Machine Learning Homework 6

Environment

- Language: C++
- Standard: C++20

Code with detailed explanations

Libraries

Used library => Corresponding library in homework description

- Eigen => numpy
- OpenCV => visualization
- Boost (for access file system only)
- OpenMP (for parallel acceleration)

Visualization

- **Main part**

  > Use `fittingHistory` which records the labels for every iterations to make a video.
  >
  > The duration of one iteration is 1s.
  >
  > `cv::addWeighted` function is used to combine the clustering results (segmentation mask) with original image.
  >
  > `drawMask` function is used to convert the labels to segmentation mask.

```
    kernelKMeans.fit(kernel);
    const std::vector<Eigen::VectorXi> &fittingHistory = kernelKMeans.getFittingHistory();

    writer.open(imageFile + "_video.mp4", codec, fps, image.size());
    // check if we succeeded
    if (!writer.isOpened())
    {
        throw std::runtime_error("Could not open the output video file for write");
    }

    for (int i = 0; i < fps; i++)
    {
        writer.write(image);
    }

    cv::Mat result;
    cv::Mat mask;
    for (std::size_t i = 1; i < fittingHistory.size(); i++)
    {
        mask = drawMask(fittingHistory[i], numberOfClusters, image.cols, image.rows);
        cv::addWeighted(image, 0.5, mask, 0.5, 0, result);
        for (int i = 0; i < fps; i++)
        {
            writer.write(result);
        }
    }

    writer.release();

    cv::imwrite(imageFile + "_mask.png", mask);
    cv::imwrite(imageFile + "_final.png", result);
```

- **Draw mask**

> Convert the labels to segmentation mask.
>
> `labels` contains the clustering results, from 0 to k-1.
>
> `cv::applyColorMap` is used to map the labels to the corresponding color to avoid duplicate colors.

```
cv::Mat drawMask(const Eigen::VectorXi &labels, int numberOfClusters, unsigned int width, unsigned int height)
{
    Eigen::VectorX<unsigned char> maskData = labels.cast<unsigned char>();
    cv::Mat mask = cv::Mat(cv::Size(width, height), CV_8UC1, reinterpret_cast<void *>(maskData.data()));

    mask *= (255 / numberOfClusters);

    cv::Mat bgrMask;
    cv::cvtColor(mask, bgrMask, cv::COLOR_GRAY2BGR);
    cv::applyColorMap(bgrMask, bgrMask, cv::COLORMAP_COOL);
    return bgrMask;
}
```

Part1

- **Kernel K-Means**
  - Pseudo-code

    > Ignore weight

ALGORITHM 1: Basic Batch Weighted Kernel k-means.

KERNEL_KMEANS_BATCH(K, k, w, t_{max}, $\{\pi_c^{(0)}\}_{c=1}^k$, $\{\pi_c\}_{c=1}^k$)
Input: K: kernel matrix, k: number of clusters, w: weights for each point, t_{max}: optional maximum number of iterations, $\{\pi_c^{(0)}\}_{c=1}^k$: optional initial clusters
Output: $\{\pi_c\}_{c=1}^k$: final partitioning of the points
1. If no initial clustering is given, initialize the k clusters $\pi_1^{(0)}, ..., \pi_k^{(0)}$ (i.e., randomly). Set $t = 0$.
2. For each point $\mathbf{a}_i$ and every cluster c, compute

$$d(\mathbf{a}_i, \mathbf{m}_c) = K_{ii} - \frac{2\sum_{\mathbf{a}_j \in \pi_c^{(t)}} w_j K_{ij}}{\sum_{\mathbf{a}_j \in \pi_c^{(t)}} w_j} + \frac{\sum_{\mathbf{a}_j, \mathbf{a}_l \in \pi_c^{(t)}} w_j w_l K_{jl}}{(\sum_{\mathbf{a}_j \in \pi_c^{(t)}} w_j)^2}.$$

3. Find $c^*(\mathbf{a}_i) = \text{argmin}_c d(\mathbf{a}_i, \mathbf{m}_c)$, resolving ties arbitrarily. Compute the updated clusters as

$$\pi_c^{(t+1)} = \{\mathbf{a} \ : \ c^*(\mathbf{a}_i) = c\}.$$

4. If not converged or $t_{max} > t$, set $t = t + 1$ and go to Step 2; Otherwise, stop and output final clusters $\{\pi_c^{(t+1)}\}_{c=1}^k$.

- Main function

> Arguments
>
> path: the image data folder
>
> numberOfClusters: the number of clusters
>
> init: the selected initialization method, random or k-means++
>
> gamma1, gamma2: the hyper-parameter of RBF kernel

```cpp
void run(const fs::path &path, int numberOfClusters, mlhw6::KMeansInitMethods init, double gamma1, double gamma2)
{
    int fps = 30;
    int codec = cv::VideoWriter::fourcc('m', 'p', '4', 'v');
    cv::VideoWriter writer;

    mlhw6::KernelKMeans kernelKMeans(numberOfClusters, 200, 1234, init);

    for (auto imageFile : IMAGE_FILES)
    {
        std::cout << imageFile << std::endl;

        auto fileName = (path / imageFile).generic_string();

        // read image
        auto image = cv::imread(fileName, cv::ImreadModes::IMREAD_COLOR);

        // extract RGB values and coordinates
        auto [pixels, coordinates] = preprocess(image);

        // calculate kernel
        auto kernel = calculateKernel(pixels, coordinates, gamma1, gamma2);

        kernelKMeans.fit(kernel);
        const std::vector<Eigen::VectorXi> &fittingHistory = kernelKMeans.getFittingHistory();

        writer.open(imageFile + "_video.mp4", codec, fps, image.size());
```

- `kmeans.h` header: k-means related classes

> Follow the scikit-learn logic design.

```cpp
namespace mlhw6
{
    enum KMeansInitMethods
    {
        Random,
        Kmeansplusplus,
    };

    class BaseKMeans
    {
    public:
        BaseKMeans(int numberOfClusters, int maximumEpochs, int seed, KMeansInitMethods init);

        virtual void fit(const Eigen::Ref<const Eigen::MatrixXd> &x) = 0;
        virtual Eigen::VectorXi predict(const Eigen::Ref<const Eigen::MatrixXd> &x) const = 0;
        virtual Eigen::VectorXi fitAndPredict(const Eigen::Ref<const Eigen::MatrixXd> &x);

        const std::vector<Eigen::VectorXi>& getFittingHistory() const;

    protected:
        Eigen::VectorXi initializeCenters(const Eigen::Ref<const Eigen::MatrixXd> &x, KMeansInitMethods init, int seed, bool precomputed = false) const;

        int numberOfClusters;
        int maximumEpochs;
        int seed;
        KMeansInitMethods init;

        std::vector<Eigen::VectorXi> fittingHistory;
    };

    class KMeans : public virtual BaseKMeans
    {
    public:
        KMeans(int numberOfClusters, int maximumEpochs = 200, int seed = 1234, KMeansInitMethods init = KMeansInitMethods::Kmeansplusplus);

        void fit(const Eigen::Ref<const Eigen::MatrixXd> &x) override;
        Eigen::VectorXi predict(const Eigen::Ref<const Eigen::MatrixXd> &x) const override;
    private:
        Eigen::VectorXi assignLabels(const Eigen::Ref<const Eigen::MatrixXd> &x) const;
        Eigen::MatrixXd centers;
    };

    class KernelKMeans : public virtual BaseKMeans
    {
    public:
        KernelKMeans(int numberOfClusters, int maximumEpochs = 200, int seed = 1234, KMeansInitMethods init = KMeansInitMethods::Kmeansplusplus);

        void fit(const Eigen::Ref<const Eigen::MatrixXd> &x) override;
        Eigen::VectorXi predict(const Eigen::Ref<const Eigen::MatrixXd> &x) const override;
    private:
        Eigen::VectorXi assignLabels(const Eigen::Ref<const Eigen::MatrixXd> &x) const;
    };
}
```

- `KernelKMeans` constructor

```cpp
KernelKMeans::KernelKMeans(int numberOfClusters, int maximumEpochs, int seed, KMeansInitMethods init) : BaseKMeans(numberOfClusters, maximumEpochs, seed, init)
{
}
```

```cpp
BaseKMeans::BaseKMeans(int numberOfClusters, int maximumEpochs, int seed, KMeansInitMethods init) : numberOfClusters(numberOfClusters), maximumEpochs(maximumEpochs), seed(seed), init(init)
{
    if (!numberOfClusters > 0)
    {
        throw std::runtime_error("The number of clusters should be larger than 0.");
    }
}
```

- `preprocess` function: extract RGB values and coordinates

> Arguments
>
> image: (H, W, 3), BGR values

```cpp
std::pair<Eigen::MatrixX3d, Eigen::MatrixX2i> preprocess(const cv::Mat &image)
{
    auto rows = image.rows;
    auto columns = image.cols;
    auto size = rows * columns;

    cv::Mat rgb;
    cv::cvtColor(image, rgb, cv::COLOR_BGR2RGB);

    std::vector<int> coordinates(size * 2);
#pragma omp parallel for collapse(2)
    for (unsigned int i = 0; i < rows; i++)
    {
        for (unsigned int j = 0; j < columns; j++)
        {
            auto index = (i * columns + j) * 2;
            coordinates[index] = i;
            coordinates[index + 1] = j;
        }
    }

    using MatrixX3ucRowMajor = Eigen::Matrix<unsigned char, Eigen::Dynamic, 3, Eigen::RowMajor>;
    using MatrixX2iRowMajor = Eigen::Matrix<int, Eigen::Dynamic, 2, Eigen::RowMajor>;

    return std::make_pair<Eigen::MatrixX3d, Eigen::MatrixX2i>(
        MatrixX3ucRowMajor::Map(rgb.data, size, 3).cast<double>(), MatrixX2iRowMajor::Map(coordinates.data(), size, 2));
}
```

- `calculateKernel` function: calculate all kernel values

> Arguments
>
> pixels: (N, 3), RGB values
>
> coordinates: (N, 2), coordinates
>
> gamma1: γ_c scalar
>
> gamma2: γ_s scalar
>
> Formula

$$k(x, x') = e^{-\gamma_s \|S(x) - S(x')\|^2} \times e^{-\gamma_c \|C(x) - C(x')\|^2}$$

$S(x)$ is the spatial information (coordinate)

$C(x)$ is the color information (RGB)

```cpp
Eigen::MatrixXd calculateKernel(const Eigen::MatrixX3d &pixels, const Eigen::MatrixX2i &coordinates, double gamma1, double gamma2)
{
    auto colorKernel = mlhw6::rbf(pixels, pixels, gamma1);

    const Eigen::MatrixX2d &tmp = coordinates.cast<double>();
    auto coordinateKernel = mlhw6::rbf(tmp, tmp, gamma2);

    return colorKernel.cwiseProduct(coordinateKernel);
}
```

- `rbf` function: RBF kernel

  Arguments

  x1: x vector

  x2: x' vector

  gamma: γ scalar

  Formula

  $k(x, x') = e^{-\gamma ||x - x'||^2}$

```cpp
    template <typename DerivedA, typename DerivedB, typename Out = Eigen::Matrix<double, DerivedA::RowsAtCompileTime, DerivedB::RowsAtCompileTime>>
    Out rbf(const Eigen::MatrixBase<DerivedA> &x1, const Eigen::MatrixBase<DerivedB> &x2, double gamma)
    {
        Out result(x1.rows(), x2.rows());
#pragma omp parallel for
        for (Eigen::Index i = 0; i < x1.rows(); i++)
        {
            result.row(i) = (-gamma * (x2.rowwise() - x1.row(i)).rowwise().squaredNorm().transpose()).array().exp();
        }
        return result;
    }
```

- `kernelKMeans.fit` function: fit the data

  Previously, we already calculated the kernel first.

  So, we use the precomputed kernel values directly.

  Arguments

  x: (N, N), the kernel values (gram matrix, similarity matrix)

  Steps

  1. Pick k centers
  2. Calculate the cost between the data and centers
  3. Assign the label which has the smallest distance to the data
  4. Keep repeating 2, 3 step until the labels are not changed

  `fittingHistory` is used to store the labels for every iterations.

  Note: At the line 183, we do $1 - x$ to get the distance matrix.

```cpp
void KernelKMeans::fit(const Eigen::Ref<const Eigen::MatrixXd> &x)
{
    this->fittingHistory = std::vector<Eigen::VectorXi>{Eigen::VectorXi::Constant(x.rows(), -1)};

    // x is similarity matrix (gram matrix)
    auto centers = this->initializeCenters(1 - x.array(), this->init, this->seed, true);
    this->fittingHistory.back()(centers).setLinSpaced(0, this->numberOfClusters - 1);

    int epoch = 0;
    bool sameLabels = false;
    do
    {
        this->fittingHistory.push_back(this->assignLabels(x));

        sameLabels = (this->fittingHistory[epoch].array() == this->fittingHistory[epoch + 1].array()).all();
        epoch++;
    } while (!sameLabels && epoch < this->maximumEpochs);

    std::cout << "Finished at " << epoch << " epoch" << std::endl;
}
```

- `initializeCenters` function: pick k centers initialized by the selected method

  > Arguments
  >
  > x: (N, N) precomputed distance matrix or (N, features) data
  >
  > init: the selected initialization method, random or k-means++
  >
  > seed: the random seed
  >
  > precomputed: x is the precomputed distance matrix or not.

```cpp
Eigen::VectorXi BaseKMeans::initializeCenters(const Eigen::Ref<const Eigen::MatrixXd> &x, KMeansInitMethods init, int seed, bool precomputed) const
{
    if (precomputed)
    {
        if (x.rows() != x.cols())
        {
            throw std::runtime_error("The precomputed x should be a squared matrix.");
        }
    }

    switch (init)
    {
    case KMeansInitMethods::Random:
        return randomInitialization(x, this->numberOfClusters, seed);
    case KMeansInitMethods::Kmeansplusplus:
        return kMeansPlusPlusInitialization(x, this->numberOfClusters, seed, precomputed);
    default:
        throw std::runtime_error("The initialization method is not supported.");
    };
}
```

- `randomInitialization` function: randomly pick k centers

  > Arguments
  >
  > x: (N, N) precomputed distance matrix or (N, features) data
  >
  > numberOfClusters: k clusters
  >
  > seed: the random seed
  >
  > Steps
  >
  > 1. generate the sequence of indexes, 0 ~ N-1
  > 2. shuffle the sequence
  > 3. pick the top k rows as the centers

```cpp
Eigen::VectorXi randomInitialization(const Eigen::Ref<const Eigen::MatrixXd> &x, int numberOfClusters, int seed)
{
    auto rng = std::mt19937_64(seed);
    std::vector<int> sequence(x.rows());
    std::iota(sequence.begin(), sequence.end(), 0);
    std::shuffle(sequence.begin(), sequence.end(), rng);

    Eigen::Map<Eigen::VectorXi> tmp = Eigen::VectorXi::Map(sequence.data(), sequence.size());
    return tmp.topRows(numberOfClusters);
}
```

- `assignLabels` function: calculate the cost and assign labels

  > Arguments
  >
  > x: (N, N), the kernel values
  >
  > Steps
  >
  > 1. Calculate the cost between the data and centers
  >
  > $$\mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_p\sum_q \alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$
  >
  > 2. Assign the label which has the smallest distance to the data

```cpp
Eigen::VectorXi KernelKMeans::assignLabels(const Eigen::Ref<const Eigen::MatrixXd> &x) const
{
    Eigen::MatrixXd distance(x.rows(), this->numberOfClusters);
#pragma omp parallel for
    for (int k = 0; k < this->numberOfClusters; k++)
    {
        Eigen::VectorXd selector = this->fittingHistory.back().cwiseEqual(k).cast<double>();
        auto numberOfXInKCluster = selector.sum();
        Eigen::MatrixXd xToKCluster = x * selector.asDiagonal();

        Eigen::VectorXd secondTerm = 2 * xToKCluster.rowwise().sum() / numberOfXInKCluster;
        auto thirdTerm = (selector.asDiagonal() * xToKCluster).sum() / std::pow(numberOfXInKCluster, 2);
        distance(Eigen::all, k) = (x.diagonal() - secondTerm).array() + thirdTerm;
    }

    std::vector<int> labels(x.rows());
#pragma omp parallel for
    for (Eigen::Index i = 0; i < x.rows(); i++)
    {
        // find nearest neighbor
        distance.row(i).minCoeff(&labels[i]);
    }
    return Eigen::VectorXi::Map(labels.data(), labels.size());
}
```

- **Spectral Clustering**
  -

Part2

> Use command to control the number of clusters and other parameters.

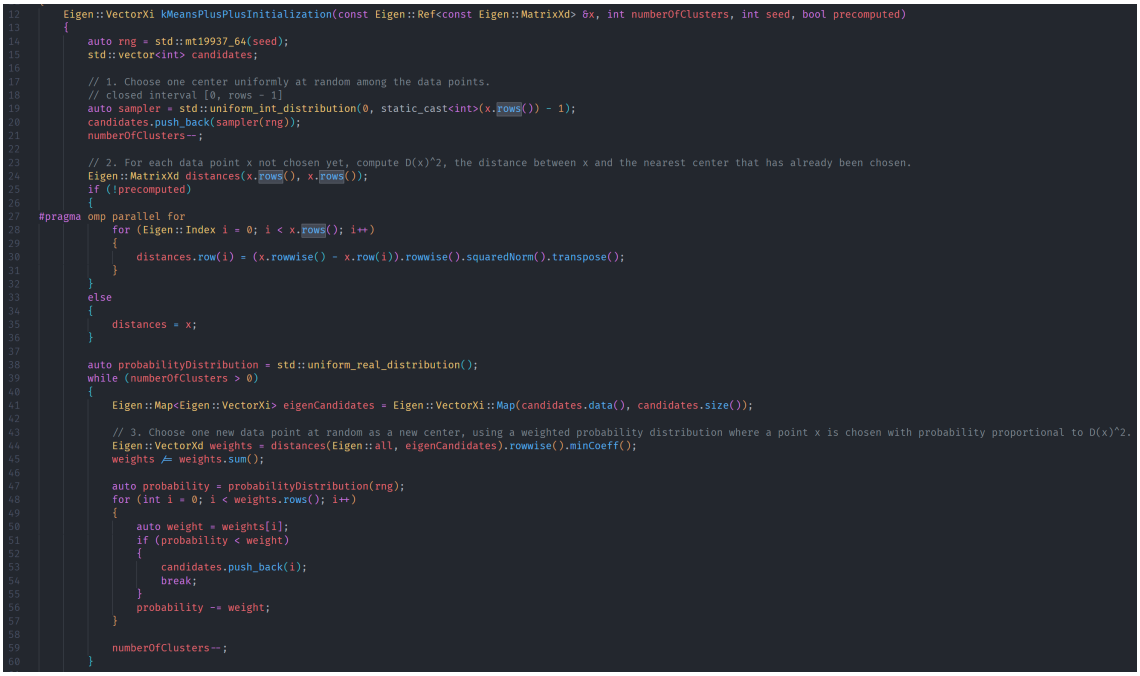
```cpp
int main(int argc, char *argv[])
{
    if (argc < 6)
    {
        std::cerr << "Usage: " << argv[0] << " <data path> <number of cluster> <init> <gamma1> <gamma2>" << std::endl;
        return 1;
    }
```

```
"version": "2.0.0",
"tasks": [
    {
        "label": "launch kernel kmeans",
        "type": "shell",
        "command": "${command:cmake.launchTargetPath}",
        "args": ["${cwd}/data", "3", "1", "0.00001", "0.00001", ">", "output.log"],
        "options": {
            "cwd": "${workspaceFolder}/build",
            "env": {
                "OMP_NUM_THREADS": "16"
            }
        }
    },
    {
        "label": "launch spectral clustering",
        "type": "shell",
        "command": "${command:cmake.launchTargetPath}",
        "args": ["${cwd}/data", "3", "1", "0.00001", "0.00001", ">", "output.log"],
        "options": {
            "cwd": "${workspaceFolder}/build",
            "env": {
                "OMP_NUM_THREADS": "16"
            }
        }
    }
]
```

Part3

- **Kernel K-Means**: K-Means++ initialization

> Arguments
>
> x: (N, N) precomputed distance matrix or (N, features) data
>
> numberOfClusters: k clusters
>
> seed: the random seed
>
> Steps
>
> ```
> 1. Choose one center uniformly at random among the data points.
> 1. For each data point x not chosen yet, compute $D(x)^2$ (the squared Euclidean distance) or use the precomputed distance matrix, the distance between x and the nearest center that has already been chosen.
> 1. Choose one new data point at random as a new center, using a weighted probability distribution where a point x is chosen with probability proportional to $D(x)^2$. (The farthest point will be chosen.)
> ```



```cpp
Eigen::VectorXi kMeansPlusPlusInitialization(const Eigen::Ref<const Eigen::MatrixXd> &x, int numberOfClusters, int seed, bool precomputed)
{
    auto rng = std::mt19937_64(seed);
    std::vector<int> candidates;

    // 1. Choose one center uniformly at random among the data points.
    // closed interval [0, rows - 1]
    auto sampler = std::uniform_int_distribution(0, static_cast<int>(x.rows()) - 1);
    candidates.push_back(sampler(rng));
    numberOfClusters--;

    // 2. For each data point x not chosen yet, compute D(x)^2, the distance between x and the nearest center that has already been chosen.
    Eigen::MatrixXd distances(x.rows(), x.rows());
    if (!precomputed)
    {
#pragma omp parallel for
        for (Eigen::Index i = 0; i < x.rows(); i++)
        {
            distances.row(i) = (x.rowwise() - x.row(i)).rowwise().squaredNorm().transpose();
        }
    }
    else
    {
        distances = x;
    }

    auto probabilityDistribution = std::uniform_real_distribution();
    while (numberOfClusters > 0)
    {
        Eigen::Map<Eigen::VectorXi> eigenCandidates = Eigen::VectorXi::Map(candidates.data(), candidates.size());

        // 3. Choose one new data point at random as a new center, using a weighted probability distribution where a point x is chosen with probability proportional to D(x)^2.
        Eigen::VectorXd weights = distances(Eigen::all, eigenCandidates).rowwise().minCoeff();
        weights /= weights.sum();

        auto probability = probabilityDistribution(rng);
        for (int i = 0; i < weights.rows(); i++)
        {
            auto weight = weights[i];
            if (probability < weight)
            {
                candidates.push_back(i);
                break;
            }
            probability -= weight;
        }

        numberOfClusters--;
    }
```

- **Spectral Clustering**

Part4

Experiments settings and results & Discussion

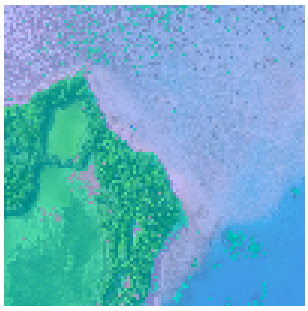
Part1

- **Kernel K-Means**

> Number of clusters: 2
>
> Initialization method: random
>
> Gamma1: 0.00001
>
> Gamma2: 0.00001

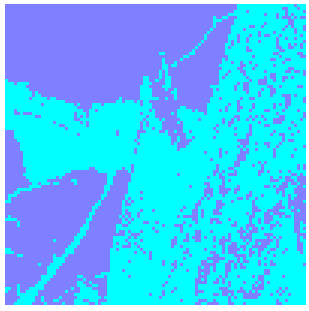
  - image1
    - Video
    - Final

        - Mask



    - image2
        - Video
        - Final

        - Mask



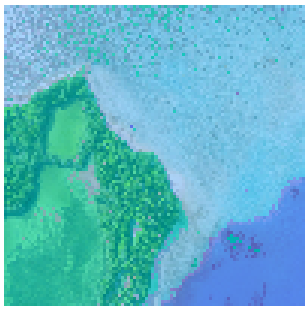
- **Spectral Clustering**

Part2

- **Kernel K-Means**

> Initialization method: random
>
> Gamma1: 0.00001
>
> Gamma2: 0.00001

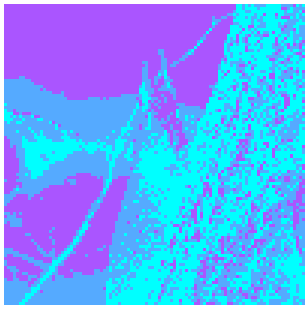
    - K = 3 · image1
        - Video
        - Final



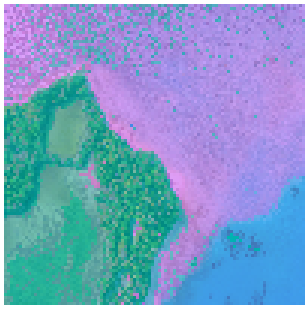
- Mask

- K = 3 · image2
  - Video
  - Final

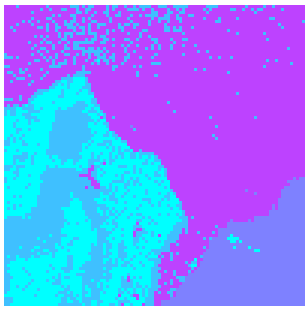
  - Mask



- K = 4 · image1
  - Video
  - Final

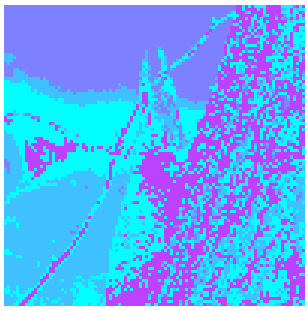


  - Mask



- K = 4 · image2
  - Video
  - Final

  - Mask



- **Spectral Clustering**

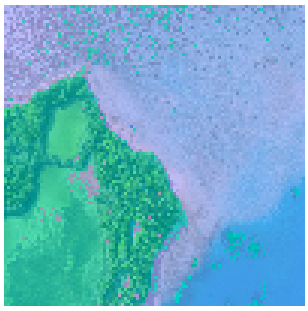
Part3

- **Kernel K-Means**

  I think there is no significant difference between random and k-means++.

  > Initialization method: k-means++
  >
  > Gamma1: 0.00001
  >
  > Gamma2: 0.00001

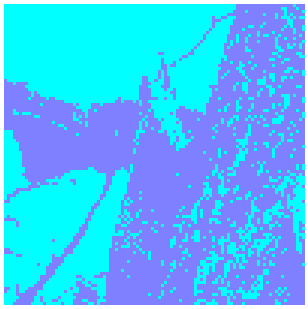
  - K = 2 · image1
    - Video
    - Final



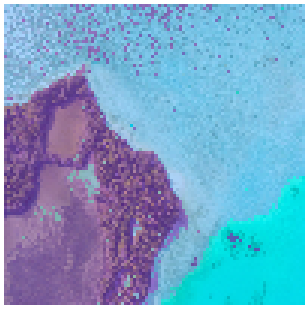
- Mask

- K = 2 · image2
  - Video
  - Final

  - Mask



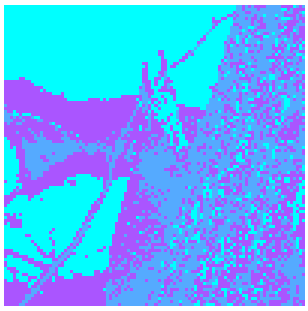
- K = 3 · image1
  - Video
  - Final



  - Mask

- K = 3 · image2
  - Video
  - Final

  - Mask



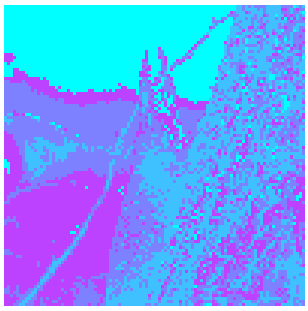
- K = 4 · image1
  - Video
  - Final

  - Mask

- K = 4 · image2
  - Video

- Final

- Mask



- **Spectral Clustering**

Part4

Observations and discussion

- Coming soon...