Starid

Noah Smith

Dec 28, 2022

CONTENTS:

code is split into two folders, with the starid folder playing a special role as a python package for release on the pypi package repo.

CHAPTER

ONE

STARID PYTHON

ideally, a lot can be done in python without knowing much about the underlying cpp - there's a model of the sky, a toolbox for working with it, and it's available via starid.py.

CHAPTER

TWO

LIBSTARID CPP

fast inner loops for working with star triangles. also useful for working with lots of three-dimensional star pointing vectors, though this is probably reasonable in python as well. in any case, hardware acceleration of vectorized computations, matrix and vector math via eigen. when computations become heavier, move them from python into cpp.

CHAPTER

THREE

INSTALL

have switched to a container build environment - reproducible using the dockerfile. everything needed is there and can be reproduced for a local environment. docker image is available on docker hub.

CHAPTER

FOUR

HIGHER-LEVEL

python starid object making all of the lower-level stuff available - it's the interface between python and the underlying cpp. the starid object could in some sense be a singleton - there should be only one. on the other hand, it's possible to imagine paths where this is no longer true... imagine using two sets of stars from the star catalog - one including fainter stars. in short, two skies. we could have two starid objects at the same time, one for each sky.

4.1 starid.py

4.2 api.cpp

`class libstarid_.api.Api`

handles calls from python code, for example starid.py. where reading and writing of data files is done, the headers-only cereal library is used. this is the cpp version of the python pickle library. binary objects are moved directly to and from disk. binary data is more efficient - it's smaller and faster.

`NOMAD`(*image_pixels*)

performs identification and outputs an id - if that matches the starndx used by image_generator to create the image, identification was a success.

`SETTLER`(*image_pixels*)

performs identification and outputs an id - if that matches the starndx used by image_generator to create the image, identification was a success.

`image_generator`(*starndx*)

for the star indicated by starndx, generate a standard lo-fi image, with the sky randomly rotated. this is an image for which we want to perform star identification. it's the input to the image_identifier method, which performs identification and outputs the resulting id - if that matches the starndx, identification was a success.

`read_sky()`

start a sky object from a sky data file using cereal. the key part of sky data is a three-dimensional pointing vector representing the direction to each star in the celestial reference frame.

`read_starpairs()`

start a starpair object from a starpair data file using cereal. this object is the key ingredient for star triangles - a star pair is a triangle-side in star triangles, it's the 'fundamental particle' of a triangle view of the sky. every triangle is made of three starpairs. in a starpair object, for each star, the starpairs with each of its near neighbors are represented, pre-computed and ready for use. computing this object is relatively heavy and we want to do it once, in advance, and then reuse it from there.

`write_sky()`

generate a sky object and write a sky data file using cereal. generating a sky object from scratch can take a noticeable amount of time - a few seconds?

`write_starpairs()`

generate star pairs for a given set of stars and write a starpair data file using cereal. this scales up quickly with star density on the sky - so with the brightness threshold we're using. including fainter stars increases the density exponentially, along with the computational cost of generating star pairs. for each star, we only care about its neighbors that can appear in the images we're using. bigger images mean more neighbors to generate star pairs for. our current baseline is visual magnitude 6.5 - the roughly eight thousand stars visible to the human eye - and an image size of eight by eight degrees, typical for a certain generation of star trackers.

CHAPTER

FIVE

SKY MODEL

interactive model of the sky, based on a set of stars from the nasa skymap star catalog. the stars are defined by a brightness cutoff - all stars brighter than the cutoff. with a cutoff of visual magnitude 6.5, this means slightly more than all stars visible to human eyes - 8876 in total.

5.1 skymap.cpp

`class libstarid_.skymap.`**`Skymap`**

bring the nasa skymap sky2000 v5r4 star catalog in. there are peculiarities to this catalog, and they should be reflected in its representation here. briefly, v5r4 was targeted at real world star tracker users - it tried to fuse results from multiple predecessor catalogs to provide useful information.

5.2 sky.cpp

`class libstarid_.sky.`**`Sky`**

model the sky, based on the skymap object. the key input parameter is the star brightness threshold - with visual magnitude 6.5 the sky is about nine thousand stars, and that number grows exponentially as dimmer stars are included.

`image_generator`(*starndx*)

creates a standard image for the target star, ready for feeding into a star identifier. the format is 28 x 28 pixels - lo-fi, the way we like it. makes thing tougher on us. and also by no coincidence matching the classic mnist character recognition data set. the story behind that is a long one, discussed elsewhere in the project.

`stars_in_ring`(*p*, *radius*, *table*)

when we break the skies three-dimensional search space down into three one-dimensional search spaces, the one-dimensional spaces represent rings on the sky. we have three rings, and the stars we're interested in are in their intersection. this intersection of three rings is in some sense a three-dimensional hash map into the sky.

`stars_near_point`(*x*, *y*, *z*)

given a three-dimensional pointing vector in the celestial reference frame, return the identifiers for nearby stars. this is fundamental - we have to be able to call up the stars near a target on the sky. it's a rich problem we'll be discussing throughout the project documentation. here we break the three-dimensional search space down into three one-dimensional search spaces, and create a map or hash-index into each of those. in a sense - it's a three-dimensional hash map into the sky.

`start`(*pathin*)

initialize the sky. first generates a skymap object and then picks out the information needed here, with some enrichment - in particular with three-dimensional vectors in the celestial reference frame.

5.3 starpairs.cpp

`class libstarid_.starpairs.Starpairs`

foundation for star triangles - a star pair is a triangle-side - it's the fundamental particle of a triangle view of the sky. every triangle is made of three starpairs. in a starpair object, for each star, the starpairs with each of its near neighbors are represented, pre-computed and ready for use. computing this object is relatively heavy and we want to do it once, in advance, and then reuse it from there. so cerealize it to a starpairs file and read in the starpairs object from that whenever possible, rather than generating from scratch.

`generate`(*sky*)

create a starpairs object from scratch. this can written to disk using cereal, and read from there in the future to bypass these computations.

`pair_labeler`(*catndx1*, *catndx2*)

returns a unique string for the pair, consisting of the catalog ids for the member stars - a useful identifier for the pair.

`pairs_for_angle`(*angle*, *tol_radius*)

for an angle, what are the candidate star pairs? creates the representation of stars used in star triangles. there, a star is a collection of associations with its near neighbors - its essential feature is its membership in pairs and triangle sides. what we do here is look at each star in turn, asking the question - what pairings do we care about for the star triangle representation of the sky we're going to use? the tuning parameters representing the answer to that question are the angle between pair members and a measure of tolerance or sensitivity.

CHAPTER

SIX

IDENTIFICATION

view the sky as triangles of stars. for the target star, it's a member of a set of triangles - eliminate candidate ids based on the geometry of these triangles. this is an iterative process and the inner loop is comparing triangle geometries. the overall speed depends on this inner loop, so the focus is on making it as efficient as possible.

identifies the target of a star image, using the triangles formed by neighboring stars within the image. the fundemental particles are actually pairs of stars - in a sense individual stars don't exist here, what exists are pairs of stars, acting as sides of triangles - so a key object handed to the identifier in its constructor is a starpairs object, containing all of the relevant pairs. when possible, the starpairs object was loaded from a cerealized starpairs file, rather than generated at run-time.

6.1 startriangleidentifier.cpp

`class libstarid_.startriangleidentifier.`**`NOMAD`**

star recognition focused on a chain of triangles and basesides - side2 of each triangle is the baseside of the following triangle. during feedback, these shared side2 -> baseside pairs are the path for information to flow backwards - increasing the constraints on the initial triangle baseside and basestar. the name NOMAD relates to how the chain of triangles wanders away from the target star and initial triangle.

`run`(*pixels*)

recognize target star from the image pixels.

`class libstarid_.startriangleidentifier.`**`SETTLER`**

the target star ia always star a. star b is a neighbor star, and an abside is a star pair and triangle side with the target as the first member of the pair. in the inner loops, additional stars c and d are involved. first an abca triangle is formed. this constrains the abside. then for an abca triangle, a sequence of abda triangles are formed, further constraining the abside. when we reach an abda that eliminates all but one star pair possibility for the abside, we've recognized the target star. the name SETTLER comes from the idea that we never move away the target star, we're settling around it.

`get_angs_c`()

examine a candidate for star c before using it to form triangle abca. we want the angles between stars a, b, and c to be appreciable. the angles remain in angs_c for later use.

`get_angs_d`()

examine a candidate for star d before using it to form triangle abda. we want the angles from stars a, b, and c to be appreciable. the angles remain in angs_d for later use

`run`(*pixels*)

recognize target star from the image pixels.

6.2 startriangle.cpp

class `libstarid_.startriangle.`**`StartriangleNOMAD`**

NOMAD triangle. focus is on the basestar and baseside - nomad is about a chain of basesides, each increasing the constraints on the preceding basestars. first constructor here is for the initial triangle and has the target star as basestar. second constructor is for following triangles. each takes side2 from its predecessor and uses that as its baseside. the chain of triangles is a train of baseides - side2 of each triangle is the baseside of the following triangle. during feedback, these shared side2 -> baseside pairs are the path for information to flow backwards, all the way backward from latest triangle to the initial triangle - increasing the constraints on the initial triangle baseside and basestar.

`constrain``()`

in each of the three sides, there's a pairhalf1 -> pairhalf -> 0 or 1 concept. 0 is the default and means drop this pair. here we will mark pairs to keep by setting them to 1, all others will be dropped.

`feedback``()`

increase the constraints on the baseside in the prev triangle, using the baseside of the following triangle in the chain. as triangles are added, constraints flow backwards through preceding basesides and basestars. the chain of triangles is a train of baseides - side2 of each triangle is the baseside of the following triangle. during feedback, these shared side2 -> baseside pairs are the path for information to flow backwards, all the way backward from latest triangle to the initial triangle - increasing the constraints on the initial triangle baseside and basestar.

`stop``()`

stopping condition. true if basestars and basesides have been constrained to the point where only one possible basestar remains.

class `libstarid_.startriangle.`**`StartriangleSETTLER`**

SETTLER triangle. acts as the triangles abca and abda within the star triangle identifier inner loops. their are three triangle sides - representing three star pairs, each with an angular separation. each side is acted by a star triangle side object. star recognition focused on triangles that contain the target star - star a is always the target star, star b is a neighbor star, and an abside is a star pair and triangle side with the target as the first member of the pair. in the inner loops, additional stars c and d are involved. first an abca triangle is formed. this constrains the abside. then for an abca triangle, a sequence of abda triangles are formed, further constraining the abside. when we reach an abda that eliminates all but one star pair possibility for the abside, we've recognized the target star. until that happens, we continue picking new absides, with new abca triangles, with new abda triangles. the name SETTLER comes from the idea that we never move away the target star, we're settling around it.

`constrain_abca``()`

test candidate star pairs for the sides of an abca triangle.

`constrain_abda``(`*triangles*`)`

test candidate star pairs for the sides of an abda triangle.

6.3 startriangleside.cpp

class `libstarid_.startriangleside.`**`Startriangleside`**

act as one of the three triangle sides within a parent star triangle object. here stars is a representation of candidate star pairs that could belong to the side. ultimately - when we've recognized the target star, all but one candidate star pair is eliminated.

`countpairs()`

how many candidate star pairs remain in this side.

`drops()`

there's a pairhalf1 -> pairhalf -> 0 or 1 concept. 0 is the default and means drop this particular pair. here we drop all pairs that have not been set to 1, and reset all that remain to 0.

`update(`*side*`)`

used just for the abside currently being investigated, to update it based on the latest abca or abda triangle.

PYTHON MODULE INDEX

INDEX