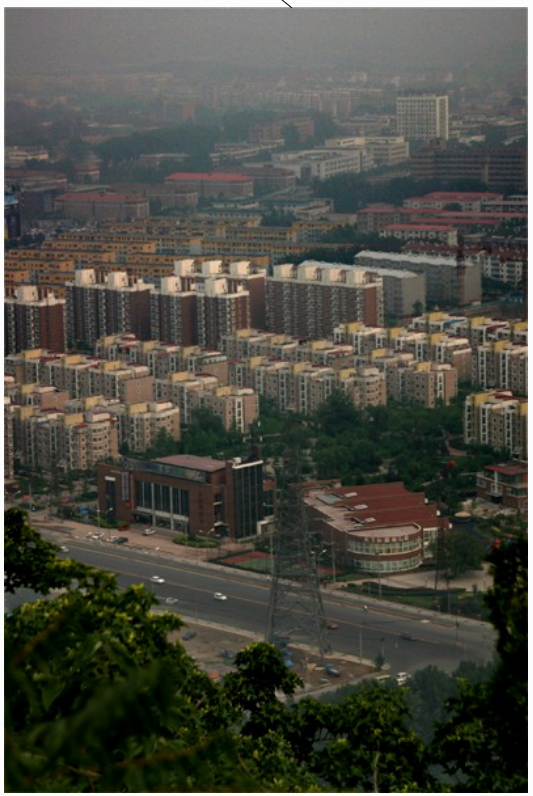
Modelamento Matemático

Luz atmosférica

$$I = J \cdot t + A \cdot (1 - t)$$



I



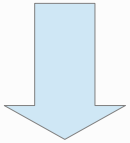
J

t

Como encontrar a transmissão ?

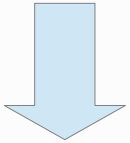
Modelamento matemático
(haze imaging model)

$$I = J \cdot t + A \cdot (1 - t)$$



Normalização

$$\frac{I}{A^c} = \frac{J}{A^c} \cdot t + A \cdot (1 - t)$$



$$\min_{\Omega}\left(\min_{c}\frac{I}{A^c}\right) = \min_{\Omega}\left(\min_{c}\left(\frac{J}{A^c}\right)\right) \cdot t + 1 - t$$

Como encontrar a transmissão ?

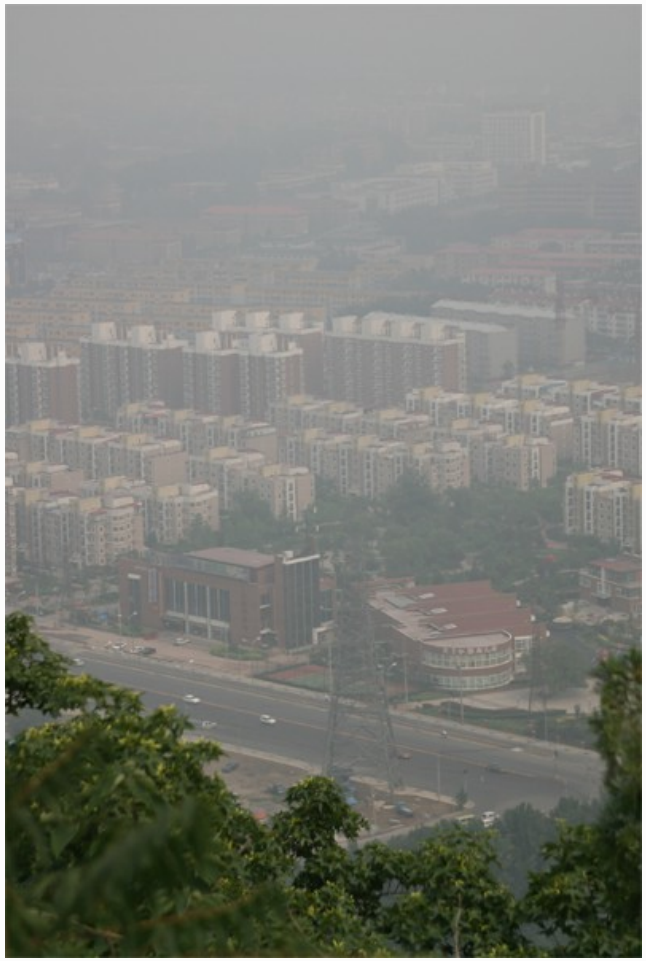
$$\min_{\Omega}\left(\min_{c}\frac{I}{A^{c}}\right)=\min_{\Omega}\left(\min_{c}\left(\frac{J}{A^{c}}\right)\right)\cdot t+1-t$$

$$J^{dark}=0\cdot t+1-t$$

$$t=1-J^{dark}$$

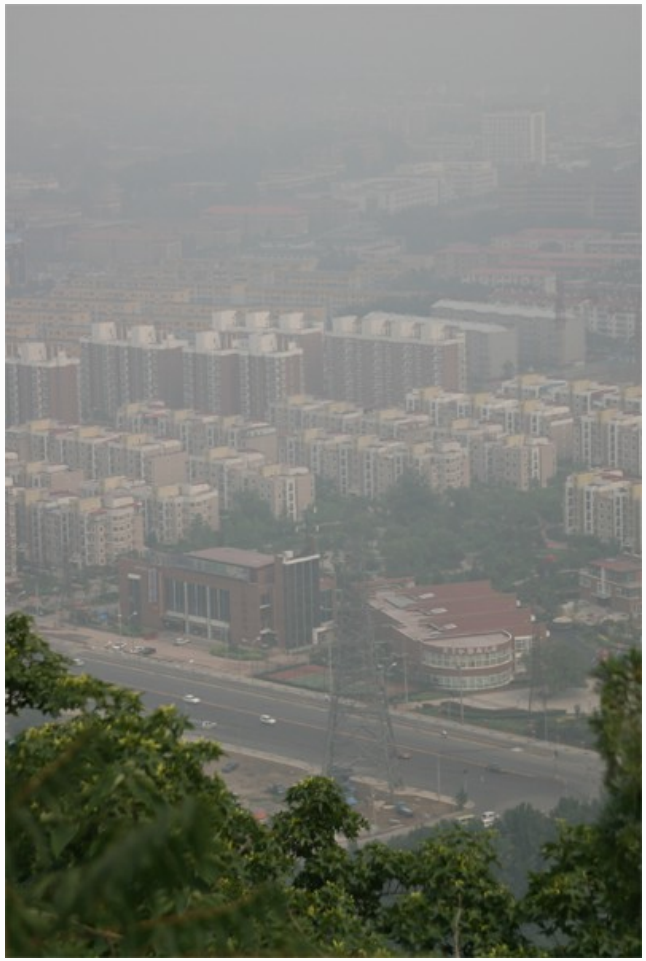
$$t=1-0{,}95\times J^{dark}$$

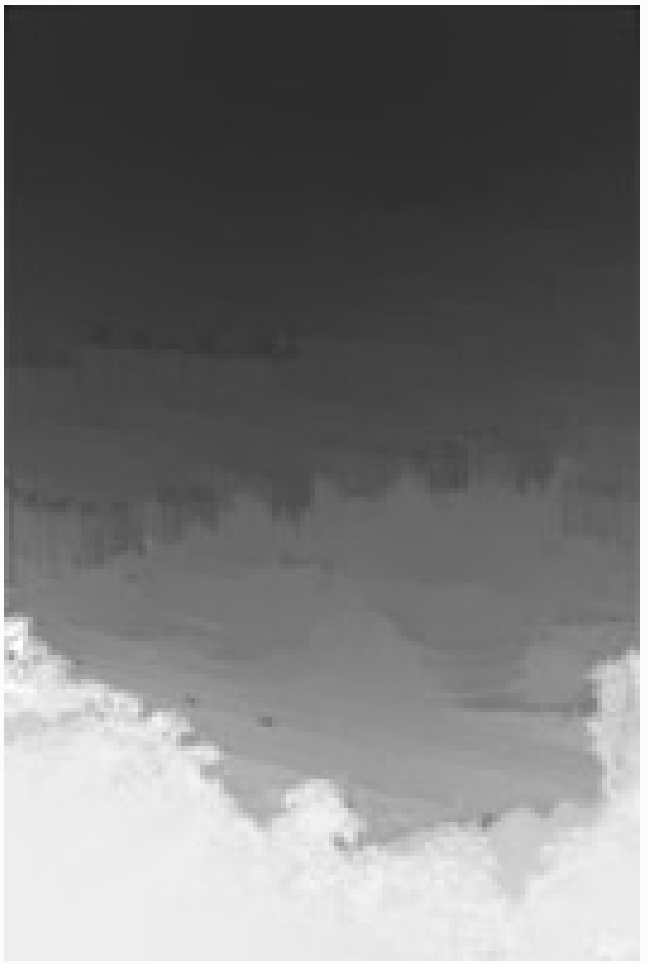
Como encontrar a transmissão ?



$$t = 1 - J^{dark}$$

Como encontrar a transmissão ?





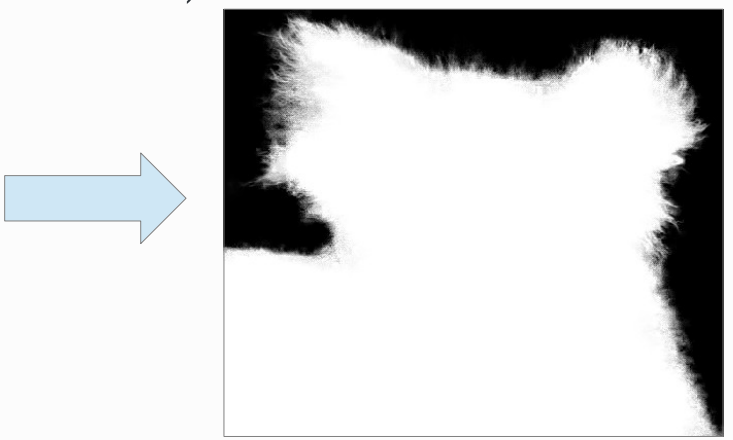
$$t = 1 - J^{dark}$$

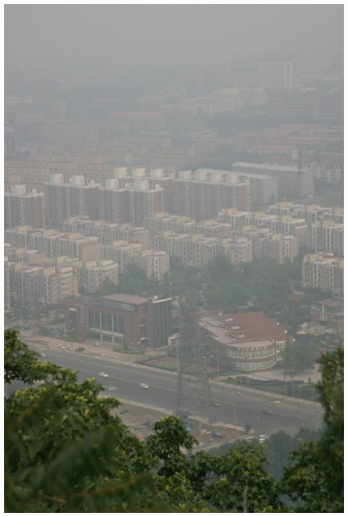
Image Matting Model

Haze imaging model $I = J \cdot t + A \cdot (1 - t)$

Matting model $I = F \cdot \alpha + B \cdot (1 - \alpha)$

+





+

Image Matting Model

- Temos que a estimativa da transmissão pode ser dada por: $E(t) = t^T L t + \lambda (t - \tilde{t})^T (t - \tilde{t})$

Cuja solução pode ser encontrada resolvendo o sistema abaixo:

$$(L + \lambda U_N) t = \lambda \tilde{t}$$

Aonde L é uma matriz que tem seus elementos definidos como:

$$L_{ij} = \sum_{k|(i,j)\in w_k} \left(\delta_{ij} - \frac{1}{|w_k|}\left(1 + (I_i - \mu_k)^T \left(\Sigma_k + \frac{\epsilon}{|w_k|} U_3\right)^{-1} (I_j - \mu_k)\right)\right)$$

δ_{ij} *é o delta de Kronecker ,* I_i *e* I_j *são as cores do pixel nos pontos i e j*

μ_k *é o valor médio e* Σ_k *é a covariância na janela w ,*

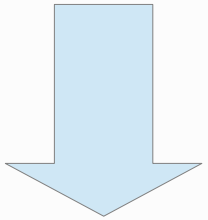
U_3 *é a matriz identidade ,* ϵ *é um parâmetro de regulação*

$|w_k|$ *é o número de pixels na janela*

Recuperando a radiância original

- Com o valor da transmitância e da luz atmosférica estimados podemos finalmente estimar a radiância original da cena:

$$I = J \cdot t + A \cdot (1 - t)$$



$$J(x) = \frac{I(x) - A}{max(t(x), t_0)} + A$$

Implementação em C++

- Uso das seguintes Bibliotecas:
  - ITK para as imagens
    - https://itk.org/
  - Armadillo para a Álgebra Linear
    - http://arma.sourceforge.net/
  - Boost para o controle de tarefas (Multithread)
    - http://www.boost.org/

Implementação em C++

Problemas encontrados:

- Alguns parâmetros não foram fornecidos no paper original, estes parâmetros foram ou procurados na bibliografia ou estimados.
- Algumas funções não são o que esperamos, como a covariância da matriz, que não é exatamente o resultado da função “cov” do MATLAB/Armadillo, mas sim uma fórmula matemática: $\Sigma = W \times W^T - \mu_k \times \mu_k^T$
- A matriz L é bem difícil de depurar, e muitos erros ocorreram por confusão com os vários índices utilizados, como i e j que são o índice na matriz e que gera um ix, iy, jx, jy que são os índices dos píxels na imagem, além de encontrar todas as janelas que recobrem esses dois pixels.
- A performance do código sem otimização é muito ruim, os testes iniciais foram feitos em imagens 100x100 para contornar este problema.
- Após o processo de remoção da névoa eles citam que passam a imagem por um filtro para melhorar a resposta, mas não explicam qual filtro usam.

Implementação em C++

Definição das seguintes funções

- Minima() - Cria uma imagem com o mínimo de cada canal
- dark_channel() - Cria uma imagem com o mínimo dentro de uma janela 15x15
- AchaA() - Encontra o valor da luz atmosférica a partir da imagem e do dark_channel
- CorrigeA() - Normaliza a imagem usando o valor a luz atmosférica
- CalculaT() - Faz a imagem de t a partir do dark_channel (t = 1 – dark)
- Matting2() - Suaviza a imagem de t usando soft matting.
- Guided_filtering() - Suaviza a imagem de t usando guided filtering.
- tira_haze() - Tira a névoa da imagem original usando o mapa suavizado

Implementação em C++

```
void minima(typename ImageType::Pointer image_in,
typename ImageGrayType::Pointer image_min) {

  /* Criar imagem com minimo dentre os canais RGB */
  for (unsigned int i = 0; i < size[0]; i++) {
      for (unsigned int j = 0; j < size[1]; j++) {
          auto pixel = image_in->GetPixel({i,j});
          double min = std::min(pixel[0],
                                std::min(pixel[1], pixel[2]));
          image_min->SetPixel({i, j}, min);
      }
  }
}
```

$$J^{min}(x) = \min_{c \in (r,g,b)} J^{c}(y)$$

Implementação em C++

```
void dark_channel(typename ImageGrayType::Pointer image_min, typename ImageGrayType::Pointer
image_dark) {
  const int Patch_window = 7;  // floor(15/2);
    arma::fmat  I(size[0], size[1]);
    for (int i=0; i < size[0]; i++) {
        for (int j=0; j < size[1]; j++) {
            I(i,j) = image_min->GetPixel({i,j});
        }
    }
  I.each_row([Patch_window](arma::Row<float> &a) {
          arma::Row<float> tmp(a.n_elem);
          for (int i=0; i<a.n_elem;i++) {
            int b = std::max(0,i-Patch_window);
            int e = std::min((int) a.n_elem-1, i+Patch_window-1);
            tmp(i) = a(arma::span(b,e)).min();
          }
          a = tmp;
          });
  I.each_col([Patch_window](arma::Col<float> &a) {
          arma::Row<float> tmp(a.n_elem);
          for (int i=0; i<a.n_elem;i++) {
            int b = std::max(0,i-Patch_window);
            int e = std::min((int) a.n_elem-1, i+Patch_window-1);
            tmp(i) = a(arma::span(b,e)).min();
          }
          a = tmp;
          });
  for (int i = 0; i < size[0]; i++) {
      for (int j = 0; j < size[1]; j++) {
          image_dark->SetPixel({i,j}, I(i,j));
      }
  }
}
```

$$J^{dark}(x) = \min_{y \in \Omega(x)} \left(\min_{c \in (r,g,b)} J^{c}(y) \right)$$

Implementação em C++

```
std::vector<double> achaA(typename ImageGrayType::Pointer image_dark, typename
ImageType::Pointer image_in) {
  unsigned int total_pixels = size[0]*size[1];
  int porcento01 = ceil((double) total_pixels * 0.001);
  float pixel_max = 1.0f;
  std::vector<double> pixel_A({0, 0, 0});

  std::vector<std::pair<double, unsigned int>> pixels;
  pixels.reserve(total_pixels);
  int c = 0;
  PixelComponent *ptr = image_dark->GetBufferPointer();
  while (c < total_pixels) {
      pixels[c] = { *(ptr+c), c };
      c++;
  }
  std::sort(pixels.begin(), pixels.end(), [] (std::pair<double, unsigned int> const
&a, std::pair<double, unsigned int> const &b) { return a.first < b.first; });
  PixelType *ptr2 = image_in->GetBufferPointer();
  for (int i = 0; i < porcento01; i++) {
      PixelType *pixel = ptr2+pixels[i].second;
                  pixel_A[0] = std::max(pixel_A[0], (double) (*pixel)[0]);
                  pixel_A[1] = std::max(pixel_A[1], (double) (*pixel)[1]);
                  pixel_A[2] = std::max(pixel_A[2], (double) (*pixel)[2]);

  }
  return pixel_A;
}
```

Implementação em C++

```
void corrigeA(ImageType::Pointer image_in,
              ImageType::Pointer image_ac,
              std::vector<double> A) {
  for (unsigned int i = 0; i < size[0]; i++) {
      for (unsigned int j = 0; j < size[1]; j++) {
          auto pixel = image_in->GetPixel({i,j});
          std::vector<double> pixel2;
          pixel2.resize(3);
          pixel2[0] = pixel[0] / A[0];
          pixel2[1] = pixel[1] / A[1];
          pixel2[2] = pixel[2] / A[2];
          pixel[0] = std::min(1.0, std::max(0.0, pixel2[0]));
          pixel[1] = std::min(1.0, std::max(0.0, pixel2[1]));
          pixel[2] = std::min(1.0, std::max(0.0, pixel2[2]));
          image_ac->SetPixel({i,j}, pixel);
      }
  }
}
```

Implementação em C++

```
void calculaT(typename ImageGrayType::Pointer image_dark,
              typename ImageGrayType::Pointer image_t) {

  for (unsigned int i = 0; i < size[0]; i++) {
      for (unsigned int j = 0; j < size[1]; j++) {
          image_t->SetPixel({i,j},
                    1.0 - 0.95*image_dark->GetPixel({i,j}));
      }
  }

}
```

$$t = 1 - 0{,}95 \times J^{dark}$$

Implementação em C++

```
void matting2 (ImageType::Pointer image_in, ImageGrayType::Pointer image_tchapeu, ImageGrayType::Pointer image_t) {

    long int total_pixels = size[0]*size[1];
    std::cout << "matting2: " << size[0] << "x" << size[1] << std::endl;
    arma::sp_fmat L(total_pixels, total_pixels);

    std::vector<Cache> cache;
    cache.reserve(total_pixels);


    std::cout << "Fazendo cache..." << std::flush;
    {
        int w = 0;
        for (int y=0; y<size[1]; y++) {
            for (int x=0; x<size[0]; x++) {
                Cache tmp;
                arma::fmat W(carregaJanela(image_in, x, y));
                tmp.mean = arma::mean(W).t();
                arma::fmat Wcov(W.t()*W/wk - tmp.mean*tmp.mean.t());
                tmp.cov_inv = arma::inv(Wcov + (float) (epsilon/wk)*arma::eye<arma::fmat>(3,3));
                cache[w++] = tmp;
            }
        }
    }
    std::cout << " feito" << std::endl;

    boost::asio::io_service _io;
    std::unique_ptr<boost::asio::io_service::work> _work(
                new boost::asio::io_service::work(_io));
    boost::thread_group _threads;
    for (int i=0; i < THREADS; i++) {
        _threads.add_thread(new boost::thread(boost::bind(&boost::asio::io_service::run, &_io)));
    }
    std::mutex mtx;
    posts=0;

...
```

Implementação em C++

```
// matting2 continuacao


    std::cout << "Fazendo L..." << std::endl;
    for (int iy=0; iy < size[1]; iy++) {
        if (iy % 1 == 0) std::cout << "   -> " << iy << " de " << size[1] << std::endl;
        for (int ix=0; ix < size[0]; ix++) {
            _io.post(boost::bind(&thread_L, image_in, ix, iy, total_pixels, boost::ref(L),
                                 boost::ref(mtx), boost::ref(cache)));
            posts++;
        }
    }

    _work.reset();
    _threads.join_all();
    std::vector<Cache>().swap(cache);
    { // Faz diagonal da matriz L usando a soma de cada linha
        auto Lsum = arma::sum(L,1);
        L -= diagmat(Lsum);
        L.diag() += lambda;
    }

    arma::fmat Mtchapeu(total_pixels,1);
    {
        auto *ptr = image_tchapeu->GetBufferPointer();
        unsigned int c = 0;
        double tmin=1.0, tmax=0.0;
        while (c < total_pixels) {
            if (*(ptr+c) > tmax) tmax = *(ptr+c);
            if (*(ptr+c) < tmin) tmin = *(ptr+c);
            Mtchapeu(c,0) = *(ptr+c)*lambda;
            c++;
        }
    }
```

$$\left(L+\lambda U_N\right)t=\lambda\tilde{t}$$

Implementação em C++

```
// matting2 continuacao


    std::cout << "Fazendo Mt..." << std::endl;
    arma::fmat Mt;
    arma::superlu_opts opts;
    opts.symmetric=true;
    opts.refine=arma::superlu_opts::REF_NONE;
    arma::spsolve(Mt, L, Mtchapeu, "superlu", opts);
    std::cout << "Fazendo t..." << std::endl;
    {
        double tmin=Mt(0,0), tmax=Mt(0,0);
        auto *ptr = image_t->GetBufferPointer();
        unsigned int c = 0;
        while (c < total_pixels) {
            *(ptr+c) = Mt(c,0);
            c++;
        }
    }
}
```

$$\left(L+\lambda U_N\right)t=\lambda\tilde{t}$$

Implementação em C++

```
void thread_L(ImageType::Pointer image, int ix, int iy, int total_pixels, arma::sp_fmat &L, std::mutex &mtx,
std::vector<Cache> const &cache) {
    std::unordered_map<int, float> Ltmp;

    int i = ConverteXY2I(ix, iy);

    for (int jx = ix+1; jx < std::min(ix+3,(int) size[0]); jx++) {
            double tmp = matting_L(image, ix, iy, jx, iy, cache);
            if (!std::isnan(tmp)) {
                Ltmp[ConverteXY2I(jx,iy)] = tmp;
            }
    }

    if (iy+1 < size[1])
    for (int jy = iy+1; jy < std::min(iy+3,(int) size[1]); jy++) {
        for (int jx = std::max(ix-2,0); jx < std::min(ix+3,(int) size[0]); jx++) {
                double tmp = matting_L(image, ix, iy, jx, jy, cache);
                if (!std::isnan(tmp)) {
                    Ltmp[ConverteXY2I(jx,jy)] = tmp;
                }
        }
    }
    {
        std::lock_guard<std::mutex> guard(mtx);
        for (auto &tmp : Ltmp) {
            L(i,tmp.first) = L(tmp.first,i) = tmp.second;
        }
    }
    posts--;
}
```

$$L_{ij} = \sum_{k|(i,j)\in w_k} \left(\delta_{ij} - \frac{1}{|w_k|}\left(1 + (I_i - \mu_k)^T \left(\Sigma_k + \frac{\epsilon}{|w_k|} U_3\right)^{-1} (I_j - \mu_k)\right)\right)$$

Implementação em C++

```
double matting_L (ImageType::Pointer image,
                  int ix, int iy, int jx, int jy,
                  std::vector<Cache> const &cache) {

    std::vector<std::pair<int, int>> Ws(achaJanelas (ix, iy, jx, jy));

    if (Ws.size() == 0) return std::numeric_limits<double>::quiet_NaN();

    double soma = 0.0;

    for (auto &w : Ws) {
        soma += calculaL (image, ix, iy, jx, jy, w, cache);
    }

    return soma;

}
```

$$L_{ij} = \sum_{k|(i,j)\in w_k} \left(\delta_{ij} - \frac{1}{|w_k|}\left(1 + \left(I_i - \mu_k\right)^T \left(\Sigma_k + \frac{\epsilon}{|w_k|} U_3\right)^{-1} \left(I_j - \mu_k\right)\right)\right)$$

Implementação em C++

```
std::vector<std::pair<int, int>>
           achaJanelas (int ix, int iy, int jx, int jy) {
    std::vector<std::pair<int, int>> ws;
    if (abs(ix-jx) < 3 && abs(iy-jy) < 3) {
        int xmin = std::max(std::min(ix,jx)-2,0),
            xmax = std::min(std::max(ix,jx)+2,largura-1),
            ymin = std::max(std::min(iy,jy)-2,0),
            ymax = std::min(std::max(iy,jy)+2,altura-1);

        for (int x = xmin; x <= xmax; x++) {
            for (int y = ymin; y <= ymax; y++) {
                if (    abs(x-ix) < 2 &&
                        abs(x-jx) < 2 &&
                        abs(y-iy) < 2 &&
                        abs(y-jy) < 2) {
                    ws.push_back({x,y});
                }
            }
        }
    }
    return ws;
}
```

Implementação em C++

```
double calculaL (ImageType::Pointer image, int ix, int iy, int jx, int jy,
std::pair<int, int> wp, std::vector<Cache> const &cache) {

    using namespace std;
    int w = ConverteXY2I(wp.first, wp.second);
    double resultado = (ix==jx && iy == jy) ? 1.0 : 0.0;
    auto pixelI = image->GetPixel({ix, iy});
    auto pixelJ = image->GetPixel({jx, jy});
    arma::mat Ii, Ij;

    Ii << pixelI[0] << arma::endr << pixelI[1] << arma::endr << pixelI[2];
    Ij << pixelJ[0] << arma::endr << pixelJ[1] << arma::endr << pixelJ[2];

    arma::mat tmp =
              (Ii - cache[w].mean).t() * cache[w].cov_inv * (Ij - cache[w].mean);

    if (DEBUG) tmp.print(cout, "tmp");
    resultado -= (1+tmp(0,0))/wk;
    return resultado;
}
```

$$L_{ij} = \sum_{k|(i,j)\in w_k} \left(\delta_{ij} - \frac{1}{|w_k|}\left(1 + \left(I_i - \mu_k\right)^T \left(\Sigma_k + \frac{\epsilon}{|w_k|} U_3\right)^{-1} \left(I_j - \mu_k\right)\right)\right)$$

Implementação em C++

```
void guided_filtering (ImageType::Pointer image_in, ImageGrayType::Pointer image_tchapeu, ImageGrayType::Pointer
image_t) {
    std::cout << "Starting Guided Filtering" << std::endl;
    arma::fmat   I(altura, largura);
    arma::fmat   P(altura, largura);
      for (int i=0; i < altura; i++) {
            for (int j=0; j < largura; j++) {
                  I(i,j) = image_in->GetPixel({j,i}).GetLuminance();
                  P(i,j) = image_tchapeu->GetPixel({j,i});
            }
      }
    arma::fmat meanI(fmean(I));
    arma::fmat meanP = fmean(P);
    arma::fmat corrI = fmean(I % I);
    arma::fmat corrIP = fmean (I % P);
    arma::fmat varI = corrI - (meanI % meanI);
    arma::fmat covIP = corrIP - (meanI % meanP);
    arma::fmat a = covIP / (varI + epsilonG);
    arma::fmat b = meanP - a % meanI;
    arma::fmat meanA = fmean(a);
    arma::fmat meanB = fmean(b);
    arma::fmat q = (meanA % I) + meanB;

      for (int i=0; i < altura; i++) {
            for (int j=0; j < largura; j++) {
             image_t->SetPixel({j,i}, q(i,j));
            }
      }
}
```

Algorithm 1. Guided Filter.

Input: filtering input image p, guidance image I, radius r, regularization ϵ

Output: filtering output q.

1: $\text{mean}_I = f_{\text{mean}}(I)$
$\text{mean}_p = f_{\text{mean}}(p)$
$\text{corr}_I = f_{\text{mean}}(I. * I)$
$\text{corr}_{Ip} = f_{\text{mean}}(I. * p)$
2: $\text{var}_I = \text{corr}_I - \text{mean}_I. * \text{mean}_I$
$\text{cov}_{Ip} = \text{corr}_{Ip} - \text{mean}_I. * \text{mean}_p$
3: $a = \text{cov}_{Ip}./(\text{var}_I + \epsilon)$
$b = \text{mean}_p - a. * \text{mean}_I$
4: $\text{mean}_a = f_{\text{mean}}(a)$
$\text{mean}_b = f_{\text{mean}}(b)$
5: $q = \text{mean}_a. * I + \text{mean}_b$

/* f_{mean} is a mean filter with a wide variety of O(N) time methods. */

Implementação em C++

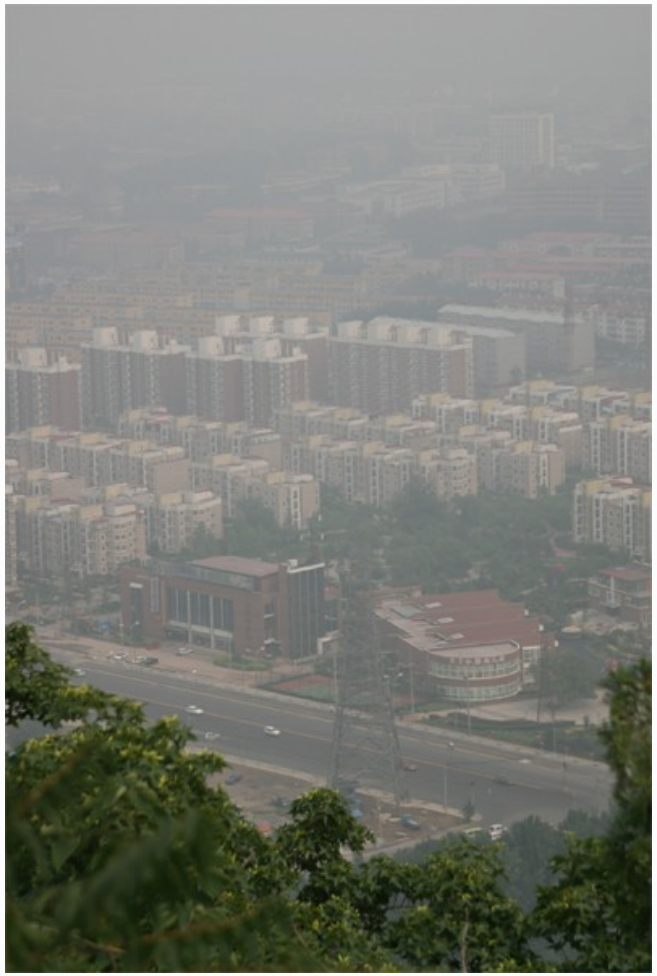
```
void sumRow(arma::fmat const &I_in, int r, arma::fmat &retorno, int i, std::mutex &mtx) {
    arma::Row<float> a = I_in.row(i), tmp(a.n_elem);
    for (int i=0; i < a.n_elem; i++) {
        int b = std::max(0,i-r/2), e = std::min(i+r/2,(int) a.n_elem-1);
        tmp(i) = (arma::mean(a(arma::span(b,e))));
    }
    {   std::lock_guard<std::mutex> lock(mtx);
        retorno.row(i) = tmp;                    }
}
void sumCol(arma::fmat const &I_in, int r, arma::fmat &retorno, int i, std::mutex &mtx) {
    arma::Col<float> a = I_in.col(i), tmp(a.n_elem);
    for (int i=0; i < a.n_elem; i++) {
        int b = std::max(0,i-r/2), e = std::min(i+r/2,(int) a.n_elem-1);
        tmp(i) = (arma::mean(a(arma::span(b,e))));
    }
    {   std::lock_guard<std::mutex> lock(mtx);
        retorno.col(i) = tmp;                     }
}

arma::fmat fmean(arma::fmat const &I_in) {
    int r = 200;
    arma::fmat retorno(arma::size(I_in)), retorno2(arma::size(I_in));
    std::vector<std::future<void>> jobs;
    std::mutex mtx;
    for (int i=0; i<I_in.n_rows; i++) {
        jobs.push_back(std::async(std::launch::async, sumRow, std::cref(I_in), r, std::ref(retorno), i,
std::ref(mtx)));
    }
    for (auto &i : jobs) i.wait();
    std::vector<std::future<void>>().swap(jobs);

    for (int i=0; i<I_in.n_cols; i++) {
        jobs.push_back(std::async(std::launch::async, sumCol, std::cref(retorno), r, std::ref(retorno2), i,
std::ref(mtx)));
    }
    for (auto &i : jobs) i.wait();
    return retorno2;
}
```

Implementação em C++

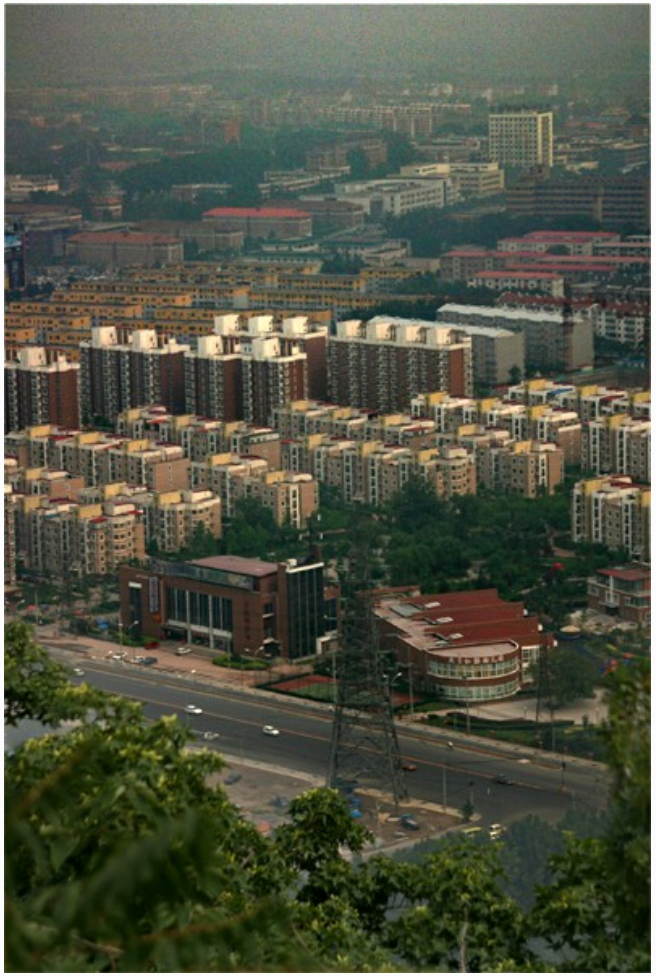
```
void tiraHaze(ImageType::Pointer image_in, ImageGrayType::Pointer image_dark,
ImageType::Pointer image_out, std::vector<double> A) {

  const PixelComponent t_max = 0.1;

  for (unsigned int i = 0; i < size[0]; i++) {
      for (unsigned int j = 0; j < size[1]; j++) {
          PixelType pixel = image_in->GetPixel({i,j}), pixelout;
          PixelComponent t_dark = image_dark->GetPixel({i,j});
          double t = std::max(t_max,t_dark);

          pixelout[0] = ((double) pixel[0] - A[0])/t + A[0];
          pixelout[1] = ((double) pixel[1] - A[1])/t + A[1];
          pixelout[2] = ((double) pixel[2] - A[2])/t + A[2];

          pixelout[0] = std::min(1.0, std::max(0.0, (double) pixelout[0]));
          pixelout[1] = std::min(1.0, std::max(0.0, (double) pixelout[1]));
          pixelout[2] = std::min(1.0, std::max(0.0, (double) pixelout[2]));

          image_out->SetPixel({i,j}, pixelout);
      }
  }
}
```

$$J(x) = \frac{I(x) - A}{max(t(x), t_0)} + A$$

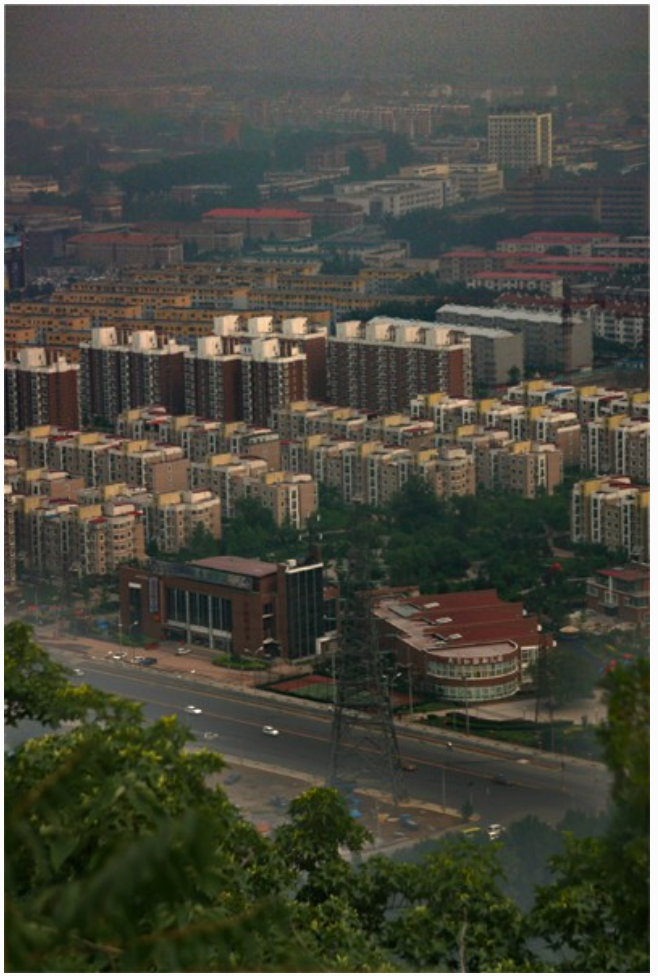
Resultados



Original

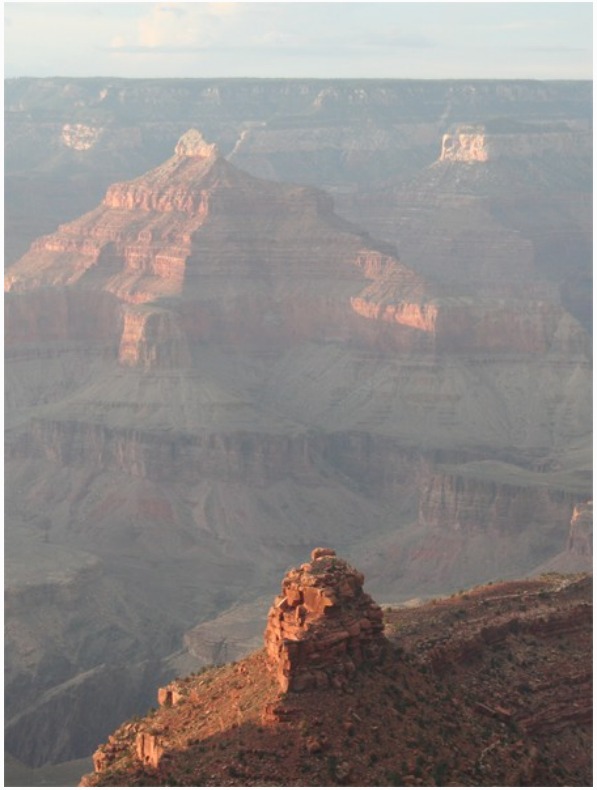


Paper

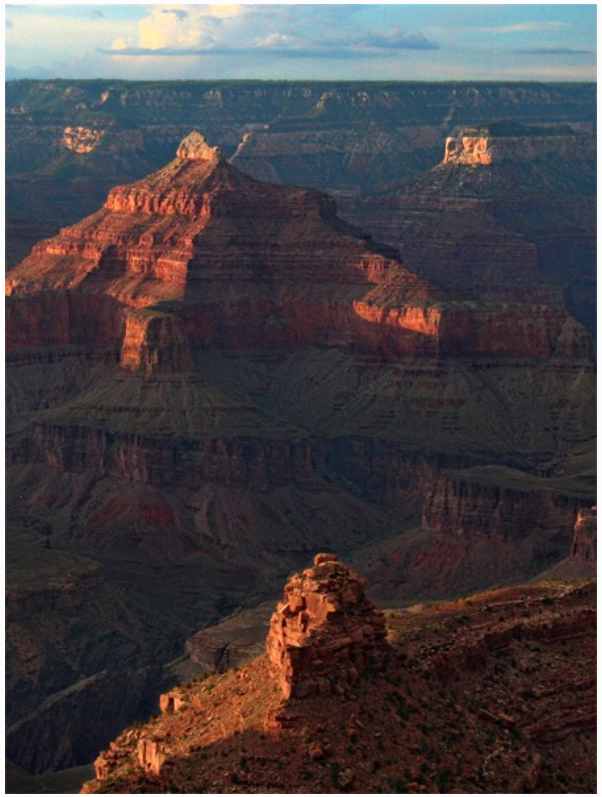


Implementação

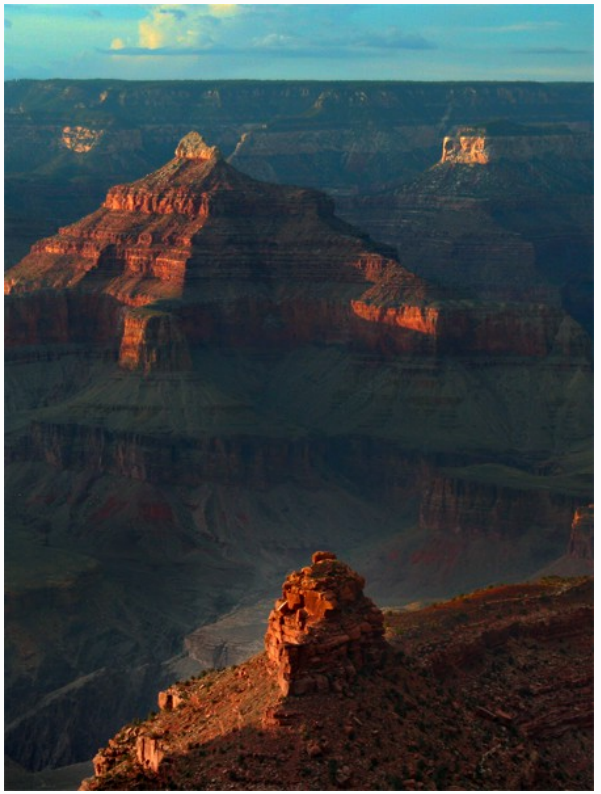
Resultados



Original



Paper



Implementação

Resultados

Original

Paper

Implementação

Resultados

Resultados

Resultados

Resultados

Resultados

Resultados inesperados

FIM