User-Driven Geolocation of Untagged Desert Imagery Using Digital Elevation Models

Eric Tzeng, Andrew Zhai, Matthew Clements, Raphael Townshend, and Avideh Zakhor

Abstract We propose a system for user-aided visual localization of desert imagery without the use of any metadata such as GPS readings, camera focal length, or field-of-view. The system makes use only of publicly available datasets—in particular, digital elevation models (DEMs)—to rapidly and accurately locate photographs in non-urban environments such as deserts. Our system generates synthetic skyline views from a DEM and extracts stable concavity-based features from these skylines to form a database. To localize queries, a user manually traces the skyline on an input photograph. The skyline is automatically refined based on this estimate, and the same concavity-based features are extracted. We then apply a variety of geometrically constrained matching techniques to efficiently and accurately match the query skyline to a database skyline, thereby localizing the query image. We evaluate our system using a test set of 44 ground-truthed images over a $10{,}000\,\mathrm{km}^2$ region of interest in a desert and show that in many cases, queries can be localized with precision as fine as $100\,\mathrm{m}^2$.

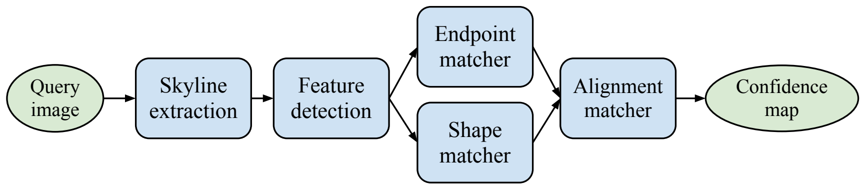

Fig. 1 Block diagram for the query localization system.

1 Introduction

Automatic geolocation of imagery has many exciting use cases. For example, such a tool could semantically organize large photo collections by automatically adding location information. Additionally, real-time solutions would serve as an alternative method of localization in instances where GPS systems are typically unreliable, such as urban canyons.

Researchers have attempted to solve this localization problem on the global scale. The authors of [1] use large image databases to identify probable query locations based on image properties such as color and texture. Additionally, [2] shows that using multiple queries with temporal information can improve results. However, because of their reliance on existing imagery, such systems are suited to situations in which ground-level imagery is abundant in the region of interest.

Most of the previous work achieving localization precision on the order of meters has focused on urban environments, relying on distinctive man-made landmarks. In particular, researchers have had great success using standard feature descriptors such as SIFT and street-view databases to localize imagery in cities [3–5]. These approaches typically detect salient keypoints using a feature descriptor of their choosing, then use a bag-of-visual-words matching scheme to retrieve a corresponding view from a database of street-level imagery.

In this paper however, we are interested in localizing of imagery in natural environments such as deserts. Techniques that work well in urban environments fail to produce usable results in these settings, since they lack the abundance of discriminative keypoints that is characteristic of urban settings. Additionally, even if typical descriptors did work well, available ground-based imagery datasets are too sparse in these environments to result in reliable ground-to-ground image matching.

The recent work by Baatz et al. [6] on geolocalization of images in mountainous terrain is perhaps the only prior work on large-scale localization in natural environments. In their work, they assume a known camera field-of-view for a given query image. However, in many practical scenarios, there is no knowledge of camera parameters or additional image metadata whatsoever. There has also been much work in the context of robot localization [7, 8]. However, in addition to making use of prior knowledge of camera parameters, these systems generally operate on a much smaller scale.

In this paper we propose a system to solve the large-scale localization problem in desert terrain by building upon existing works while overcoming many of their limitations. Similar to Baatz et al. [6], we focus on the boundary between land and sky as our main source of discriminative information. However, inspired by the previous work of Lamdan et al. [9], we use a different feature descriptor based on concavities in the skyline. These features have stable endpoints even when scales and in-plane rotations are applied, allowing us to operate in cases where camera parameters such as the field-of-view are unknown.

Armed with this feature descriptor, we use a digital elevation model (DEM) to synthesize skylines at a regular sampling grid within our region of interest, then build a database out of these skylines and their detected features. Our choice to use

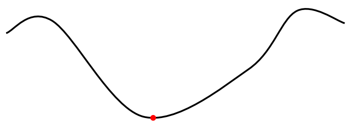

Fig. 2a The input curve to feature extraction and the detected point of extreme curvature.

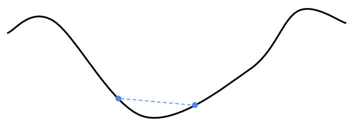

Fig. 2b Initial estimates of endpoint locations.

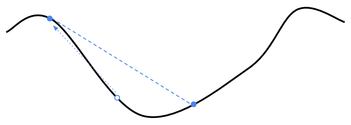

Fig. 2c One iteration of the refinement process.

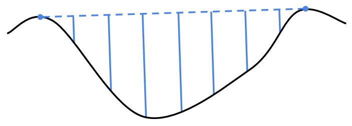

Fig. 2d The final refined endpoint locations shown by the dashed line and the sectors used to form the shape vector.

DEMs is due in part to their high availability across the world. When a query photograph needs to be localized, the image can then be sent through our processing pipeline, outlined in Fig. 1. First, a user marks a rough estimate of the skyline location, which is then automatically refined by our system. We then extract features from the query skyline to be matched to database skyline features in order to locate a corresponding view in the database. Using geometric hashing [9] and nearest-neighbor techniques [10], we rapidly and aggressively prune the space of candidate matches. We then compute alignments between the query skyline and a reduced subset of the database skylines in order to determine which database skyline matches the query image, thus completing the localization process.

We begin with a discussion of our concavity and convexity features in Section 2, before moving onto generation of a synthetic skyline database in Section 3. We then outline the query recognition process in Section 4, and discuss our experimental results in Section 5. Finally, Section 6 concludes with a summary of our contributions and a discussion of future work.

2 Concavity Features and Feature Detection

In order to cope with unknown camera parameters of the query photograph, it is necessary to build scale-invariance into our system. Lamdan et al. use scale-invariant concavity and convexity features for object recognition from boundary curves [9]. We utilize features that are similar in concept for skyline matching. However, since the notion of a concavity or convexity is poorly defined on open curves such as skylines, our method differs significantly in its details. For the remainder of this paper,

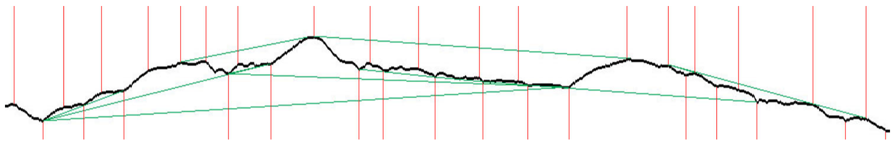

Fig. 3 A query skyline, the detected points of extreme curvature (indicated with vertical lines), and the refined concavity features (indicated with straight lines lying on the skyline). stretched vertically by a factor of 1.5 for illustrative purposes.

we use "concavity" to refer to both concavities and convexities, unless otherwise noted. A high-level overview of feature extraction is provided in Fig. 2. We now outline the process in greater detail.

First, we use the method developed by Fischler and Wolf [11] to detect points of extreme curvature on the input skyline, as shown in Fig. 2a. These points are found almost exclusively within concavities on the skyline. However, in practice they are too unstable to be used for localization by themselves. Rather, we use them to initialize the locations of our concavity features as follows: for each detected point of extreme curvature, we select a point to the left and to the right as a rough estimate of the concavity's endpoints, as seen in Fig. 2b. We then refine this estimate by iteratively and alternately moving the endpoints away from each other. Specifically, we push each endpoint out until the slope formed between the initial curvature point and the endpoint reaches a local maximum. Once this occurs, we continue examining an additional δ points. If a higher slope is found within these additional points, we continue pushing the endpoint outwards; otherwise, we leave the endpoint at the local maximum and begin pushing out the other endpoint. A sample iteration of this refinement process is shown in Fig. 2c. This process repeats alternately between the two endpoints until neither endpoint can move any further. For the sake of efficiency, we also impose an iteration limit, although in practice this limit is rarely reached. The final, refined concavity feature can be seen as the dashed line in Fig. 2d.

Even though this process results in an initial set of basic features, we can obtain additional, more complex features by examining overlapping features. Specifically, for any two overlapping features, we select their distant endpoints as an initial estimate of the concavity containing the two and refine as before. Further linking e.g. linking non-overlapping concavities is possible, but has been empirically found to negatively impact localization performance.

We perform an additional filtering step to remove features that are of low reliability. In particular, features with endpoints near the edges of the image are of dubious quality: the concavity slope may continue to rise past the end of the image, or it may fall off steeply just beyond the edge of the image. To avoid false detections, features with at least one endpoint within a certain distance of the image edge are discarded.

In addition to the endpoints, we also characterize each feature as a d-dimensional vector as shown in Fig. 2d. In this step, we use feature curves, which are the portions of the skyline lying between a feature's endpoints. We apply a similarity transformation to the feature curve that maps its endpoints to $(-1, 0)$ and $(1, 0)$, thus normal-

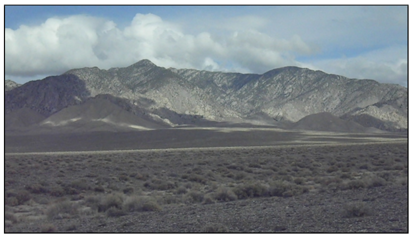
Fig. 4a A query image.

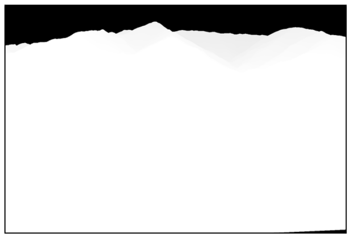
Fig. 4b Its corresponding database view.

izing each feature to a fixed length and orientation. This has two important effects. First, in normalizing the length of the feature, we account for any scale differences between feature curves, thereby achieving scale invariance. Second, because the orientation of each feature is normalized to lie along the x-axis, we achieve in-plane rotation invariance as well. Thus, barring any quantization effects, any two features curves with the same shape are normalized to the same final feature curve. After this normalization step, the d-dimensional feature vector is formed by computing the area between the normalized curve and the x-axis for d disjoint regions of equal width, as shown in Fig. 2d. Figure 3 shows an example output of feature detection, along with the sectors used to form the feature vector.

3 Database Generation

Localization of a query's skyline is performed via matching to a database of synthetic skylines. The skylines upon which we perform our tests are generated using DEMs with a resolution of $\frac{1}{3}$ arc-second, or $10\,\mathrm{m}$, and which span a square region of $10{,}000\,\mathrm{km}^2$. They were obtained from the National Elevation Dataset of the United States Geological Survey. The vertical heights of the dataset have a root mean square error of 2.44 m.

We evenly sample the DEM at a $1000\,\mathrm{m}$ resolution along both the north-south and east-west directions, forming a 2D grid of sample points as discussed by Baatz et al. [6]. This results in a 100×100 grid of samples. We then use a skyline generator to synthesize the horizon as seen from each sample point. We have implemented two versions of the skyline generator: a GPU-based approach where the full scene is rendered, and a CPU-based approach where only the primary skyline is extracted.

For the GPU version, at each sample point we render twenty-four $30^\circ \times 20^\circ$ images at 15° offsets, covering the full 360° panorama at the location. If the skyline is not completely captured by the viewport, we progressively pitch the camera upward until the full horizon is rendered. For our tests, we generate $100 \times 100 \times 24 = 240{,}000$ distinct images in total. Due to the large volume of images to render and the high resolution of the DEM, optimizations are needed in order to efficiently generate skyline databases. The DEM is initially tiled into chunks of 256×256 points,

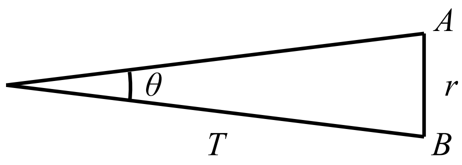

Fig. 5 From the spatial resolution of the downsampled DEM r and the angle θ swept between adjacent points A and B on the output skyline, we can determine at what distance T to begin using lower-resolution DEM tiles.

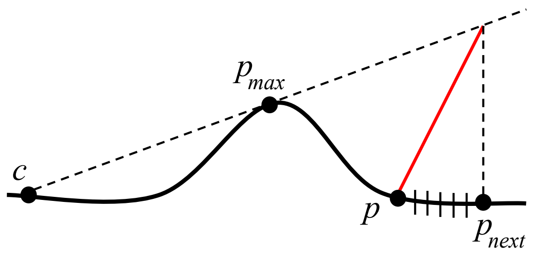

Fig. 6 A side view of the DEM during CPU-based skyline synthesis. Using an estimate of the maximum slope of the DEM, indicated by the slope of the diagonal line extending from p, we allow ourselves to jump from p to p_{next} without sampling the points between.

and an additional subsampled version of each tile is also created. The points within each tile are then formed into a mesh of triangles, which is saved along with the tile.

When rendering a viewpoint, we only render the triangles for the tiles that are in view, and of those we only select the high-resolution version if the angular resolution of the skyline is finer than the angular diameter of neighboring DEM points within the lower-resolution tile. In practice, this translates to setting a distance threshold T where tiles further away than this threshold can be of lower resolution. Consider Fig. 5, in which A and B are adjacent points between output skylines. Using the fact that the angle θ swept between these points is very small, we approximate T as r/θ where r is the resolution in meters per point of the downsampled DEM tiles. Given that the horizontal field-of-view for our system is $30°$ and assuming that each view in the database is 1500 pixels wide, the angular resolution per pixel θ is 3.49×10^{-4} radians. Then, for a subsampled DEM resolution of 30 meters between points, T can be estimated as $30/(3.49 \times 10^{-4})$ m, or about 86 km. This procedure allows for fine-grained selection of the DEM points to render. On an NVIDIA GeForce GT 650M, our skyline renderer generates over four 1500×1500-pixel images per second. Figure 4b shows an example rendered skyline. Once the $240{,}000$ skylines have been synthesized, we extract features as outlined in Section 2. The final database then consists of all skylines and their features.

The CPU version directly generates the primary horizons from the DEM, allowing for faster performance. For each column in the output skyline, a ray passing through that column is shot outwards from the camera location. We trace each ray over the DEM, sampling points along it, in order to find the DEM point along each ray with the highest elevation angle to the camera point. This angle then determines how high the skyline extends in that particular column. As we are only interested in the highest elevation angle along a ray, further optimizations are possible. A naïve approach would uniformly sample the elevation of every DEM point along the ray; however, if we assume an upper bound on the terrain's slope across the region, then we can sample points along the ray more sparsely, as shown in Fig. 6. In this figure, c denotes the camera position, p_{max} denotes the highest elevation point sampled so far, p denotes the last point sampled, and the slope of the diagonal line extending from p shows the estimate of maximum terrain slope. The naïve approach would sample at every point, represented by the tick marks, but under the maximum slope assumption, we can skip all points until p_{next} as we are guaranteed that no intermediate point has a higher elevation angle than p_{max}. Thus, the distance along each ray we can advance before sampling another point is a function of two factors: the maximum slope, and the difference between the maximum elevation angle seen thus far and the elevation angle of the point last sampled. In addition, we can use the determined maximum elevation angle of neighboring pixels in the skyline to estimate the elevation angle of the current pixel. Due to these optimizations, this method runs orders of magnitude faster than the previous version. Specifically, on a quad-core Intel Core i7, it generates over two hundred and fifty 2000-pixel-wide skylines per second, which is 60 times faster than the GPU based method.

4 Query Localization

The query localization process consists of four major steps, as diagrammed in Fig. 1. First, a skyline is extracted from a query photograph. Next, features are extracted from the query skyline. Then, a first round of matching feature shapes and endpoints occurs in order to rapidly narrow the set of candidate match locations. Finally, a more exhaustive alignment match occurs, producing a final ranked list of candidate database views. This list is used to generate a confidence map for user use in localizing the query photograph.

4.1 Skyline Extraction

Query processing begins with a user-directed skyline extraction process that follows a simple basic pattern: the user draws an approximate skyline on the query image, edge detection is run on the vicinity of that approximation, and finally a skyline is extracted from the resulting edge map.

For extraction of a skyline from a map of edge vs. non-edge pixels, we use the strategy detailed in Lie et al. [12]. The dynamic programming algorithm first builds paths across the edge pixels in the image, then selects the one that is cheapest when evaluated on criteria such as smoothness, altitude, and number and size of gaps. For edge detection, we use a variant of Canny edge detection [13].

4.2 Endpoint Matcher

The first stage of skyline matching deals with matching endpoints of concavity features in the query and database skylines. It is done through a geometric hashing technique inspired by the line matching technique in [9]. In contrast to [9] which uses a line segment and an additional point to create an affine-invariant basis, we use a concavity feature as our basis, as explained in Section 2. We opt to forego affine-invariance because we have empirically found affine-invariant features to not possess as much discriminative power as similarity-invariant features. Additionally, the loss of affine-invariance is mitigated by the fact that our database is sampled uniformly, and thus usually has a view that is similar to the query view. We now outline the basic implementation of our geometric hashing method.

After generating our feature database as described in Section 3, we create a hash-table to be used in the localization stage. Assume that a given skyline s has n features. For each feature $f \in s$, we find a 2D similarity transformation that maps its two endpoints to $(0,0)$ and $(1,0)$. We then apply this transformation to s and all of its features, generating a new configuration that we refer to as being normalized with respect to f. This transformation removes the effects of in-plane rotations and scales: any transformed version of this skyline s', when normalized with respect to its corresponding transformed feature f', results in the same configuration of feature endpoints as s when normalized to f. The new endpoints of the remaining $n-1$ features of s are used as indices to the hash-table. At each index, we store the pair (s, f)—that is, the skyline and the feature used to normalize it.

This process is repeated for all skylines in the database to create a single hash-table used for endpoint matching. Note that, since the building process does not require any knowledge of the query skyline or features, this hash-table can be pre-computed offline and saved for later use.

When a query skyline and its n features are provided as input to the endpoint matcher, we perform a similar normalization step. For each feature, we normalize the skyline and its other features with respect to it. We then use each of the other normalized $n-1$ features to index into the hash-table built in the previous step. Each index operation produces a list of skyline/feature pairs, and each pair receives one vote. We then construct a list of votes for candidate skylines, where the number of votes a skyline receives is the maximum number of votes of any of its skyline/feature pairs.

After this process has been repeated for each of the n query features, we are left with n ranked lists of candidate database skylines. We combine these lists into a final list by simply summing all votes together. This final list is then resorted to generate the final, ranked list of database skylines for the endpoint matcher, where more votes translates into a better ranking. Figure 7 shows an example match detected by this

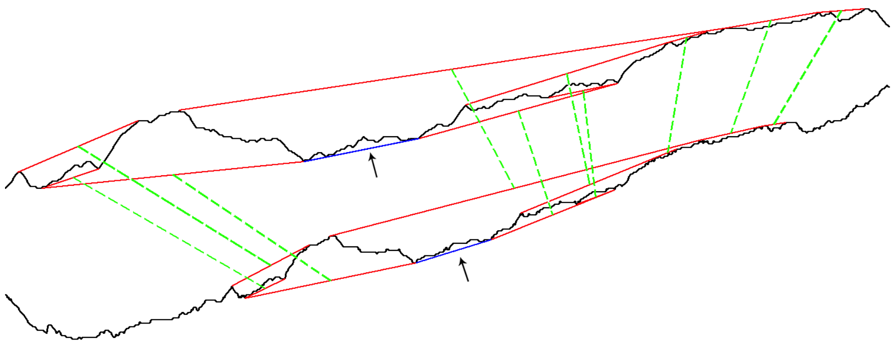

Fig. 7 A resulting match from the endpoint matcher between a database skyline (top) and a query skyline (bottom). The skylines have been stretched vertically by a factor of 3 for illustrative purposes.

matcher, where the features marked with arrows form the pair used for alignment, and the dashed lines denote features correspondences found by the matcher.

As previously discussed, by normalizing skylines with respect to their features, we achieve similarity invariance. Furthermore, our system is robust to partial occlusions; since the voting step uses every feature in every skyline, no single occluded feature significantly impacts the result. We now discuss a variety of strategies we employ to ensure the endpoint matcher is as robust as possible.

Soft Quantization in Matching Each time we query the database hash-table with a normalized feature in the matching stage, we use a soft quantization of the normalized feature to account for noise in the query skyline. We achieve this soft quantization by querying a set of points rather than a single point for each hash-table access. For a given query feature q and one of its endpoints x, the size of this set is determined by

$$s(q,x) = R + \min\left(\frac{||x||_2}{F + \sqrt[4]{q_{len}}}, M\right) \tag{1}$$

where $||x||_2$ indicates the distance of x from the origin, q_{len} indicates the distance between the endpoints of q and R, F and M are user-specified parameters to be described shortly. When indexing into the hash table, we index into all entries that fall within a box with side length $2s(q,x)$, centered at the endpoint x's location.

The above equation can be explained as follows. To account for noise, we allow the user to specify range R, ensuring that we always examine at least within a $2R \times 2R$ border from the original endpoint. Furthermore, since noise is amplified for points further away from the origin, we expand this window with an additional $||x||_2$ term. However, if left unchecked, this term can dominate. Thus, we attenuate its effect by F, a falloff term, and $\sqrt[4]{q_{len}}$. Larger features tend to be more stable after normalization, so we reduce the effect of the $||x||_2$ term by $\sqrt[4]{q_{len}}$. Finally, as a final precaution, we limit the contribution of this term to maximum value M. For our current system, we set $R = 20$, $F = 200$, $M = 10$.

Vote Threshold During the matching stage, n skyline vote lists are aggregated into a single final list. In order to reduce the influence of feature correspondences that occur by chance, we prevent an individual vote list from contributing to the final vote list for a given skyline unless the number of votes for that skyline in the individual list is above some threshold t. For our current system, $t = 4$.

Concavity/Convexity Alignment When querying the database hash-table with a normalized feature, we ensure that the curve type of the normalized feature used matches the one in the database. In other words, if a convex normalized feature is used as a key to an entry to the hash-table, we ensure that only a convex normalized feature can be used to get that entry in the hash-table, since a convex-to-concave alignment is almost certainly erroneous.

Normalization by Features in Window If a database skyline has a large number of features, then the likelihood that the skyline receives feature correspondence votes simply by chance is higher. As a result, we need to normalize the number of votes a database skyline receives based on the number of features it contains. However, this normalization should not consider features that have no chance of matching query features, such as those that fall entirely outside the query skyline after alignment occurs. Thus, we normalize the skyline's votes by the number of features in the query's window, where the window is defined as the smallest axis-aligned bounding box that contains all query feature endpoints.

4.3 Shape Matcher

The matcher described in Section 4.2 considers only locations of feature endpoints without regards to the feature shapes. We now outline another basic matcher that considers only shape without regards to feature configurations.

Since each feature's shape is characterized as a d-dimensional vector shown in Fig. 2d, we can evaluate the similarity in shape between two features by computing the Euclidean distance of the shape vectors in $\mathbb{R}^d$. Thus, when processing a query skyline and its features, we use a k-d tree containing the database features for efficient retrieval of nearby shape vectors [10]. Since the k-d tree does not depend on the query features at all, it can be precomputed offline and stored. Doing so allows for matches to be performed in a matter of seconds.

To perform a match, we examine each query feature and index into the k-d tree. We retrieve all database skylines containing a feature within distance D from the query feature, and any database skyline with such a feature receives a vote. Since rare features are more discriminative than common features, we weigh a feature's vote using a function of the number of database skylines it votes for. More specifically, we weigh each vote by its inverse document frequency defined as

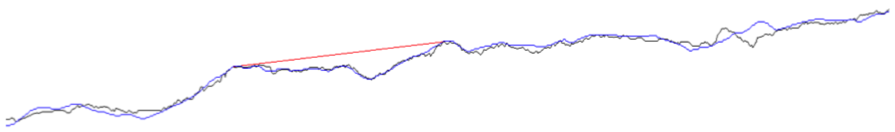

Fig. 8 An example alignment found by the alignment matcher. The query skyline is drawn in black, the database skyline is drawn in blue, and the feature used for alignment is drawn in red.

$$idf(f_q) = \ln \frac{|S|}{|\{s \in S \mid \exists f_d \in s : ||f_d - f_q||_2 < D\}} \tag{2}$$

where f_q and f_d are query and database features respectively and S is the set of all database skylines.

Once voting is complete, the shape matcher outputs a ranked list of database skylines, ordered by descending vote count.

4.4 Alignment Matcher

As shown in Fig. 1, a small subset of the best-ranked results from the endpoint and shape matchers is selected and fed as input into the final alignment matcher. Specifically, to collect a set of N candidate database skylines, we collect the intersection of the top p endpoint matcher results and the top q shape matcher results. Any remaining slots are then split with a $1 : 2$ ratio between the top endpoint matcher and shape matcher results respectively. This combination step is crucial to the performance of the system. The value of N can be set so as to balance accuracy and speed: higher values of N favor accuracy, whereas lower values favor speed. For our system, $N = 50{,}000$, $p = 30{,}000$, $q = 60{,}000$, resulting in an overall runtime of 2 hours on an 8-core 2.13 GHz Intel Xeon. The alignment step is by far the most computationally intense part of our entire system and as such dominates the run time in a significant way.

When aligning a query skyline against a candidate database skyline, we consider the Cartesian set product of the skylines' features. Each query-database feature pair defines a potential alignment of the two skylines. In particular, for query and database skylines s_q, s_d and features f_q, f_d from each skyline respectively, we find a similarity transformation that maps the endpoints of f_d onto the endpoints of f_q. This transformation is applied to s_d to effectively "overlay" the two skylines onto each other.

We evaluate each potential alignment by sampling 1000 points from the overlapping region, then computing an error function

$$E(s_q, s_d) = \ln |O(s_q, s_d)|^{-1} \sum_{i=1}^{1000} (s_d[i] - s_q[i])^2 \tag{3}$$

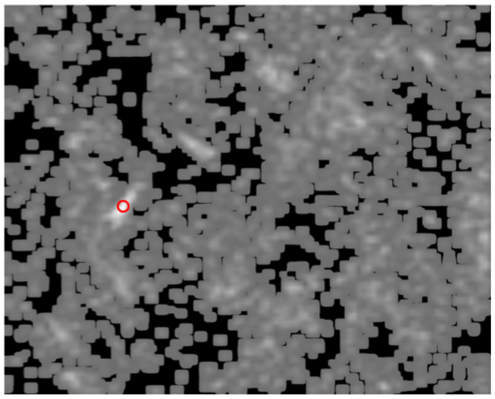

Fig. 9 An example confidence map, with the ground truth location circled. Higher intensities denote locations with higher confidence.

where $|O(s_q, s_d)|$ denotes the size of the overlap between the skylines s_q and s_d. We divide the squared distance between the two curves by the logarithm of the overlap size, since a larger overlap should translate into a more confident match i.e. a lower error score.

For a particular candidate database skyline, we find its best alignment and assign it that score. We can then produce a ranked list of candidate locations by sorting the skylines in ascending order of error score. Figure 8 shows an example top database match overlaid on its corresponding query.

To exclude degenerate alignments such as alignments in which only a small fraction of the skylines overlap, we require any alignment to result in at least a third of both skylines to overlap. We also exclude any alignments that match a concavity to a convexity or vice versa similar to the endpoint matcher, as well as any alignments with a scale difference above a threshold.

4.5 Confidence Map Generation

After the alignment matcher outputs the final rankings for each candidate database skyline, we generate a confidence map such as the one in Fig. 9 indicating how likely a match is for each grid location within the region of interest.

Even though our localization system uses orientation information during the matching process, our confidence maps show only location. Thus, the very first step is to find the minimum error score for each grid location. This results in a ranked list of locations, rather than individual views. We now take the top k locations and plot them on a confidence map. The intensity value of each location is determined based on the error score; the top location has the lowest error score and thus the highest intensity, whereas the k-th location has the highest error score and thus the lowest intensity. The intensity of all other locations is determined through linear interpolation on their error scores.

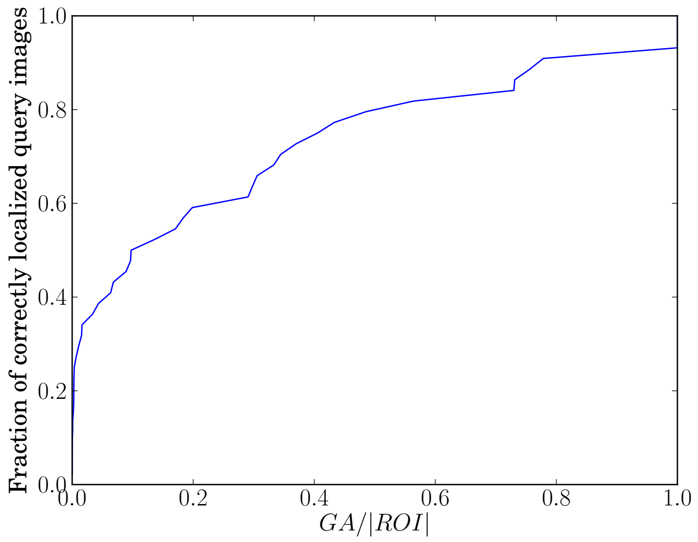

Fig. 10 Curve of system performance.

As a final processing step on our confidence maps, we apply a uniform box filter with side lengths equal to that of two grid locations. This provides more lenience and allows grid locations with high intensity to count for their neighboring locations as well.

5 Evaluation and Results

Since our primary motivation is user-aided geolocation, typical metrics such as the percentage of top-1 matches are ill suited for evaluating our system. Rather, we use the geolocation area (GA), or the area of the region with confidence value greater than or equal to that of the ground truth location, and compare it to the area of the entire region of interest ($|ROI|$). Intuitively, the GA represents the area over which a user would have to search in order to find the ground truth location, assuming the search is done in order from highest to lowest confidence value. Figure 10 plots the fraction of queries correctly located against different $GA/|ROI|$ values. As shown, our system correctly locates over 25 % of the queries with a $GA/|ROI|$ of less than 0.01, and over 50 % of the queries with a $GA/|ROI|$ of less than 0.10.

Figure 11 shows examples in which the system performs well or when it fails. As seen in Fig. 11a, the best performing query images are those of distant ridges taken from flat ground, as these images are invariant to small shifts in location. When these conditions are met, we obtain $GA/|ROI|$ scores as low as 1×10^{-5}. In contrast, photographs of nearby formations or photographs taken on mountainous

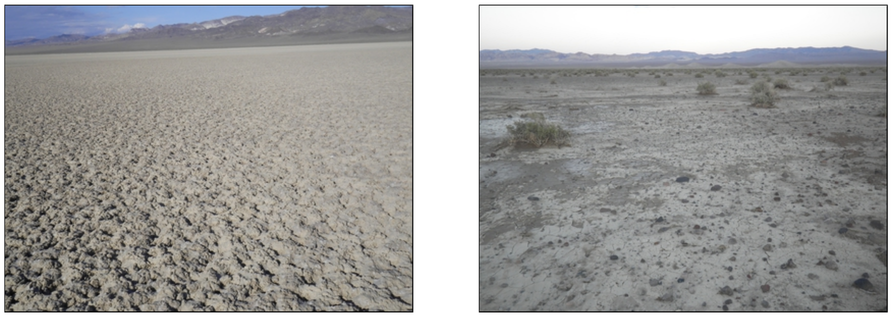

Fig. 11a Two of the best performing query images, taken from flat ground with distant skylines.

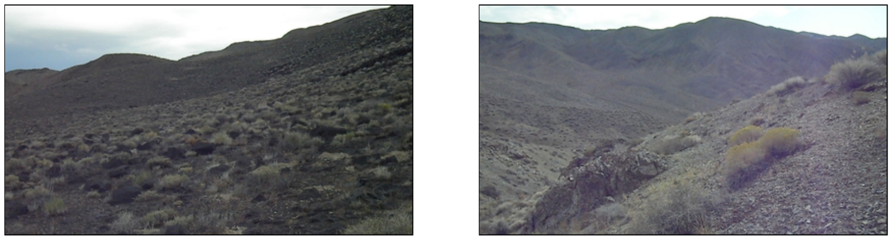

Fig. 11b Two of the worst performing query images, taken on a slope of mountainous terrain with nearby ridges and skylines.

terrain, as shown in Fig. 11b, are very sensitive to small changes in location. Often, this means that our database is not sampled densely enough to contain a good match.

6 Conclusions and Future Work

In this paper we have presented a system for geolocation of untagged imagery using only digital elevation models. Our approach extracts the skyline from a query photograph and detects stable concavity features for use in matching against a database. We introduced a modular matching system that allows users to balance accuracy with speed as necessary and can be easily parallelized by simply distributing the database across multiple machines. Finally, we showed that this method is robust, attaining incredibly precise locations on many test queries.

By far the major weakness of our system is the relatively sparse sampling of our database. Our concavity features are sensitive to out-of-plane rotations, so denser sampling would ensure that a proper match exists in our database. However, since our current database generation uses a uniform grid across the ROI, increasing the sampling density causes a multiplicative increase in the number of database skylines and thus the runtime. If we can detect database sampling points at which the skyline is nearby or has sloped terrain—the two conditions under which the view is highly

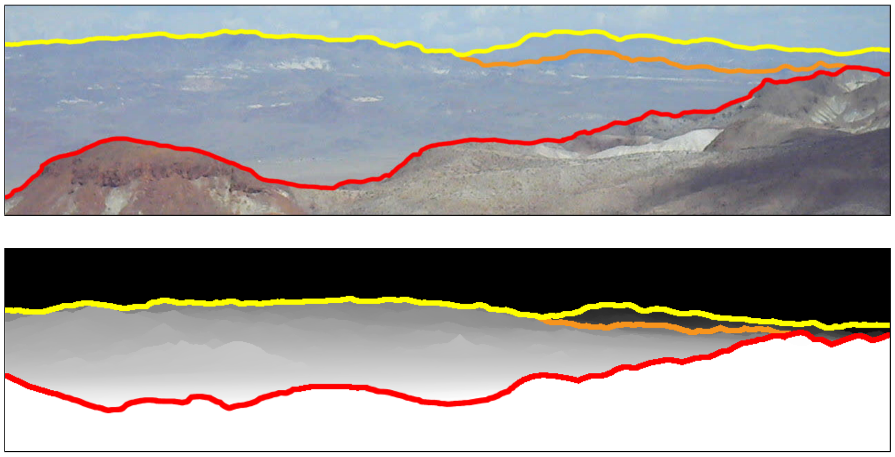

Fig. 12 A query (top) and corresponding database view (bottom) with multiple corresponding skylines identified.

sensitive to precise location—then we can adaptively increase the sampling density at only those points, mitigating the multiplicative increase effect.

There is also an abundance of discriminative information in secondary "horizons" below the primary one, formed by the interplay between ridges of different heights, as shown in Fig. 12. The additional skylines provide more features with which to perform alignment, and might allow the system to recover in cases where the primary skyline is unclear or heavily occluded.

Acknowledgments

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Air Force Research Laboratory, contract FA8650-12-C-7211. The U.S.Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, AFRL, or the U.S. Government.

References

1. J. Hays, A. Efros, in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on* (IEEE, 2008), pp. 1–8
2. E. Kalogerakis, O. Vesselova, J. Hays, A. Efros, A. Hertzmann, in *Computer Vision, 2009 IEEE 12th International Conference on* (IEEE, 2009), pp. 253–260

3. G. Schindler, M. Brown, R. Szeliski, in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on* (IEEE, 2007), pp. 1–7
4. J. Zhang, A. Hallquist, E. Liang, A. Zakhor, in *Image Processing (ICIP), 2011 18th IEEE International Conference on* (IEEE, 2011), pp. 3677–3680
5. W. Zhang, J. Kosecka, in *3D Data Processing, Visualization, and Transmission, Third International Symposium on* (IEEE, 2006), pp. 33–40
6. G. Baatz, O. Saurer, K. Koeser, M. Pollefeys, in *Proc. European Conference on Computer Vision* (2012)
7. F. Stein, G. Medioni, Robotics and Automation, IEEE Transactions on **11**(6), 892 (1995)
8. R. Talluri, J. Aggarwal, Robotics and Automation, IEEE Transactions on **8**(5), 573 (1992)
9. Y. Lamdan, J.T. Schwartz, H.J. Wolfson, in *Computer Vision and Pattern Recognition, 1988. Proceedings CVPR'88., Computer Society Conference on* (IEEE, 1988), pp. 335–344
10. M. Muja, D.G. Lowe, in *International Conference on Computer Vision Theory and Application VISSAPP'09)* (INSTICC Press, 2009), pp. 331–340
11. M.A. Fischler, H.C. Wolf, Pattern Analysis and Machine Intelligence, IEEE Transactions on **16**(2), 113 (1994)
12. W.N. Lie, T.C.I. Lin, T.C. Lin, K.S. Hung, Pattern recognition letters **26**(2), 221 (2005)
13. J. Canny, Pattern Analysis and Machine Intelligence, IEEE Transactions on (6), 679 (1986)