# Predicting Atom Location Using Machine Learning Algorithms

Adam Martini, Ran Tian, Wes Erickson

Computer and Information Science Department
University of Oregon
Eugene, Oregon 97403, USA
martini@cs.uoregon.edu
jmty0083@cs.uoregon.edu
wwe3@uoregon.edu

**Abstract.** In this project, we develop the Steck Lab focusing on Magneto Optical Trop (MOT). It simulates the ray generations, classify and predicts the probability distribution. The parallel computing is also included in every steps of Steck Lab to significantly improve the performance. The implementation includes Cilk, MPI and parallelism scripts of qsub. We also use machine learning technique to operate the classification and prediction. From this project, we also obtain remarkable experience of parallel computing techniques.

**Keywords:** machine learning, logistic regression, parallel computing, Magneto Optical Trop

## 1 Introduction

The Steck Lab is investigating the motion of a single atom in a Magneto Optical Trap (MOT). The atom is imaged using a pair of aspheric lenses and a CCD camera. Given an 2D image from the CCD, the lab wants to predict the probability distribution of the atom's location within the MOT. The lab consists of several individual parts including preprocessing, ray tracer generator and classifier. The subgoal of this lab is to distribute and integrate the individual parts.

This report describes our efforts to both simulate atom images, and to create a scalable probabilistic classifier capable of atom location prediction. Section 2 provides an overview of our system design and objectives. We discuss the details of our development and implementation efforts in Section 3. Experimental results are outlined in Section 4 followed by an discussion of the lessons learned in Section 5. We conclude the report in Section 6.

## 2 Design and Objectives

To tackle this problem we run simulations of the CCD output for a given atom location using a raytracing simulation. The output images are labeled according to atom location and used as a training set for machine learning algorithms. The algorithms can then be used to generate a probabilistic distribution for the atom's location for a new CCD image.

Figure 1 describes the data flow of our architecture. We begin by simulating the training and testing data using raytracing and filtering. The filters will apply both generate several different training sets from a single image set. The classification results for each training set are described below. Some of the instances are selected from the image set for noise filtering. The *noisy* data is used to test our classification algorithms.

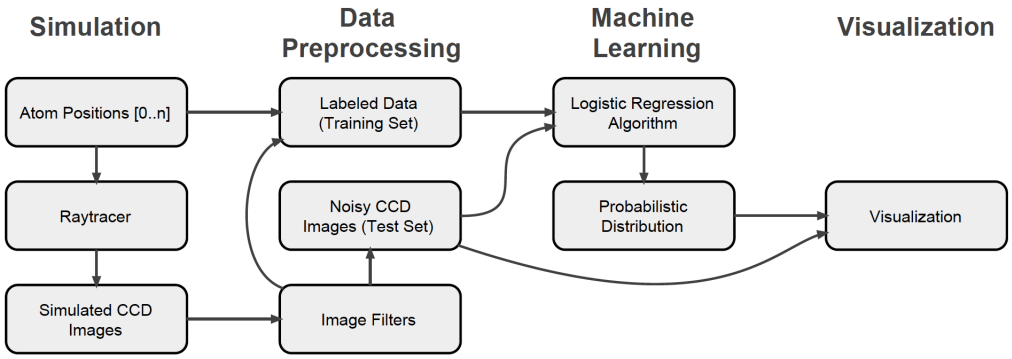



**Fig. 1:** High Level Data Flow Architecture Diagram

The objective of our project is to create and test a scalable machine learning solution for atom location prediction using CCD images. In order to scale our application we use both shared memory and distributed memory parallelism. Massive amounts of training data are generated using a shared memory parallelism, and a distributed memory logistic regression classifier is used to efficiently learn interesting patterns from the data. The remainder of this report describes our development efforts, experimentation results, and the lessons we have learned during the process.

## 3 Development and Implementation

Our development efforts are divided into sections based on our data flow architecture. There is some overlap of effort in the data preprocessing step as this is the area of our development where Simulation and Machine Learning meet. We describe each development stage in detail in the following subsections.

### 3.1 Simulation

The machine learning algorithm we implemented requires training for it to accurately predict an atom location when given just the CCD image. Thus, we first need to generate a large training set consisting of pairs of atom locations with CCD images. This is done by simulating CCD images for the atom located at different points.

The essence of the simulation is starts with a flourescent atom at a chosen location, and then simulating how light emitted from the atom is refracted by the lens system and appears on the CCD screen. We opted for a classical raytracing approach. Rays are emitted in a random direction from the atom, and obey refraction via Snell's Law as they pass interfaces between different materials. When the rays intersect with the CCD plane, we increment an array value corresponding to which CCD pixel it hit.

**Raytracer Development** A custom barebones raytracer was designed from the ground up in C, the key design goals being that it should be simplistic, fast, and readily parallelizable. This was done since the features of most raytracing systems are unnessesary for our application, in part because:

- We only need first order refracted rays. No higher order or reflected rays are needed.
- We apriori know the order in which the rays will intersect surfaces.
- Elements of graphical raytracing systems are more extravagant than the simple intensity plots we need.

Some important features that were implemented in the raytracer:

- Standard vector operations (projections, cross products, ray extrapolations, etc.)
- Structure for defining arbitrary surfaces and refractive indicies.
- Ray/Surface intersection routine.
- Calculation of surface normals.
- Implementation of Snell's law.
- Ray refraction routine.
- CCD image screen with histogram binning.

Each of these features were tested independantally by plotting ray traces with gnuplot. This was done in preparation for the more complex system with custom optics, which is harder to verify.

**Imaging System** An imaging system consisting of a pair of custom aspheric lenses was then implemented with our raytracer surfaces. The system consists of five surfaces in all. The first two surfaces form the initial aspheric lens, the second two surfaces form the second aspheric lens, and the final surface forms the CCD image plane.

For completeness, the spacing between the lenses is 90mm, the focal point is $5mm$ from the lens, and lastly, the asphere lens profiles are given by

$$Z(r) = \frac{Cr^2}{1 + \sqrt{1 - (1+k)C^2r^2}} + \sum_{i=0}^{10} D_{2i} r^{2i} \tag{1}$$


**Fig. 2:** Scale model of the atom imaging system.

with the constants

| Parameter | Value |
|---|---|
| $C$ | $-0.128$ |
| $k$ | $-0.824$ |
| $D_0$ | 17.705 |
| $D_2$ | $2.11 * 10^{-2}$ |
| $D_4$ | $8.71 * 10^{-2}$ |

(2)

### 3.2 Data Preprocessing

The original picture is designed to be 100 times 100 pixels matrix in each slice. The data preprocessing build a noise maker and filtering mechanism before the slices of matrix are actually applied to the classifier. It actually simulates a half transparent mask before the camera practically. The image generated by such mask adds Poisson distributed random number to every pixels of the image. A Poisson distribution stands for discrete probability that can give a number of events occurring in a fixed interval of time or space if events are independent with the average of $\lambda$. The value of $\lambda$ is switched between different proposed number in real test. Filtering, on the other hand, is clearing the influence caused by the noises. The filtering mechanism generally follows the work in course lab 8. A Gaussian distribution based figure blur decreases the degree of distribution caused by Poisson distribution based noises. Then, the Prewitt gradient filters all unnecessary noise with only structure, which contains the lines of rays, left.

The feature scaling step is handled in the logistic regression classifier as it is a simple extension to data loading after the instance have been loaded into memory. We use efficient a C++ template library for linear algebra [1] that provides efficient matrix operations that can quickly perform the broadcasting operations necessary for feature scaling. In addition, the distributed implementation of our algorithm lends itself well to efficient data partitioning, which ensures that this preprocessing step will scale. However, feature scaling in a distributed setting is slightly more complex in that it requires knowledge of the global min and max for the entire

dataset across all nodes. The min/max vectors is quickly computed using an Allreduce step with MPI's MIN/MAX reduction operators. Each node then formats its section of the data using a single call feature scaling utility function.

### 3.3 Machine Learning

We have developed a distributed logistic regression classifier using MPI to train on massive amounts of data efficient. The classifier is capable of multi-class classification and mini-batch processing. During training, nodes communicate with each other to both build the model and perform parameter updates.

The model building step includes data loading, feature scaling, and label formating. During data loading, each node loads a portion of the instances the provided data directory based on its task ID and the total number of tasks in the communication world. It is important to note that the instance file names are first loaded into a vector and then randomized to ensure that mini-batch processing have a consistent sample of the data with each iteration. After the data is loaded, feature scaling is performed as described above. The label formating step uses a custom MPI operator in an Allreduce call to generate global unique label set. The unique label set is used at each node to generate a label matrix from the label vector for 1-vs-all classification.

The model training involves iterative updates to a set of global parameters. During each update, the nodes compute a portion gradient update using a subset of the data they are responsible for based on the batch size. The collective gradient update is computed using an Allreduce call to sum each contribution followed by a normalization step on each node based on the total number of instances contributing to the update. The new parameter set (identical on all nodes) is then tested for convergence and used for the next iteration.

Training is finished when either the gradient has converged or the maximum number of iterations has been reached. The Master node saves the trained model parameters to the a file to be loaded on a single node for testing. The testing program generates a set of class membership probabilities for each instance in the given data directory. The chosen class is simply the maximum probability found in the probabilities set for each instance. We also use the probabilities to generate the visualizations of the models *beliefs* about an atoms location for each instance.

### 3.4 Visualization

Visualization is particularly important for visualizing three dimensional volumes, and it may become essential for feedback as to whether our simulations are functioning properly, and for our presenting our project results. However, our project focus is not on programming the visualization methods ourselves, but using existing tools such as Visit or ParaView.

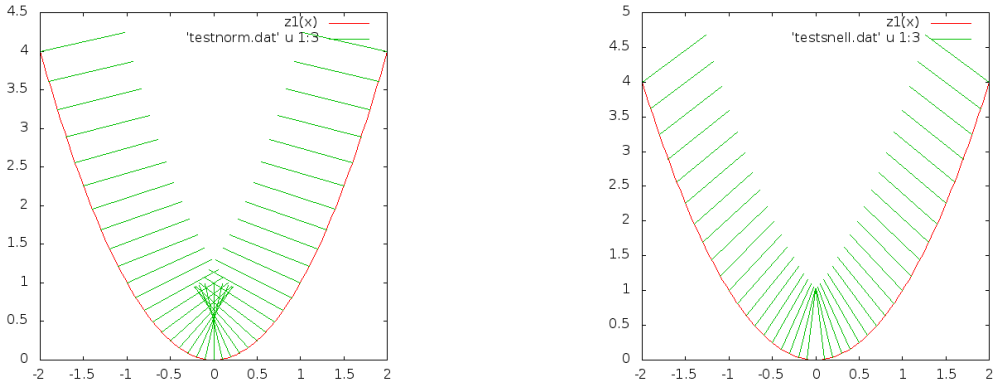
The main items to be visualized are the rays traced through the imaging system, the CCD images, and finally a prediction of the atom location given a CCD image. We can also use these

tools for post processing on the image data to highlight features for the benefit of the machine learning algorithm.

## 4 Experiments and Performance

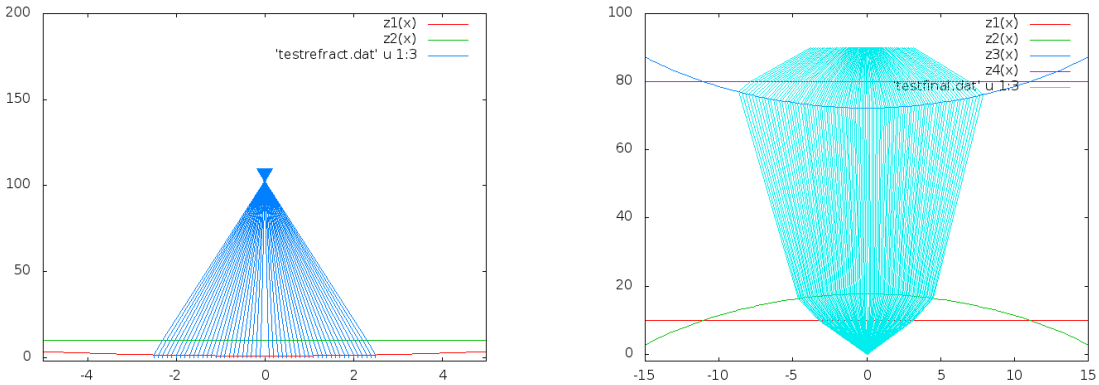
### 4.1 Raytracing

Initial tests of the raytracer.




**(a)** A norm **(b)** A snell

**Fig. 3:** A figure with two subfigures




**(a)** A refract **(b)** A lenstest

**Fig. 4:** A figure with two subfigures

### 4.2 Training and Regression

We had limited time to perform experiments due to the late arrival of our dataset. However, the tests that we did perform clearly show that our distributed logistic classifier scales well

to achieve significant performance gains. In addition to provable performance gains, we also observed Gustafson's Law in action in that the available parallelism significantly increases with our problem size.

For our initial tests, we ran two training sets through the classifier. The first training set was small (283M) and the second set was larger ( 2.9G). Our first goal was to observe the classification accuracy on of the classifier on the testing set. Using the small training set, we were easily able to achieve 100% accuracy. This indicates that the amount of noise was insufficient and/or that there were not enough rays sampled to adequately obscure label partition boundaries.

Figures 5 6 show performance gains for our initial tests. We clearly see that while available parallelism levels out around 10 nodes with a student account on ACISS, the amount of available parallelism dramatically increases with our problem size. The small dataset achieves a speedup of 7 times relative to the serial execution while the larger dataset achieves 9 times speed up. This is an embarrassingly parallel problem with limited communication demands, so it is reassuring to see almost 10 times speedup with 10 times the computational power. This implies that our implementation efficiently utilizes the parallel processing power to take advantage of available parallelism.

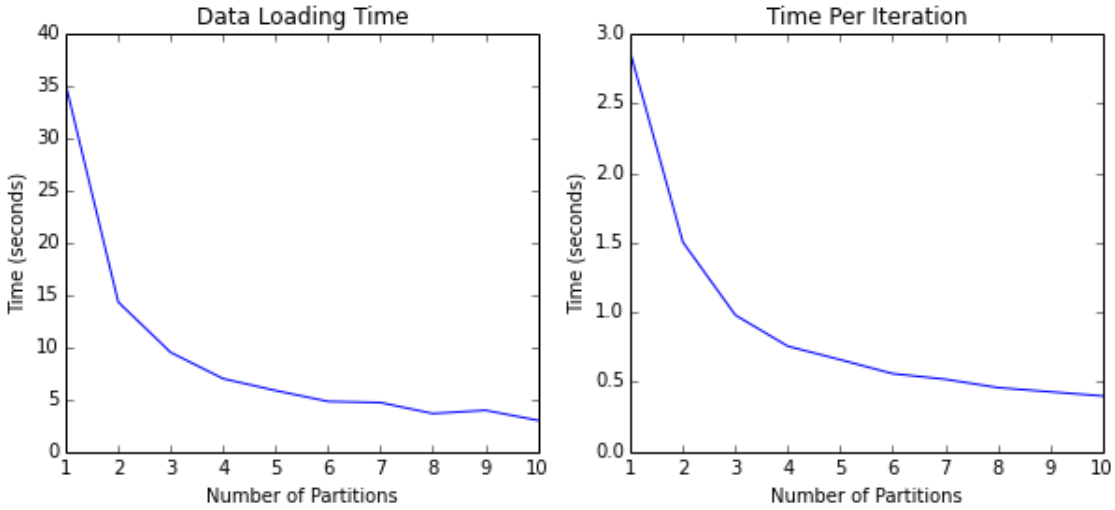



**Fig. 5:** Performance gains for both data loading and iteration time on the small training set.

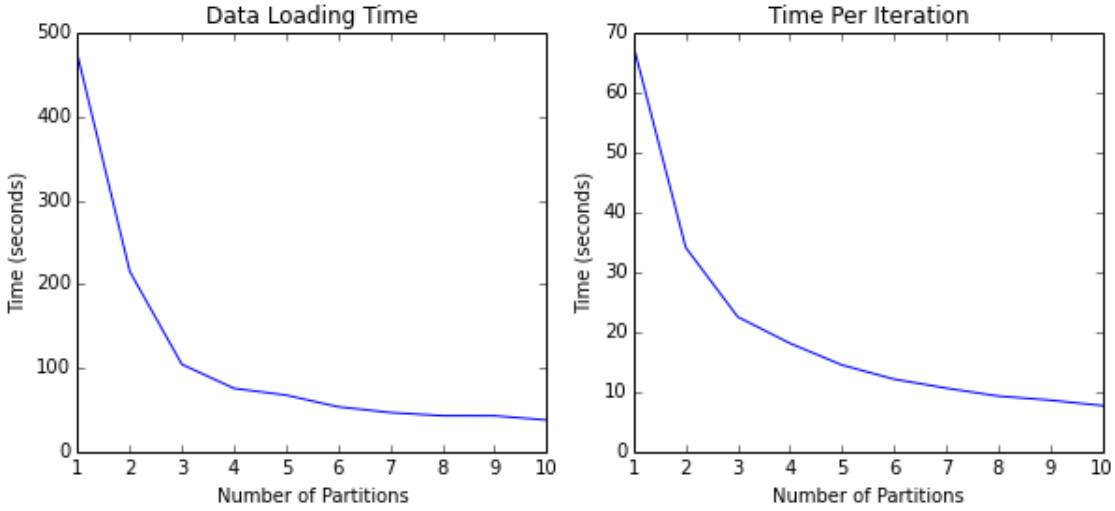



**Fig. 6:** Performance gains for both data loading and iteration time on the large training set.

## 5 Lessons Learned

### 5.1 Simulation

### 5.2 Data Preprocessing

Sharing the idea and details of the implementation of Poisson distribution based noise maker is quite a job. Due to lack of base support of Poisson distribution random number generator, a float number and the range of the floating number is quite interesting topic. The final result turns out to be a integer generator with a scaling shifting. Hence, we can satisfy our purpose and simplify the implementation at same time.

To accomplish distributed feature scaling and label formating we need to need to create a custom MPI reduce operation that is capable of merging two sets of labels into a single unique set. We learned that creating custom reducers can be the source of many bugs related to incorrect data pointers. This was also the case when created custom reducers for gradient updates, which we discuss below.

### 5.3 Machine Learning

The origin plan includes another distributed memory system based classifier, which is distributed Support Vector Machine (SVM). The distributed SVM is a backup as a classifier for training and testing the ray images. However, after three weeks of work, the result turns out to be unacceptable. The implementation of distributed SVM contains a large number of interfaces that allows user to train and test the result. The only function our project needs is a regression mode for the 100 times 100 pixels image classifier. However, the regression mode does not support

such simple functions after parsing the SVM Light files. The implementation complexes this part into more detailed partial differential equation classifier. Therefore, the decision between continuing the work to implement a corresponding function and stop to switch focusing on other object becomes quite controversial. Due to time limit and tight schedule of my last three weeks. Lacking of machine learning knowledge, we spend too much time studying on basic concept of SVM. The lesson we learned is to better schedule the timeline and estimate the workload for a totally new area.

There were several opportunities for learning during development of the distributed logistic regression classifier. As mentioned above, there were many bugs surrounding custom reduce functions and different implementations of Allreduce on different systems. The was one case where reduce was working correctly on ix-trusty and not on ACISS. However, these issued where solved by replacing custom functions, which used linear algebra library operations, with standard MPI reducers. Therefor, we learned that it is best to use standard MPI functions for operations whenever possible to enable cross-platform compatibility.

Another important lesson came from designing the classifier to be flexible and scalable so that it can easily be applied to various size systems and datasets. The first challenge was deciding how to partition the data based on task ID and the number of computation nodes. Our initial design read from separate data folders such that the user was required to preprocess the data based on how they want the computation to be divided. However, we quickly realized that the division of data should be handled by the classifier at runtime. We accomplished this by allowing the user to place all data into a single folder and the loading the data on each node according to task ID. This method assumes that all data instance are roughly equal in size, which is a safe assumption given the homogeneous nature of a classification problem.

One interesting lesson we learned about distributed classification is that data must be extremely large for mini-batching to apply. Distributed classification is already *batching* the data across nodes, and mini-batching further divides these batches. Therefore, the dataset must significantly exceed each nodes' computational capacity before mini-batching will be necessary.

### 5.4 Visualization

## 6 Conclusions

Text...

## References

1. Eigen C++ Template Library for Linerar Algebra.
`http://eigen.tuxfamily.org/index.php?title=Main_Page`.