Using Julia for Place Recognition

6.338 – Parallel Computing Final Project

Timmy Galvin
Massachusetts Institute of Technology
Department of EECS
Cambridge, MA, USA
tgalvin@mit.edu

Abstract

Place recognition is an important aspect of navigation, especially in GPS-restricted environments. Internal measurement systems and vision-aided navigation are suitable replacements for GPS in short time spans, but internal error or drift soon starts to accumulate and will destroy estimations unless a new corrective measure can be taken. Place recognition allows for a visually-driven navigation system to recognize a previous location it has visited and to update its internal model with that observation using loop closure. For an autonomous robot, this technique truly becomes effective when it begins to reach real-time operation. However, costly algorithms slow down the approach and reliance on Matlab as a simulation technology provide a distorted view of algorithm speed. This paper proposes switching to Julia for this problem as it encapsulates both speed and usability and also modifying some of the more expensive exact calculations to faster approximations in Julia.

I. Introduction

Previous work has showed that using place recognition can eliminate instrument drift when using inertial measurements in a GPS-restricted environment [1]. However, due to the size of the data sets being processed and the computational complexity of the algorithms, efforts to achieve a real-time solution have not been entirely successful. Two of the issues impeding this progress are addressed in this paper. First, the use of overly accurate algorithms with high computational complexity and slow execution times create a bottleneck for the discussed place recognition approach. Second, the use of Matlab as the language to profile different approaches' implementations leads to a perceivable skew in which are pursued further as Matlab tends to favor those algorithms that can be vectorized.

The remainder of this paper will be structured to first provide an overview of the place recognition technique that will be analyzed and optimized in Julia. Following that, I will present my modifications to the algorithm using Julia and non-black box methods. The results section will summarize the differences between the new approach in Julia and the old Matlab implementation. Finally I will discuss some the results and some pending challenges that must be addressed before Julia can begin to replace Matlab.

II. Methods

Place Recognition

Place recognition is determining whether an image of location matches a previously taken from the same location. However, a direct pixel-by-pixel comparison is insufficient as the approach must be able to handle images taken at slightly different locations, different angles, and at different times (leading to different lighting, slight changes, etc.). To compare the current image, it is vital to convert it from a pixel representation to a feature representation. Using a previously calculated vocabulary or set of features, the image can be transformed into a scene descriptor, a one-dimensional array with entries that represent whether that corresponding feature in the feature vocabulary is present or absent in the image using algorithms such as SURF in [2] and as show in Figure 1. To find how similar two individual images after this transformation, we only need to calculate the dot product between the two scene descriptors. However, this single-scene comparison performs poorly when applied to real-world data because there are often similar looking scenes that one may see (e.g., repetitive sides of a building or trees and shrubbery).

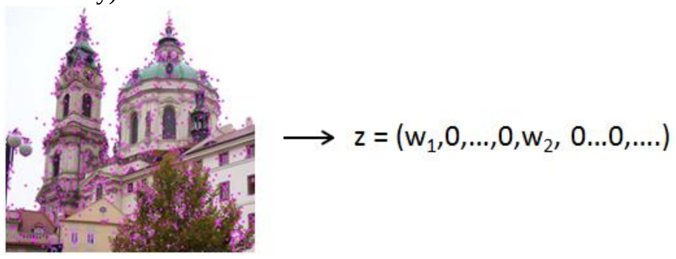
$\longrightarrow z = (w_1, 0, \ldots, 0, w_2, 0 \ldots 0, \ldots.)$

Figure 1: Generating a scene descriptor using a previously generated feature vocabulary.

To improve over the single-scene comparison approach, we can instead look at sequences of scenes. We construct a similarity matrix that represents each combination of normalized dot product as show in Figure 2. With this representation, a series of scenes that are highly similar will show itself as an off-diagonal trace or a consecutive series of high value elements on an off-diagonal. These off-diagonal traces can be found by applying a local sequence finding algorithm. We take advantage of previous work done on this problem by biologists doing DNA comparisons and use a modified version of their Smith-Waterman algorithm [3].

$$S(i,j) = \frac{\sum_{i=1}^{|\nu|} z_i z_j}{\sqrt{\sum_{i=1}^{|\nu|} z_i^2}\sqrt{\sum_{i=1}^{|\nu|} z_j^2}} \qquad \begin{aligned} z_i &= [0\ 1\ 0\ 0\ 1\ 1\ 0\ \ldots] \\ z_j &= [0\ 0\ 1\ 0\ 0\ 1\ 1\ \ldots] \end{aligned}$$

Figure 2: Constructing a similarity matrix from scene descriptors.

Searching for local sequences in the unmodified similarity matrix leads to a large number of false positives. These bad results are due to dominant features among sets of images. One can think of these dominant features as commonly occurring elements in some environmental subset (e.g., building features, trees, and roads). This observation leads us to want to remove the dominant features of our images, and therefore also our similarity matrix.

We can remove the dominant features of our similarity matrix by removing the most significant singular values and their corresponding singular vectors. Previous work has leveraged the rapid fall-off of singular values (Figure 3) and removed *r* singular values to maximize the entropy of the remaining singular values as shown in Equations 1a, b, c, and d. The combination of this reduction and the application of the modified Smith-Waterman algorithm allow us to identify places that have previously visited.

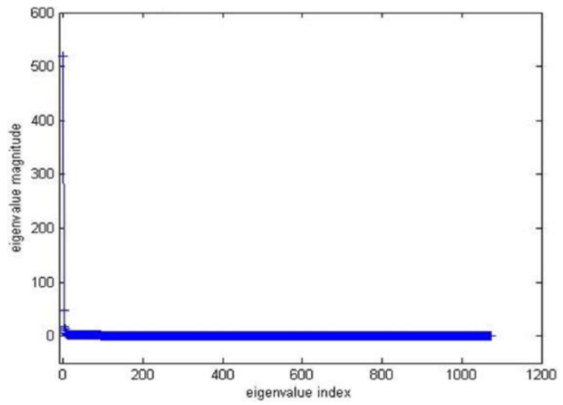

Figure 3: Singular value drop-off for a standard similarity matrix.

$$S' = \sum_{i=r^*}^{n} u_i \lambda_i v_i^T \qquad r^* = \arg\max_r H(M,r)$$

$$H(M,r) = \frac{-1}{\log(n)} \sum_{k=r}^{n} \rho(k,r) \log(\rho(k,r)) \qquad \rho(i,r) = \frac{\lambda_i}{\sum_{k=r}^{n} \lambda_k}$$

Equations 1a,b,c,d: Reducing the similarity matrix by maximizing singular value entropy.

While this full approach yields accurate results, both the full singular value decomposition and the Smith-Waterman algorithm parts of it are too slow to allow for real-time deployment. This is where this paper attempts to make improvements.

Proposed Improvements with Julia

I chose to work on improving this place recognition algorithm in Julia as it would allow for better general performance than the same implementation in Matlab, it would remove the dependence on black-box libraries, and it would maintain its general usability by the engineering community that has developed skills with Matlab over the years. As mentioned above, I implemented the algorithm in Julia to mainly focus on the bottlenecks of the singular value decomposition and the Smith-Waterman local sequencing algorithm.

Partial Decomposition Algorithm

As seen in Figure 4, the reduced version of the similarity matrix only depends on the *k* least significant singular values and corresponding singular vectors. While this fact might immediately lead one to think that a full and accurate decomposition is necessary, this relationship can be trivially restated as a function of only the *n-k* most significant singular values and related singular vectors, as shown in Equation 2. This simple rephrasing of the reduction opens the door to the use of a partial, or approximate, singular value decomposition.

$$A' = A - \sum_{i=1}^{r^*-1} v_i \lambda_i v_i^T$$

Equation 2: Rephrasing of reduction to require only approximate decomposition.

While both ARPACK, PROPACK, and other libraries have implementations of approximate singular value decompositions (SVDS) available (Figure 4), a goal of this project, and of some Julia contributors, is to remove some of that black-box algorithm dependency. So in that effort, I put together a Golub-Kahan-Lanczos Bidiagonalization algorithm in Julia (the same approach used in PROPACK) to calculate the approximate singular value decomposition. The Golub-Kahan-Lanczos technique is an iterative solver that constructs a bidiagonal matrix with the same singular values as the original matrix as seen in Equation 3. The decomposition of the bidiagonal matrix can be computed quickly using the QR decomposition [4].

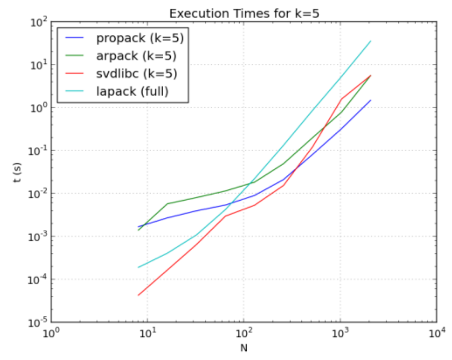

Figure 4: Execution times of various library implementations of approximate singular value decompositions.

$$B_n = P_n^* A Q_n = \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ & \alpha_2 & \beta_2 & & & \\ & & \alpha_3 & \beta_3 & & \\ & & & \ddots & \ddots & \\ & & & & \alpha_{n-1} & \beta_{n-1} \\ & & & & & \alpha_n \end{bmatrix}$$

Equation 3: Bidiagonalization with same approximate singular (Ritz) values as *A*.

An issue with the Golub-Kahan-Lanczos method is that through its iterations, the left and right Lanczos vectors lose their mutual orthoginality. This behavior requires special attention as it will cause the calculation of repeated Ritz values (the approximated singular values). In modifying the simple Golub-Kahan-Lanczos algorithm, I investigated multiple techniques: full orthoginalization, partial orthoginalization, and restarting. While restarting is the approach often implemented in libraries that use the Golub-Kahan-Lanczos approach, I investigated the different solutions for this specific place recognition problem. For matrices of the form used in place recognition, Figure 5 shows the performance of my implementations of the approximate SVD algorithms on sample similarity matrices.

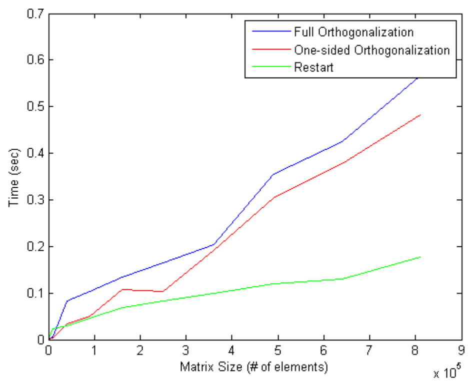

Figure 5: Performance of different GKL implementations.

The main method for the GKL approach with thick-restart is shown below in Code 1.

```
function
gkl_enhanced{T<:BlasFloat}(A::AbstractMatr
ix{T})
  n = size(A, 2)
  a, b = Array(T, n), Array(T, n-1)
  K = KrylovSubspace(A, 1)
  initrand!(K)
  p = Array(T, n, n)
  p[:,1], q[:,1] = nextvec(K), lastvec(K)
  stored_rho = Array(T,n)
  q[:,1] = norm(q)
  L = 0
  #restart loop
  while true
    if exit
      break
    end
    p[:,L+1] = A * q[:,L=1]
    for i=1:L
      p[:,L+1] = p[:,L+1] -
b[i,L+1]*p[:,i]
    end
    for j=L:k
      q[:,j+1] = conj(A)*p[:,j]
      c = q[:,1:j]*q[:,j+1]
      rho = norm(q[:,j+q])
      a[j] = norm(p[:,j]
      p[:,j] = p[:,j]/a[j]
      q[:,j+1] = q[:,j+1]/a[j]
      c = c / a[j]
      rho = rho / a[j]
      q[:,j+1] = q[:,j+1] - q[:,1:j]*c
      b[j] = sqrt(rho^2 - sum(c.^2))
      if b[j] < nu * rho
        c = conj(q[:,1:j])*q[:,j+1)
        rho = norm(q[:,j+1)
        q[:,j+1] = q[:,j+1] - q[:,1:j]*c
        b[j] = sqrt(rho^2 - sum(c.^2))
      end
      q[:,j+1] = q[:,j+1] / b[j]
      if j < k
        p[:,j+1] = A*q[:,j+1]-b[j]*p[:,h]
      end
    end
    B = Bidiagonal(a, b, true);
    B[1:L,L+1] = saved_norms[1:L]
    [X,S,Y] = svd()
    norms = Array(T,n)
    for i=1:k
      norms[i] = b[k]*conj(e[k])*x[i]
      if evaluate_norms(norms)
        exit = true
    end
    L = update_L(L)
    ritz_q = q[:,1:k]*Y[:,1:L]
    ritz_p = p[:,1:k]*X[:,1:L]
    stored_rho[1:L] = norms[1:L]
  end
  [p[:,1:k]*X[:,1:L],S,[q[:,1:k]*Y[:,1:L]]
end
```

Code 1: Main method for GKL with thick-restarting.

Smith-Waterman Algorithm

The Smith-Waterman algorithm, unlike other sequencing algorithms, looks at local sequences instead of the total sequence. As it is a dynamic programming algorithm, it does not perform well in Matlab as it relies on either nested for loops or recursion. The algorithm generates a matrix H in manner show in Equation 4 [6].

$$H_{i,j} = \begin{cases} H_{i-1,j-1} + M'_{i,j} & \text{if } H_{i,j} \text{ is maximal} \\ H_{i,j-1} + M'_{i,j} - \delta & \text{if } H_{i,j-1} \text{ is maximal} \\ H_{i-1,j} + M'_{i,j} - \delta & \text{if } H_{i-1,j} \text{ is maximal} \\ 0 & \text{if } M'_{i,j} < 0 \end{cases}$$

Equation 4: Smith-Waterman algorithm for the creation of *H*.

I chose to re-implement the Smith-Waterman algorithm in Julia to achieve a better comparison to a deployable C implementation while still leaving the code in a singular form that is easily understood and modifiable. It would also make another algorithm available to multiple disciplines as this local trace algorithm spans from biology to vision-aided navigation. The comparison of the Matlab and Julia implementations are shown in Figure 7. It can be seen that Julia performed significantly better than Matlab for all dimensions of the reduced similarity matrix.

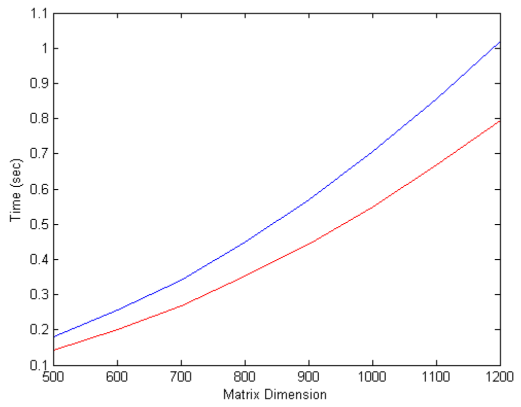

Figure 7: Run-time of Matlab (blue) and Julia (red) Smith-Waterman implementations.

III. Discussion

I chose to re-implement some previous work in place recognition algorithms in Julia to show off what Julia has to offer to professional engineering. It allows one to profile realistic deployment run-times without requiring the maintenance of multiple sources (Matlab and C). It also has very high usability and is a quick language to learn for someone who has experience in Matlab.

Members of the Julia community are working on an IterativeSolvers package–now just in its fledgling stages–in an effort to move away from the black-box nature of the standard linear algebra libraries. The goal would be to create readable and organic methods and remove the reliance on ARPACK. My work showed me just how unreadable those libraries are and also that revealing the implementation exposes a great deal of fine-tuning to the user.

While I was unable to fully match the performance of the ARPACK implementation of approximate singular value decomposition, I still believe this work was a success. In converting this work into Julia and slightly modifying the algorithms used (full decomposition to partial), I was able to achieve notable speedups towards the goal of real-time deployment. Figure 8 shows the difference between Matlab and Julia implementations of each stage of the place recognition algorithm. It is important to note that the Matlab implementation includes a full decomposition instead of the approximate decomposition utilized in the Julia version.

Step	Runtime of Matlab Code	Runtime of Julia Code
Decomposition	0.61	0.22
Maximize entropy	0.0024	0.0024
Reduce matrix	0.011	0.013
Calculate S-W	0.40	0.31
Total	1.02	0.54

Figure 8: Runtime of original Matlab code versus Julia implementation.

I found that this work reinforced my belief that engineers should begin to use Julia over Matlab, especially for some of their simpler implementations. Julia definitely has significant ground to cover when compared to the number of toolboxes that Matlab supports (at cost). And Matlab is easier to install and to perform some operations like plotting in. However, the recent work in increasing collaborative documentation for Julia and in creating a Matlab-to-Julia converter will hopefully convert more engineers and programmers into avid Julia users.

IV. Conclusions

In this paper I argue the use of Julia in comparison to Matlab for profiling purposes. To support my claim, I converted a place recognition algorithm into Julia and in doing so, wrote various implementations of the Golub-Kahan-Lanzcos Bidiagonalization algorithm and a Julia version of the local sequencing Smith-Waterman algorithm. I showed that speed improvements were achieved through this change of language and slight algorithmic modification. Finally, I discussed some recent work that should further encourage the use of Julia in the engineering community.

REFERENCES

[1] R. Madison, *et al.*, "Soldier Affixed, Vision Aided Navigation Technology (SAVANT): Part 1," in *ION Joint Navigation Conference*, Colorado Springs, Colorado, 2011.

[2] H. Bay, *et al.*, "SURF : Speeded Up Robust Features," in *Computer Vision and Image Understanding (CVIU)*, 2008, pp. 346-359.

[3] K. L. Ho and P. Newman, "Detecting loop closure with scene sequences," *International Journal of Computer Vision,* vol. 74, pp. 261-286, 2007.

[4] V. Hernandez, *et al.* "Restarted Lanczos Bidiagonalization for the SVD in SLEPc." SLEPc Technical Report STR-8. 2007.

[5] L. Hamilton, *et al.* " Nav-by-Search: Exploiting Geo-referenced Image Databases for Absolute Position Updates." ION GNSS+ 2013, 2013.