CR11 — Mathematical methods for image synthesis

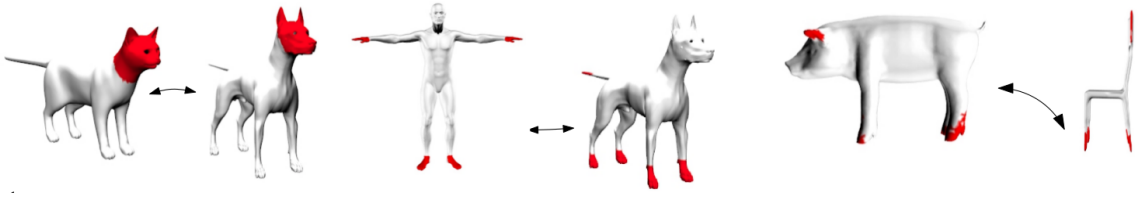
Simon Mauras

December 20th 2017

1 Article: Stable Region Correspondences

1.1 Motivations

The article I decided to study is "Stable Region Correspondences Between Non-Isometric Shapes". Authors are V. Ganapathi-Subramanian, B.Thibert, M. Ovsjanikov and L. Guibas. It has been published in *Computer Graphics Forum*. Vol. 35. No. 5. 2016.



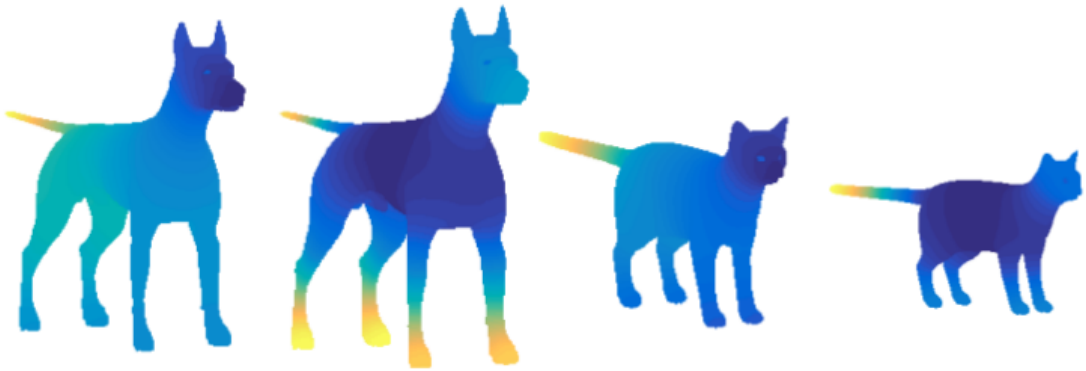
The goal of this article is to find similar regions between two shape. The case where both shapes are isometric is easier, as there is a one to one map from between the set of vertices. Here the approach is to use classical feature functions (e.g. Gaussian curvature) to achieve good and stable results.

1.2 Affinity Matrix

We are given two shapes as triangulated meshes. Their vertex sets are $S^{(1)} = \{p_1, \dots, p_{d_1}\}$ and $S^{(2)} = \{q_1, \dots, q_{d_2}\}$. We define two functions $f^{(1)}$ and $f^{(2)}$ implementing the same feature.

$$f^{(1)} : S^{(1)} \mapsto \mathbb{R} \quad \text{and} \quad f^{(2)} : S^{(2)} \mapsto \mathbb{R}$$

In the image below (from the article), are ploted two features : a multiscale mean curvature and a WKS signature.



We can notice that even if values are different on the two shapes, the rank of a value is meaningfull.

We define to permutations $r^{(1)}$ and $r^{(2)}$ of $\mathfrak{S}_{d_1}$ and $\mathfrak{S}_{d_2}$:

$$r_i^{(1)} = j \text{ if } f^{(1)}(p_j) \text{ is the } i^{th} \text{ value in sorted order.}$$

$$r_i^{(2)} = j \text{ if } f^{(2)}(q_j) \text{ is the } i^{th} \text{ value in sorted order.}$$

Let K be an integer that divides d_1 and d_2. For $1 \leq k \leq K$:

$$C_k^{(1)} = \left\{ r_i^{(1)} \;\middle|\; (k-1)\frac{d_1}{K} \lneq i \leq k\frac{d_1}{K} \right\}$$

$$C_k^{(2)} = \left\{ r_i^{(2)} \;\middle|\; (k-1)\frac{d_2}{K} \lneq i \leq k\frac{d_2}{K} \right\}$$

The affinity matrix of feature $(f^{(1)}, f^{(2)})$ can be defined as follow.

$$W = \sum_{k=1}^{K} \mathbb{1}_{C_k^{(2)}} \cdot \mathbb{1}_{C_k^{(1)}}^T = \underbrace{\left(\begin{array}{c|c|c} \begin{matrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{matrix} & & \\ \hline & \begin{matrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{matrix} & \\ \hline & & \begin{matrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{matrix} \end{array}\right)}_{\begin{matrix} r_1^{(1)} & r_2^{(1)} & & & r_{d_1}^{(1)} \end{matrix}} \begin{matrix} r_1^{(2)} \\ r_2^{(2)} \\ \\ \\ r_{d_2}^{(2)} \end{matrix}$$

We can notice that coefficient i, j of W is equal to 1 if $f^{(1)}(p_j)$ and $f^{(2)}(q_i)$ have similar ranks.

If in a more general setting we consider N different features, we can define the affinity matrix is the sum of each individual affinity matrix. Then we "normalize" the matrix by multiplying each coefficient by $K/(Nd_1d_2)$.

1.3 Stable pairs

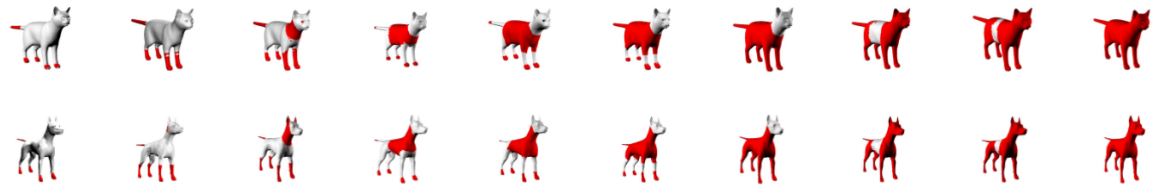
To define stable pairs, and be able to compute similar regions between two shapes, some definitions are still required. Let M be a matrix in $\mathcal{M}_n(\mathbb{R})$.

- The norm $||M||$ is defined as the sum of the absolute value of all coefficients.
- The submatrix $M_{I,J}$ is obtained when keeping only lines $I \subseteq \{1, \dots, n\}$ and columns $J \subseteq \{1, \dots, n\}$.

We are now ready to define a stable pair $(\Omega^{(1)}, \Omega^{(2)})$ of size (n, m). It is a subset $\Omega^{(1)} \times \Omega^{(2)} \subseteq S^{(1)} \times S^{(2)}$ with $|\Omega^{(1)}| = n$ and $|\Omega^{(2)}| = m$, such that $||W_{\Omega^{(2)},\Omega^{(1)}}||$ is maximal.

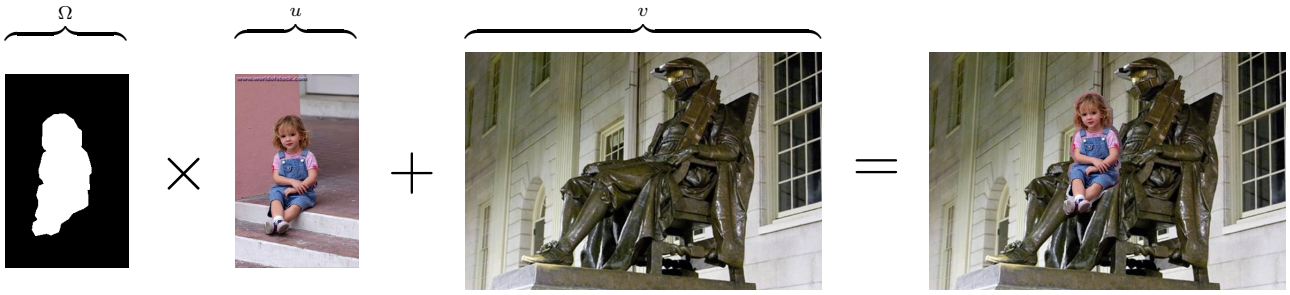
The existence of a global maximum is guaranted because there is only a finite number of submatrices. However we are interested in an efficient solution, that is going to be compute as the fixpoint of an operator (unfortunately non-linear).

The procedure obtained is stable when adding i.i.d. random features. The image below gives example of the results that can be obtained.



2 Project: Poisson Image Editing

Our goal is to integrate an object (here a little girl) in an image (background). The figure below gives the naive solution for this problem. We can notice that the quality of the result is poor, the separation between the object and the background being visible.



We have seen in class that the eye is more sensitive to color differences than absolute color values. A solution is the following, let $\tilde{u}$ such that $\tilde{u} = v$ on $\partial\Omega$ and $\mathcal{L}(\tilde{u}) = \int_\Omega |\nabla(\tilde{u} - u)|^2$ is minimized.

This approach is developped in an article by Pérez, Gangnet and Blake: "Poisson image editing." in ACM Transactions on graphics (TOG). Vol. 22. No. 3. ACM, 2003.

2.1 Poisson equation

We can deduce a Poisson equation (using calculus of variation) from the previous minimization problem.

$$\nabla^2 \tilde{u} = 0 \quad \text{in } \Omega$$

This equation can be discretized, to obtain a sparse linear system. We can solve it using the conjugate gradient method.

2.2 Implementation

I decided to implement this project in Python, as manipulating images is easier in this language. However performances are very far from those obtained using a low level language as C.

2.3 Results

Below are pictures after 1, 10, 100, 1000 and 10000 iterations of the conjugate gradient, starting with the 0 vector.

Execution time of 10000 iterations of the conjugate gradient method is about 4.5 seconds. We solve three similar systems (3 colors: Red, Green and Blue), the total running time is therefore about 15 seconds.

If we start from a cleverer vector (the naive solution), we reach an approximate solution (relative or absolute residual $\leq 10^{-5}$) in about 270 iterations. Thus the total running time is reduced to 0.5 seconds.

3 Project: Image Inpainting

The goal of this project is to remove a part of an image (*e.g.* a person, ...) and replace it by a texture automaticaly generated from the rest of the image.

The approach used is developped in the article "Image Analogies" from Hertzmann *et al.*

3.1 Description of the algorithm

The idea is to replace the value of a pixel by the value of the pixel with the most similar neighborhood. In order to achieve good results, pixels are treated only once, in a linear scan.

However, some techniques must be used to have a satisfactory output: multiscale methods.

- Recurcively divide image size by two
- Use smaller solution to deduce a "best guess"
- Compute solution

3.2 Implementation

I implemented in Python the algorithm described above. When reaching an image with only dozens of pixels, we can assume that we already have a good texture, replacing the object we wanted to remove.

To compute the best neighborhood, I choosed to use a datastructure called KD-Tree to speed up the process of finding in a dictionary the nearest neighbor of the vector containing the values of the neighboring pixels.

Below are the results obtain with the first version of the algorithm.

We can notice that the texture generated doesn't exactly match the expectation we had. In order to generate a good texture, we provide the algorithm a "hint". Then we use KMeans clustering to build a partition of the pixels in clusters of textures (using multiscape, hint image becomes blurred). We restrict the nearest neighbor search to the pixels of a cluster.

The total running time is about 90 seconds (80% of the time used to compute the last and biggest image). Without KD-Trees implemented in C (external library), 10 hours weren't enough to obtain a solution.

4 Project: Seam Carving

Reducing the size of an image can be done in several ways (resize, crop, ...). But an important part of the image can be either removed of distorded. The goal of seam carving is to detect "important" patterns, and preserve them when removing pixels from the image.

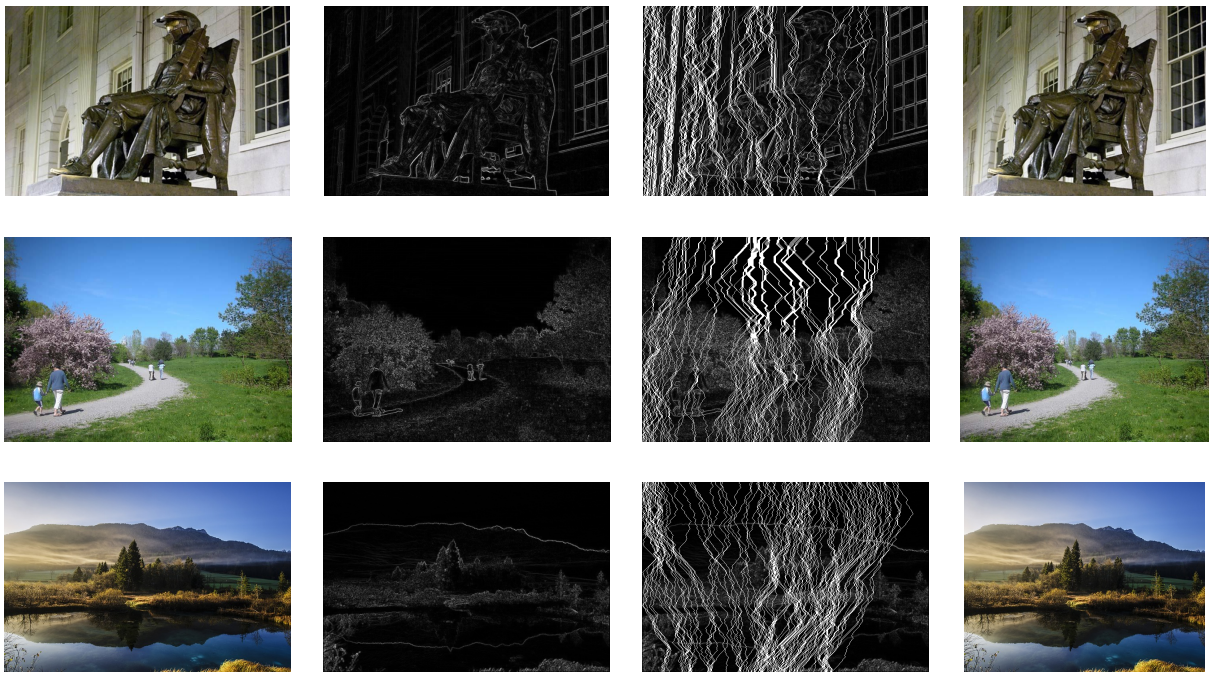
The method we describe has been introduced by Avidan and Shamir in an article named "Seam carving for content-aware image resizing", published in ACM Transactions on graphics, 2007.

4.1 Description of the algorithm

The main idea is that important part of an image have a lot of details. Therefore, if we use an edge detector (*e.g.* norm of the gradient of the value of pixels), we can quantify how much it would "cost" to remove one pixel from the image.
To remove one column from the image, without shifting to much the rest, we can decide to delete a "continuous" path from the top-most to the lowest row. The path minimizing the total cost can be computing using dynamic programming.
Once we remove one row, we can start over and remove an other one.

4.2 Implementation

I implemented the algorithm above in Python. In order to speed up the running time, I decided to keep the cost function from one iteration to one other, and setting the cost of already deleted pixels to $+\infty$ to avoid deleting them twice.



Running time for the first image is 2 seconds to compute the cost matrix, then 40 seconds to remove 100 columns. This can be easily improved, for example by using C or C++, indeed the algorithm only consists in 3 or 4 nested for loops, no external libraries are used (except to read an image from a file). Moreover, dynamic programming is one of the textbook examples when starting parallel programming or distributed systems, I haven't tried this approach but it must give good results.