MCMC inference over latent diffeomorphisms using parallel computing

Angel Yu

Massachusetts Institute of Technology
77 Massachusetts Avenue, Cambridge, MA 02139, USA
E–mail: angelyu@mit.edu

1 Introduction

Diffeomorphisms are bijective differentiable transformations such that the inverse is differentiable as well. They have many applications in computer vision such as correspondence based image warping. However, current representations of these spaces are complicated and hard to compute on large datasets. Freifeld et al. [1] proposed a representation of a subspace of these transformations that can be computed in high accuracy efficiently while maintaining the expressiveness of diffeomorphisms. In this project, we implemented this representation in both 1D and 2D and performed Monte Carlo Markov Chains (MCMC) inference in this space of transformations using parallel computing in Julia.

2 CPAB transformations

The proposed space of transformations is based on the integrations of Continuous Piecewise-Affine (CPA) velocity fields. Hence, they are referred to as CPA-based (CPAB) transformations. CPA velocity fields are continuous velocity fields that are piecewise affine with respect to a tessellation of the space. Because these are continuous velocity fields, we are able to define a trajectory $\phi(x,t)$ at every point x in the space and this is given by the integral equation:

$$\phi^\theta(x,t) = x + \int_0^t v^\theta(\phi^\theta(x,\tau))d\tau$$

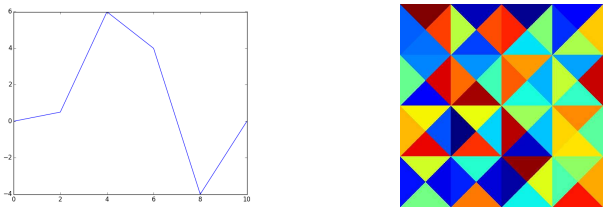
where θ is the parameter of the CPA velocity field and v^θ is velocity function. By fixing t and mapping all points to the point at time t on its trajectory, we obtain a CPAB transformation. We can easily see that these transformations are indeed diffeomorphisms as we can take the negative of the velocity field to obtain the inverse transformation.

To calculate this trajectory, we can use numerical methods to approximate the integral. However, Freifeld et. al. [1] showed that it can be calculated in a closed form if the point remains in the same cell of the tessellation, so we would only need to use the numerical approximation when crossing boundaries

of cells. This allows a highly accurate as well as efficient calculation of CPAB transformations

2.1 Tessalation and Parameters

For 1 dimensional spaces, we can easily tessellate the space into segments on the x-axis. As the velocity field is CPA, the velocity in each cell is linear and has to be continuous across different cells. An example of such a velocity field is given in Figure 1a. In the 1D case, the number parameters is given by the number of vertices which in this example is 6.



(a) (b)

Figure 1: (a) An example of a 1d velocity field. Here the tessellation contains 5 cells equally spaced between 0 and 10. (b) An example of a 2d tessellation we will use. Here the tessellation contains 64 cells.

For 2 dimensional spaces, we choose a tessalation which allows easy computation of which cell a point is in. We first partition the 2D into grids and then for each grid, we draw the diagonals to create 4 triangles. An example of such a tessellation is given in Figure 1b. There are 64 cells in this example tessellation but because the velocity field has to be continuous across cells, the parameter of velocity fields in this tessellation has a dimension of 58.

3 Inference

We now apply this transformation to the problem of correspondence based image warping. The problem is given 2 images and a set of correspondences, infer the underlying transformation that mapped the source image to the destination image. This is illustrated in Figure 2 where we would like to find a CPAB transformation that maps the red points to the blue points. This becomes an optimization problem and we can create a least squared objective function as follows:

$$f(\theta) = \sum_i ||T^\theta(x_i, t) - y_i||^2$$

where θ is the parameter of the CPAB transformation, T^θ is the CPAB transformation, t is the time to evaluate the trajectory, $X = x_1, x_2, \ldots, x_n$ are the source points and $Y = y_1, y_2, \ldots, y_n$ are the destination points.

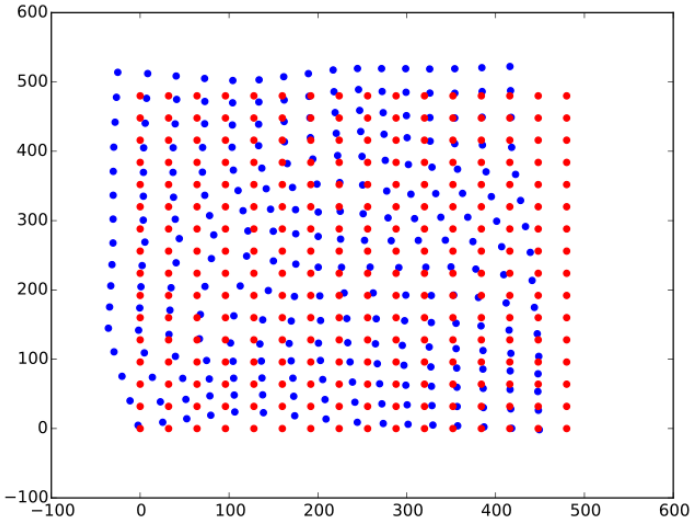


Figure 2: An inference problem: Inferring the underlying parameters of a CPAB transformations that maps the red points to the blue points

3.1 Gradient Descent

Gradient descent is one of the most popular methods in optimization as it is easy to implement and understand. Gradient descent finds a local minimum of an objective function by taking steps in the direction of the steepest descent. We start with an initial value of θ_0. At each iteration, we find the gradient $f'(\theta_i)$ numerically and update the parameter:

$$\theta_{i+1} = \theta_i - \gamma f'(\theta_i)$$

where γ is a chosen step size which determines the rate of convergence.

A significant drawback of using gradient descent is that it can only find local minimums and in most cases we would like to find the global minimum. A common work-around is to try multiple initial θ_0 and running gradient descent on each of them. However, even with this, it will perform poorly on a function with many local minimums.

3.2 Metropolis' Algorithm

Metropolis' Algorithm is an MCMC sampling algorithm that allows us to sample from a difficult to sample probability distribution $P(x)$. Again, we start off with an initial value of x_0. At each iteration, we have a symmetric proposal

distribution $Q(x|x_i)$. This proposal distribution is usually taken to be a Gaussian distribution centered at x_i. We generate a sample x' from this proposal distribution. We then calculate the acceptance ratio $\alpha = P(x')/P(x_i)$. We automatically accept this sample if the acceptance ratio $\alpha \geq 1$ and set $x_{i+1} = x'$. Otherwise we accept x' with probability α by setting $x_{i+1} = x'$ and reject with probability $1 - \alpha$ by setting $x_{i+1} = x_i$. Note that we don't actually need to be able to calculate $P(x)$, we just need to be able to calculate $g(x) \propto P(x)$ since we only need the ratio.

Using Metropolis' Algorithm to sample from a distribution $P(x)$, we can also find the mode of the distribution by storing the greatest value of $P(x)$ seen so far at each iteration.

We can also use it to optimize our objective function with respect to a prior distribution over θ. In our case, we would like to minimize $f(\theta) = \sum_i ||T^\theta(x_i, t) - y_i||^2$. To do this, we have the following likelihood distribution:

$$\begin{aligned} P(X|\theta) &= C_1 e^{-\frac{\sum_i ||T^\theta(x_i,t)-y_i||^2}{2\sigma^2}} \\ &= C_1 e^{-\frac{f(\theta)}{2\sigma^2}} \end{aligned}$$

which is based on a Multivariate Gaussian distribution where C_1 is a constant and σ is a parameter representing the standard deviation. We can see that this distribution obtains its maximum when $f(\theta)$ obtains its minimum. In addition, we also have a Gaussian prior over θ. So our target posterior distribution becomes:

$$\begin{aligned} P(\theta|X) &\propto P(X|\theta)P(\theta) \\ &= C_1 e^{-\frac{f(\theta)}{2\sigma^2}} C_2 e^{-\frac{1}{2}(\theta-\theta_\mu)^T \Sigma_\theta^{-1}(\theta-\theta_\mu)} \\ &\propto e^{-\frac{f(\theta)}{2\sigma^2} - \frac{1}{2}(\theta-\theta_\mu)^T \Sigma_\theta^{-1}(\theta-\theta_\mu)} \end{aligned} \tag{3.1}$$

where C_1, C_2 are constants, θ_μ is the mean of the prior and Σ_θ is the covariance of the prior. To make calculations easier, we can calculate the acceptance rate in the log domain:

$$\begin{aligned} \log(\alpha) &= \log\left(\frac{P(\theta'|X)}{P(\theta|X)}\right) \\ &= \frac{f(\theta) - f(\theta')}{2\sigma^2} + \frac{1}{2}(\theta - \theta_\mu)^T \Sigma_\theta^{-1}(\theta - \theta_\mu) - \frac{1}{2}(\theta' - \theta_\mu)^T \Sigma_\theta^{-1}(\theta' - \theta_\mu) \end{aligned}$$

This optimization method is likely to produce better results than gradient descent as we are accepting samples that might not be better than the previous sample. This allows us to not get stuck in local minimums which is one of the biggest problems in gradient descent.

3.3 Particle Filter

Another popular MCMC algorithm is the Particle Filter. Instead of sampling one sample each iteration in Metropolis, we now sample multiple samples (particles) each iteration that approximates the distribution. This algorithm is usually used for changing probability distributions such as in robotics or a video. However, we will apply it to a fixed probability distribution here. For a target distribution $P(x)$ we first initialize a set of particles randomly sampled from the prior. At each iteration, we calculate a weight for each particle proportional to their probability such that the weights sum up to 1. We then re-sample another set of particles from these particles based on the weights distribution. So, higher weighted samples will have a higher chance of being picked. After getting the re-sampled particles, for each particle, we perturb it with a perturbation distribution centered at that particle. We have now obtained our set of particles for the next iterations. After a few iterations, we should obtain a good approximation of the target distribution.

We now apply it to our problem. In our case, our weights w_j are defined to be:

$$\tilde{w_j} = P(\theta_j^i|X)$$
$$w_j = \frac{\tilde{w_j}}{\sum_k \tilde{w_k}}$$

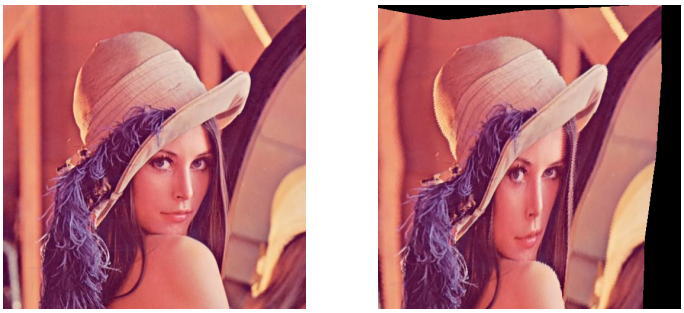
where θ_j^i is the jth particle in the ith iteration and $P(\theta|X)$ is given by Equation 3.1. Since we are doing operations on multiple particles in each iteration, we can see that we can easily parallelise over the particles. In addition, this algorithm does not have the problem of being stuck in local minimums as long as the perturbation distribution is chosen so that the model explores particles sufficiently far from the original particles.

4 Results

We implemented both 1-dimensional and 2-dimensional calculations of CPAB transformations in Julia. Figure 3 shows an example of a 2-dimensional CPAB transformation on an image. Since, we are computing each pixel independently, this problem is embarrassingly parallel. The time needed to compute a 2-dimensional CPAB transformation on a 512x512 image is shown in Table 1. Even though this problem is embarrassingly parallel, we can see there is much overhead in the parallelization. This is due to the communication overhead as we need to send parts of the image to different processes.

4.1 Gradient Descent

We used the Optim package and applied gradient descent to the problem described in Figure 2 namely inferring the underlying CPAB transformation given 16x16=256 correspondences. The algorithm converged after 359 and the result



(a) (b)

Figure 3: (a) Original Image. (b) Image after applying a CPAB transformation.

Number of processes	1	2	4	8
Time (s)	0.78	0.49	0.38	0.37
Speedup	1.00x	1.59x	2.05x	2.11x

Table 1: Time to compute a CPAB transformation on a 512x512 image.

is shown in Figure 4. As we can see, the transformation inferred only somewhat matches the underlying transformation. This is most likely because the transformation found is a local minimum in our objective function. We tried parallelizing over the 16x16 points when computing the objective function. However, because

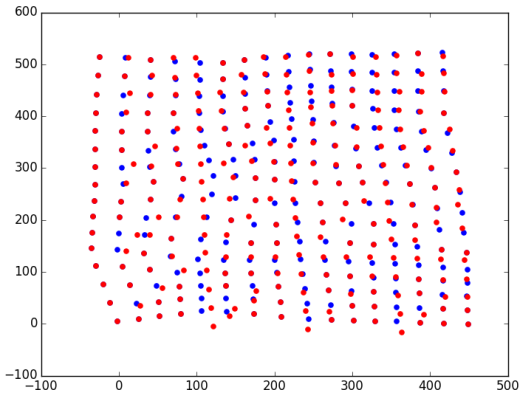
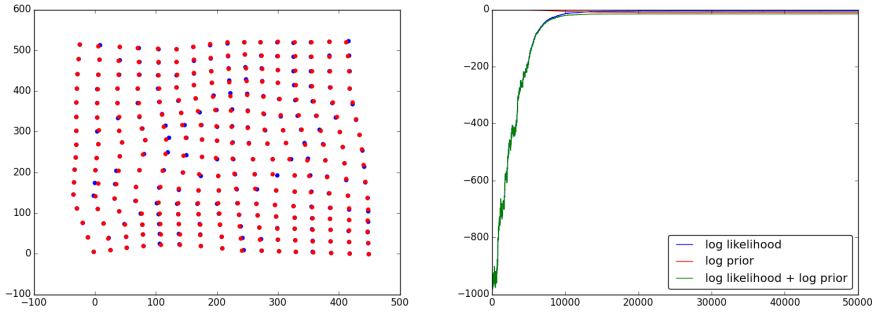


Figure 4: Applying gradient descent to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation

4.2 Metropolis' Algorithm

We implemented Metropolis' Algorithm described in Section 3.2 and applied it to infer the underlying CPAB transformation in Figure 2 given a prior distribution. The prior is a multivariate Gaussian distribution with a mean of $\vec{0}$ and some covariance `priorCov`. We ran 50,000 iterations and set our proposal distribution to be a Gaussian centered at the current sample with covariance 0.00001`priorCov`. The result of this algorithm is shown in Figure 5a. We can see that the points obtained using the inferred transformation match pretty well with the target points. This is much better compared to the results we obtained from using gradient descent. In Figure 5b, we can see that the algorithm converges at around 20,000 iterations. We tried parallelizing this by parallelizing the calculation of CPAB transformations described earlier. However, since there are only 256 points, it runs very fast and parallelizing it actually slows it down because of the overhead in communication. We actually found a Julia while running these experiments. It seems that there is a race condition when spawning the processes and a seg fault will occur when spawning a large number of processes. We submitted an issue ticket on github at: `https://github.com/JuliaLang/julia/issues/13999`.



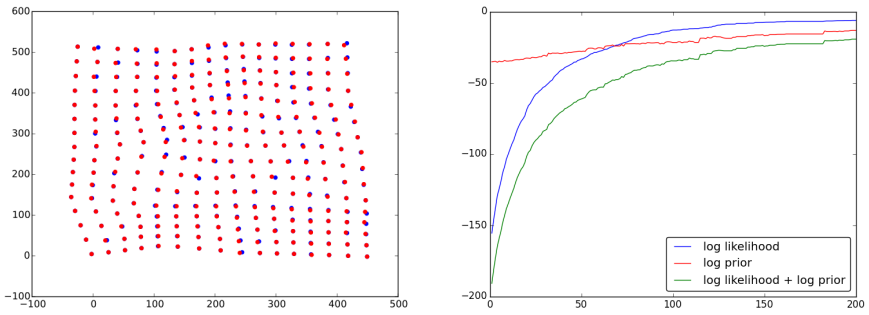
(a) (b)

Figure 5: (a) Applying Metropolis' Algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 50,000 iterations

4.3 Particle Filter

We implemented the Particle Filter algorithm described in Section 3.3 and applied it to infer the underlying CPAB transformation in Figure 2 given a prior distribution of θ. We again have the prior as a multivariate Gaussian distribution with a mean of $\vec{0}$ and covariance of `priorCov`. We took the number of particles to be 1,000 and the number of iterations to be 200. We have much less

iterations than in Metropolis' as now we explore 1,000 samples in each iteration instead of just 1. In addtition, we let the perturbation distribution be a Gaussian centered at the current particle with covariance 0.001`priorCov`. The result of this algorithm is shown in Figure 6a. We can see that the inferred points almost match the target points and it is comparable with the results from using Metropolis' algorithm. However, this algorithm is much easier to parallelize as we can parallelize over the particles. The time needed is summarized in Table 2. We can see that we get almost linear speedup when using 2 processes, but we get much less than linear speedup when using 4 and 8 processes. This is because there is a big communication overhead when re-sampling from the current particles as processes are pulling data from other processes. And this communication overhead dominates the time to compute the CPAB transformations when using many processes.



(a) (b)

Figure 6: (a) Applying the Particle Filter algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 200 iterations

Number of processes	1	2	4	8
Time (s)	199	105	63	54
Speedup	1.00x	1.90x	3.06x	3.67x

Table 2: Time to run the Particle Filter inference algorithm with 1,000 particles for 200 iterations on 256 correspondences

In addition to using 16x16=256 correspondences, we experimented with using full 512x512 correspondences. We ran the experiments on a cluster of 4 12-core machines with 24 threads each. The timing for them are summarized in Table 3 and the speedup is illustrated in Figure 7. We can see that we obtain almost linear speedup when parallelizing over 96 processes and it did not show signs on slowing down. In the end, we were limited by the number of machines with

the same architecture as Julia did not allows parallelization over machines with different CPU architectures. We realize it is hard to parallelize when machines have different architectures, but it will be a really nice feature to add.

Number of processes	1	8	24	48	72	96
Time (s)	2240	308.5	98.54	54.71	40.54	30.09
Speedup	1.00x	7.26x	22.7x	40.9x	55.3x	74.4x

Table 3: Time to run each iteration of the Particle Filter inference algorithm with 1,000 particles on 512x512 correspondences using distributed computing

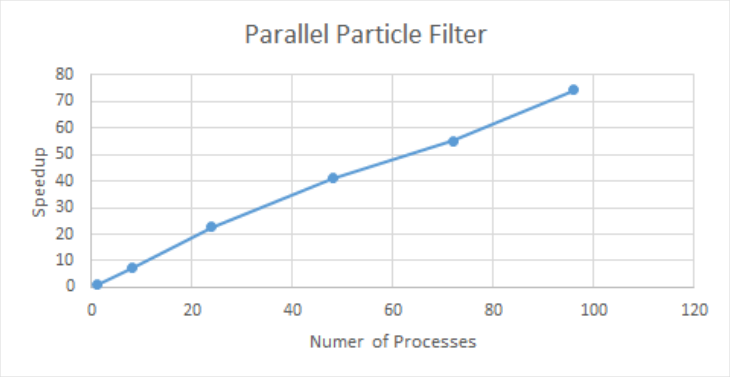


Figure 7: Graph showing the speedup as we parallelize across multiple machines

5 Conclusion

In this project, we implemented the calculation of CPAB transformations in Julia in both parallel and serial. We saw that it did not benefit too much from the parallelization as the time to compute the transformation was already quite fast and there was a good amount of overhead in spawning the processes. We also applied these transformations to solving the correspondence based inference problem and experimented with 3 different inference algorithms in Julia. We used Optim's Gradient Descent implementation and compared it with our implementations of Metropolis' Algorithm and Particle Filter. We found that Gradient Descent tends to get stuck at local minimums and both Metropolis' Algorithm and Particle Filter achieved much better accuracy. The results of those two however are quite similar, but Particle Filter allowed for easy parallelization over the number of particles and will therefore run much faster for bigger problems and good hardware. We saw that we got excellent speedup using the Particle Filter algorithm. The code for this project will be available on Github soon.

Acknowledgments. The author would like to thank Dr Oren Freifeld for his patient guidance, through which the author was exposed to CPAB transformations as well as MCMC methods and Professor Alan Edelman for offering 18.337/6.338 introducing the author to parallel computing in Julia.

References

[1] Freifeld, O., Hauberg, S., Batmanghelich K. and Fisher III, J. "Highly-Expressive Spaces of Well-Behaved Transformations: Keeping It Simple." ICCV 2015.