Single Photon Depth Sensing

Daniel Ayoub

December 17, 2016

Abstract

An implementation of a depth sensing algorithm using data collected via single-photon imaging. Traditional **LI**ght **D**etection **A**nd **R**anging (LIDAR) systems approximate subject distance by measuring photon travel times. These models use hundreds of photons per pixel to compensate for background light noise. The non-convex union-of-subspaces optimization constraint introduced by Shin et al. [1] achieves negligible error with only about fifteen photons per pixel by using a modified CoSaMP algorithm to estimate the depth of the scene as well as the amount of noise. In this project, I aim to formulate the depth-sensing problem in a way that relates to our class, implement the algorithm presented by Shin et al., and optimize the implementation to achieve a faster running time.

1 Background

With the advent of smarter devices operating autonomously in the world, giving these devices an accurate sight of the outside world has become very important. Humans and most animals have binocular vision, which provides a parallax effect which is then interpreted by the brain to give an accurate sense of depth of the world around is. The devices around us, however, are sometimes limited by space or cost to only have one light sensor, and other methods can be used to detect depth.

Using light to measure distance has many benefits over other detection systems, such as RADAR (which uses radio waves) or SONAR (which uses acoustic waves). SONAR systems can have higher error rates at low distance [4] than LIDAR systems, and RADAR systems require huge sensor dishes to detect the large wavelengths of radio light. LIDAR is particularly relevant today as a host of new applications for efficient, accurate, range detection are appearing; particularly, in autonomous vehicle computer vision. The particular algorithm presented by Shin et al. [1] only requires a few photons and no system calibration, making it most suitable for low-light situations where the background noise is dynamic.

LIDAR is a depth detection scheme that uses only one light sensor and one light emitter. LIDAR operates like this:

1. A laser flashes a pulse of light at a mirror, which directs the pulse towards the scene.
2. The pulse of light bounces off of the scene and back into the mirror, which directs it towards a light detector.
3. The light detector is triggered by the returning photons, and the total travel time between the laser emission and photon detection is recorded.
4. The mirror swivels around until data has been collected for the entire scene.
5. An algorithm uses the collected data to generate estimates of the scene depth.

In principle, the plan is a simple and effective way to accurately derive scene depth. Given an accurate travel time t, the scene depth d can be derived from the following simple linear equation:

$$d = \frac{ct}{2} \tag{1}$$

Where c is the speed of light, roughly $3 * 10^8$ m/s. In reality, however, detected data is corrupted by noise and there may be slight variations in consecutively detected arrival times. These two facts combined make

it a more complex task to accurately estimate the scene depth. Also, ideally, it would be nice to be able to perform this task with a very small number of measurements. If we only have to take few measurements we can reduce the amount of time that our sensor must operate for, as well as reduce the amount of data we must store for our detection algorithm to use.

The detection methods we've been studying in class lend themselves to this situation very well. We have a signal that is described by a physical model, but our measurements contain slight variations and are corrupted by noise. Therefore, with a clever problem formulation, we should be able to use some of our methods to come up with an effective analysis method.

2 Problem Statement

Photon arrival times can be modeled as a Poisson process, which has generic PMF:

$$p[k] = \frac{\lambda^k e^{-\lambda}}{k!} \tag{2}$$

The number of photons that arrive at the sensor depends on several parameters: let a, d, and b represent the subject reflectivity, depth, and the background noise present in a given pixel region. If a signal pulse $s(t)$ is fired, the reflected signal will have waveform

$$r(t) = as(t - 2d/c) + b \tag{3}$$

(Note that the offset in t is derived from equation 1) In our case, the waveform strength outputted by the laser is described by:

$$s_{rms}(t) \propto \exp(-t^2), \qquad t \in [0, T_r)$$

Now, we can discretize the light travelling time periods by choosing a small value ϵ that T_r is divisible by. Then, by letting $N = T_r/\epsilon$, we can define $\hat{\mathbf{x}}_{N\times 1}$ as a vector with one non-zero element. The scene depth is then derived by

$$d \approx \frac{c * \text{supp}(\hat{\mathbf{x}}) * \frac{2\epsilon+1}{2}}{2} \quad \pm \frac{\epsilon}{2}$$

Where the support of $\hat{\mathbf{x}}$, $\text{supp}(\hat{\mathbf{x}})$ denotes the index of the nonzero element in $\hat{\mathbf{x}}$.

$\mathbf{x}$ is then the sparse signal we need to estimate. Similarly, our measured signal is constrained to a histogram with parameters defined by the specifications of the single photon camera. The single photon

camera detects light for the same period of time, T_r, and detects in rounds of length Δ. Let $M = T_r/\Delta$, and $\mathbf{y}_{M\times 1}$ is the histogram of detected photon arrival times.

$$\mathbf{y}_k \sim \text{Poisson}\left(\int_{\Delta(k-1)}^{\Delta k} [\eta(as(t-2d/c)+b)+b_d]dt\right) \tag{4}$$

Where $\eta \in (0,1]$ is a parameter called the *quantum efficiency* and b_d is the *dark-count rate* of the photon detector. Now, using our discretized scene parameters ϵ, Δ we can define the following two formulations:

$$\begin{aligned}\hat{\mathbf{x}}_j &= \int_{\epsilon(j-1)}^{\epsilon j} a\delta(t-2d/c)dt \\ \mathbf{S}_{k,j} &= \int_{\Delta(k-1)}^{\Delta k}\int_{\epsilon(j-1)}^{\epsilon j} \eta s(t-y)dtdy \\ B &= \Delta(\eta b + b_d)\end{aligned}$$

Using these formulae, we are closer to writing down a simple distribution parameter for $\mathbf{y}$, which we can then use to estimate the sparse signal $\hat{\mathbf{x}}$ that we are after. $\mathbf{y}$ can now approximately be described by

$$\mathbf{y}_k \sim \text{Poisson}((\mathbf{S}\hat{\mathbf{x}} + B\mathbf{1})_k) \tag{5}$$

This equation looks a lot like the linear measurement-plus-noise equations we've seen in class, with one small difference: rather than $\mathbf{y}$ being a function of a random variable, it is a random variable itself. However, if we note that in reality, $\mathbf{x}$ is a "pure" vector representing our desired data, but is an analog for another random variable (in the sense that we don't know what data $\mathbf{x}$ is going to contain). This simplification of $\mathbf{y}$, then, can equivalently be thought of as a linear function of $\mathbf{x}$.

So, this is a form we recognize. We can use the "1-trick" to get rid of the offset by defining $\mathbf{A} = [\mathbf{S}\ \mathbf{1}]$ and $\mathbf{x} = [\hat{\mathbf{x}}; B]$

$$\mathbf{y}_k \sim \text{Poisson}((\mathbf{A}\mathbf{x})_k) \tag{6}$$

Now that we have a well defined relationship between our measurements $\mathbf{y}$ and the sparse signal we are trying to determine $\mathbf{x}$, we can use a solution method like the ones we've been exploring this semester. Namely, we will define a loss function with a non-convex constraint, and then minimize this loss function using a custom algorithm.

So far, we haven't assumed anything about the correlation between $\left\{(\mathbf{y}_j, \mathbf{y}_k) \;\middle|\; j \neq k\right\}$. This means that we can use this distribution of $\mathbf{y}_k$ to generate the likelihood function of $\mathbf{y}$ as a whole. By taking the

log-likelihood function of $\mathbf{y}$'s PMF we arrive at the following loss function:

$$\mathcal{L}(\mathbf{x}; \mathbf{A}, \mathbf{y}) = \sum_{k=1}^{M} [(\mathbf{A}\mathbf{x})_k - \mathbf{y}_k \log(\mathbf{A}\mathbf{x})_k] \tag{7}$$

We noted earlier that in $\hat{\mathbf{x}}_{N\times 1}$ there can only be one nonzero element. After concatenation with B, $\mathbf{x}$ will now contain two nonzero elements; the position of the first representing the scene depth and the value of the second representing the scene noise and other corrupting factors (dark-count rate, namely). We let $\mathcal{S}_N$ be the union of these N subspaces that $\mathbf{x}$ can possibly belong to, and add this as a constraint in our minimization procedure. Our final problem is

$$\begin{aligned} \underset{x}{\text{argmin}} \quad & \mathcal{L}(\mathbf{x}, \mathbf{A}, \mathbf{y}) \\ \text{subj. to} \quad & \mathbf{x} \succeq \mathbf{0} \\ & \mathbf{x} \in \mathcal{S}_N \quad (*) \end{aligned}$$

Notice that $(*)$ is not a convex constraint. We can see this by simply noticing that a line connecting two $\mathbf{x}$'s that lie in different subspaces in $\mathcal{S}_N$ may pass through an intermediary space with three nonzero elements. Usually, in class, we've found convex relaxations of non-convex constraints and then used the method of lagrange multipliers to solve unconstrained optimization. However, in this case, we'll use a particular search algorithm that enforces membership in $\mathcal{S}_N$ at all intermediate steps.

3 Methods

In the previous section we arrived at a constrained optimization problem with non-convex constraint $(*)$. The benefit of using a convex relaxation of $(*)$ is that we can use Lagrange multipliers to plug an unconstrained optimization problem into existing tools such as those contained in MATLAB or Python. However, we have very specific requirements on our desired $\mathbf{x}$ vector (that it only contain two non-zero elements, one of which is at the end of $\mathbf{x}$), and the benefits of our problem formulation disappear if we cannot precisely enforce this requirement.

3.1 Algorithm

Instead of finding a convex relaxation, we can perform successive steps to enforce our requirement:

1. Perform a round of unconstrained optimization on $\mathcal{L}$.

2. Project the result into the closest subspace within $\mathcal{S}_N$.

3. Zero out all negative elements.

This algorithm is described by the following pseudo-code, written by Shin et al. [1]:

Data: $\mathbf{y}, \mathbf{A}, \delta$
Result: $\mathbf{x}$
initialize $\mathbf{x}^{(0)} \leftarrow \mathbf{0}$;
initialize $\mathbf{u} \leftarrow \mathbf{y}$;
initialize $k \leftarrow 0$;
repeat
  $k \leftarrow k + 1$;
  $\tilde{\mathbf{x}} \leftarrow \mathbf{A}^T\mathbf{u}$;
  $\Omega \leftarrow \text{supp}((\tilde{\mathbf{x}}_{1:N})_{[1]}) \cup \text{supp}(\mathbf{x}^{(k-1)}_{1:N}) \cup N + 1$;
  $\mathbf{b}|_\Omega \leftarrow \mathbf{A}^*_\Omega\mathbf{y}$;
  $\mathbf{b}|_{\Omega^c} \leftarrow \mathbf{0}$;
  $\mathbf{x}^{(k)} \leftarrow \mathcal{T}_0\left([(\mathbf{b}_{1:N})^T_{[1]}, \mathbf{b}_{N+1}]^T\right)$;
  $\mathbf{u} \leftarrow \mathbf{y} - \mathbf{A}\mathbf{x}^{(k)}$;
until $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2 < \delta$;

Algorithm 1: Solving the constrained optimization problem using iterative optimization & projection

In this algorithm, $\mathcal{T}_0$ denotes the projection into the positive subspace, $\mathbf{x}_{[k]}$ denotes the top k elements of $\mathbf{x}$ sorted in descending order, $\mathbf{A}^*$ denotes the pseudo-inverse of matrix $\mathbf{A}$, and $\mathbf{A}_\Omega$ denotes the columns of $\mathbf{A}$ corresponding to the indices specified in Ω.

A bit of observation shows that this algorithm runs by jumping between subspaces in $\mathcal{S}_N$. At every iteration, a proxy $\tilde{\mathbf{x}}$ is calculated. The index of the maximum value in $\tilde{\mathbf{x}}$ is used as the current subspace, and it is compared of the subspace of the previous iteration, represented by $\text{supp}\left(\mathbf{x}^{(k-1)}_{1:N}\right)$. Finally, the new estimated $\mathbf{x}^{(k)}$ is generated from the maximum value taken on between these two subspaces. This is repeated until the simple convergence criterion is satisfied, namely that the L2-norm between consecutive iterations is sufficiently small. The gradient descent of this algorithm is approximated by the squared L2-norm of $\mathbf{y} - \mathbf{A}\mathbf{x}^{(k-1)}$, a technique described in detail by Needell et al. [3] and that is outside the scope of this paper.

3.2 Parallel processing

We can take advantage of our assumption of independence to run the above optimization algorithm for our discrete sections of the scene. The algorithm was tested on a dataset collected by the MIT single photon imaging team [7], and contains a 350×350 matrix of 48 photon arrival times per pixel (these 48 arrival

times, arranged into a histogram, are $\mathbf{y}$ in the problem formulation). Since the derived depth at any pixel is independent, they can be processed in parallel without affecting the output of the results.

The MATLAB parallel computing toolbox [8] will be used to test for possible speed improvements in the derived algorithm. The entire framework would run in two nested `for`-loops: one for rows of the image, and the other for columns. The function `parfor` can be used to test for potential speedups in these loops by launching a parallel pool of workers to run the optimization.

4 Results

The algorithm was implemented in MATLAB [9] with the following results:

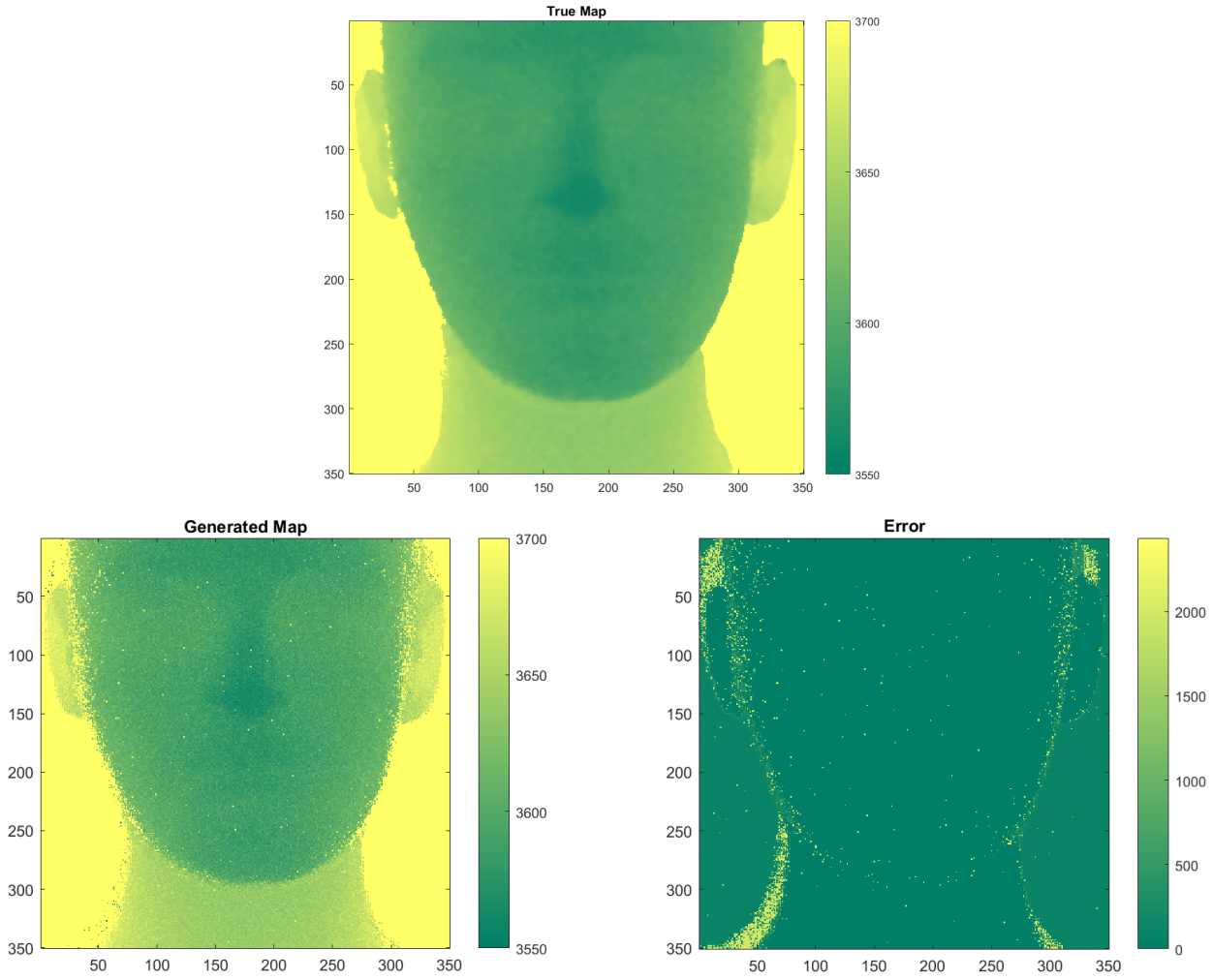

Figure 1: Optimization results.

The optimization loops took a total of approximately 3 hours to run. Despite multiple rounds of optimization, I was unable to match the speed of the sample implementation [5], which runs the optimization loops in approximately 300s. The major operation eating up the running time seems to be the matrix multiplication in the second step of the optimization loop, which accounts for approx. 97% of the total running time.

Several trials were run using different parallel processing techniques on a subset of the data to derive estimates for the total running time of the algorithm. The following trials were run with the outer `for`-loop set to run 10 times, and the inner loop set to run 350 times, corresponding to calculating depth for 10 complete rows of the output image. The column corresponding to "Par. Constant" refers to using a `parallel.pool.Constant` to hold values and reduce communication overhead between workers.

Table 1: Timing trials for different run parameters

Trial	# Workers	Outer par.	Inner par.	Par. constant	Time (s)	Projected (min)
1	1	✗	✗	N/A	333.6	194.6
2	2	✓	✗	✗	321.4	187.5
3	4	✓	✗	✗	212.6	124.0
4	4	✗	✓	✗	190.6	111.2
5	4	✗	✓	✓	195.0	113.7
6	4	✗	✓	✓	194.0	113.2
7	4	✗	✓	✓ (A)	187.9	109.6
8	4	✗	✓	✓ (A & data)	193.4	112.8
9	4	✗	✓	✓ (A & δ)	192.2	112.1
10	4	✗	✓	✓ (A)	188.9	110.2
11	4	✓	✗	✓ (A)	210.1	122.6

Trials 7 & 10 (re-run for validation) produced the best time reductions with my code. Running the inner loop in parallel produced significantly better results than running the outer loop in parallel, as can be seen in the difference in running time between trials 10 & 11. In total, the running time was reduced to 56.4% of the original running time using parallel processing.

I also used parallel processing on the sample implementation [5] to see how running time could be increased. These trials were run on the entire data set.

Table 2: Timing trials for sample implementation

Trial	# Workers	Outer par.	Inner par.	Par. constant	Running Time (s)
1	4	✓	✗	✓(A)	258.6
2	4	✗	✓	✓(A)	206.8
3	N/A	✗	✗	N/A	261.7

The inner parallel loop, again, performed better than the outer parallel loop. The running time was reduced to 79.0% of the original running time achieved by Shin et al.

5 Conclusion

In this paper, we formulated the problem of LIDAR depth sensing as a sparse signal reconstruction. We simplified a closed form of how our measurements relate to the target signal, and defined a loss function so we could formulate a solution as an optimization problem. We then implemented an algorithm designed by Shin et al. [1] to solve a non-convex optimization problem by iterating through steps of unconstrained optimization & projection into a valid subspace that our target data could exist in. The algorithm was run using MATLAB and results similar to the sample results were achieved, albeit with a longer running time. Parallel processing was then used to reduce the running time of our implementation, as well as that of the sample implementation significantly.

The use of parallel processing was fairly effective at reducing the running of the optimization algorithm, thanks to our assumption of depth independence. For future work I think it would be interesting to see how this algorithm could be implemented on a computing cluster to perform the optimization on-the-fly. In a fast-paced scenario, such as on an autonomous vehicle, perhaps this algorithm would be appropriate because it operates with a very small number of photon detections with very good accuracy. Also, given that the pixel depths are assumed independent, it would also make sense to format the input data is a single column vector, like we've done with MNIST, and parallelize the operation along a single axis. This method would remove the tradeoff between parallelizing the outer and inner loops in the implementation, and perhaps could reduce the running time further.

References

[1] Dongeek Shin, Jeffrey H. Shapiro, Vivek K Goyal, "Single-Photon Depth Imaging Using a Union-of-Subspaces Model", in IEEE Sig. Proc. Letters, vol. 22, no. 12, pp. 2254-2258, Dec. 2015.
https://arxiv.org/pdf/1507.06985.pdf

[2] Github repository for photon-efficient imaging sample data.
https://github.com/photon-efficient-imaging/sample-data

[3] D. Needell, J. A. Tropp, "CoSaMP: Iterative signal recovery from incomplete and inaccurate samples", in Appl. Comput. Harmon. Anal., Vol. 26, pp. 301-321, 2008.
https://arxiv.org/pdf/0803.2392v2.pdf

[4] LV-MaxSonar®-EZ™ Series High Performance Sonar Range Finder datasheet.
http://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf

[5] MATLAB Implementation of single-photon optimization solver
https://github.com/photon-efficient-imaging/uos-imaging/

[6] Larry Hardesty, "3-D images, with only one photon per pixel", MIT News
http://news.mit.edu/2013/3-d-images-with-one-photon-per-pixel-1128

[7] Github user for MIT Single-Photon Imaging Team
https://github.com/photon-efficient-imaging

[8] MATLAB parallel computing toolbox
https://www.mathworks.com/products/parallel-computing.html

[9] Implementation of LIDAR single-photon depth sensing algorithm
https://github.com/buoyad/DepthSensing