

Plane Sweep Matching Users' Guide

1 Introduction

The purpose of this report is to provide a brief guide for new users of CSIRO's plane sweep matching code. We have tried to develop flexible, adaptable code that can be fairly easily tailored to specific environments. This document will help you begin to tailor the code to your environment.

This document will not provide you with a justification or description of our approach. For an overview of our approach, see [1]; for details of the algorithm and implementation see [2]; for details of our benchmarking approach and system configuration, see [3].

2 Basic Structure and Steps

The basic structure of the code is a single filter, followed by a chain of zero or more refines. The filter is the most complex and interesting component. Its task is to read objects from two catalogues, and generate a set of object pairs that might match, based on spatial proximity. These object pairs are then passed to a refine module, which may reject the candidate pair as a non-match, or accept it and pass it to another refine for further checking. The candidate pairs that survive through to the end of the refine chain are considered matches.

The passing of data down the chain is based on a metaphor of producers and consumers. The filter is given access to the catalogues through two object producers; it reads the objects produced by these object producers, and passes the results to one of three consumers: a consumer of accepted object pairs, and, for each of the two object producers, a consumer of rejected objects. Refine modules are provided with candidate object pairs by direct invocation, and they pass their results to one of two consumers: a consumer of accepted object pairs and a consumer of rejected object pairs.

The object producer is created with an object reader. This is the interface to a catalogue that allows the producer to read successive objects. This

implementation allows a single object producer class to be used with different sources, each with their own readers.

Deploying the code in your environment therefore requires the following steps:

1. For each of your catalogues, construct an object reader that reads objects from the catalogue and provides it to the filter;
2. Choose or construct an object consumer to handle the objects that are rejected by the filter;
3. Choose or construct a consumer to handle the candidate object pairs rejected by the refines;
4. Choose or construct a consumer to handle the accepted candidate object pairs;
5. Implement any further refine steps that you want invoked;
6. Construct the filter-refine chain; and
7. Set the filter running.

The rest of this report focuses on the implementation of catalogue cross matching. We have also used the plane sweep approach to implement nearest neighbours as discussed in [3].

2.1 An Example

Suppose you wish to cross-match a pair of catalogues on your system. One of your catalogues is stored as a large binary file while the other is accessible via SQL Server. Your first task is to implement two `ObjectReader` subclasses. The first `ObjectReader` subclass must be able to read objects from your large binary file, and report them according to the `ObjectReader` contract. The other `ObjectReader` subclass must construct and execute the appropriate queries to extract objects from your SQL Server, and then report them according to the `ObjectReader` contract.

These `ObjectReader` subclasses are then provided to two `ObjectProducers` which are in turn used by the filter to read objects one at a time with the results of filtering being passed to its consumers. Our next task, then, is to define the consumers for the filter. Let us suppose that the objects rejected from the SQL Server catalogue should be stored back in the database in a separate table. You must therefore implement a `ObjectWriter` subclass that

writes objects by storing them in the database. We will also suppose that objects rejected from the binary file catalogue are ignored. You will still need a writer, but your writer is free to do nothing.

These `ObjectWriters` are used to construct appropriate `ObjectConsumers` which write the objects as they are processed. The default behaviour of the provided `ObjectConsumer` with the default `ObjectWriter` is to simply count the number of objects processed and report the number of objects consumed upon completion.

The final consumer needed by the filter is a consumer of object pairs: the `ObjectPairConsumer`. This consumer uses an `ObjectPairWriter` in a similar way as the `ObjectConsumer` uses an `ObjectWriter`. An `ObjectPairConsumer` will most likely be used by a refine stage to further check the candidate pair as meeting some extra criteria. This combines the refine processing, determining if an object pair satisfy the further criteria, and writing the successful and unsuccessful object pairs to the appropriate `ObjectPairConsumers`.

Examples of how these various classes relate can be seen by the provided `Refine` subclasses, for example the `AngularSeparationRefine` class. This class performs a full angular separation test. That is, you will use the `AngularSeparationRefine` refine module to perform a full angular separation test.

Having adopted a refine, you now must answer again the question of what is to be done with the output. There are two issues: what shall we do with rejected object pairs; and what shall we do with accepted object pairs? In both cases, the answer is provided by implementing an `ObjectPairWriter` subclass, used by the `ObjectPairConsumer` subclass used by the `Refine` subclass.

3 Details

3.1 Choose your filter

For testing purposes, we have provided two filters. The `NestedLoopFilter` is a baseline filter that checks every possible pair of `Objects` for a match using a bounding box criterion. It runs extremely slowly. We suggest you only use the `NestedLoopFilter` for testing purposes and only on extremely small catalogues. The `DecPlaneSweepFilter` is the efficient filter that should be used operationally.

3.2 Object and Datum

Typically, a catalogue record will contain a lot of information about an observed source. Of this, only the spatial location information is used by the filter or any of the provided refines. The other information is not accessed and so need not be accessible. However, even though the filter and the provided refines do not access any data other than location data, it is highly likely that some of the other data will need to be passed down the chain, because it will be used by a downstream refine or consumer.

For example, suppose that once the spatial tests are complete, the candidate matches are passed to a further refine that checks for consistency in red-shift. Clearly this refine must have some data to work on, and so some data specific to red-shift must be passed. As another example, suppose that our catalogues are both tables in a database, and that the end result is to be a match table that lists pairs of IDs indicating record matches. Obviously the consumer that handles this must have access to the IDs of matched records.

Clearly, we need to be able to pass a variety of data down the chain, yet we would like to keep this hidden from the components that only use a small portion of that data. To manage this tradeoff, we define an `Object` class that contains only spatial location specific properties and methods. Our filter and spatial refine modules operate only on `Objects`.

`Object` contains the methods

```
double getRa();  
double getDec();  
double getDecSD();  
double getOrthoSD();  
double getSD();
```

The first two of these provide the nominal, or mean, location of the object, the next two provide information about the standard deviation, respectively along the declination dimension and orthogonal to the declination dimension, and the last one simply returns the greater of these two standard deviations. `Object` also provides a couple of methods for computing error bounds in the declination and right ascension dimensions.

Each specific deployment of our code will want to provide more than just spatial information, and so `Object` will be insufficient. You must therefore provide a class that (a) extends `Object`, and (b) provides the additional information you require. In our case, the additional information required was just the ID of the record, so we have implemented a `Datum` class that extends `Object` but also provides access to record ID.

This approach can be used for the refinements that you implement too. For example, before implementing a `RedShiftConsistencyRefine` class, you could define the information that it would need, and encapsulate it in a `RedShiftInfo` class. Subsequently, you implement the `RedShiftConsistencyRefine` class so that it requires a pair of `RedShiftInfo` objects. You then create an overall record class that extends both `Object` and `RedShiftInfo`. This overall record class can be passed into refinement modules of either type, and in both cases the refinement module only has access to the information necessary for it to do its job.

One further point is worth making: there is a decision to be made about how information is provided to downstream refinements and consumers. In the above example, one could provide the red-shift data at the start, so that it is passed down through the refinement chain until it reaches a refinement module that accesses it; or one could provide only an ID number, and then implement the red-shift refinement module's upstream consumer so that it goes back to the database and obtains the required red-shift data before invoking the red-shift refinement. There are advantages and disadvantages of both approaches.

3.3 ObjectProducer

One of the more time-consuming tasks involved in deploying the cross matching code to a specific system is the development of `ObjectProducers` that extract `Objects` from storage. `ObjectProducer` is an interface with the methods

```
bool hasNext();
Object * next();
double getMaxDecSD();
double getMaxOrthoSD();
double getMaxSD();
```

The most important methods are `hasNext` and `next`. These provide a simple iterator interface to your records. You *must* ensure that objects are returned in ascending order of declination. If two objects have the same declination, then they may be returned in either order. The `getMaxDecSD` method allows access to the largest value of the standard deviation along the declination dimension for a catalogue; similarly, the `getMaxOrthoSD` method allows access to the largest value of the standard deviation orthogonal to the declination axis. The `getMaxSD` method returns the larger of these two. Your implementation must therefore provide methods for computing this value.

The `ObjectProducer` is not expected to be subclassed. Instead, an `ObjectReader` is used to provide access to the source catalogue. It is the

`ObjectReader` class that fully encapsulates the specific details of how to read objects from a catalogue. We have provided a range of `ObjectReader` subclasses for reading from different file formats, most notably ASCII and binary files. Because catalogues are sometimes stored in multiple files with each file storing a distinct declination strip, we have provided `FileReaders` to handle such collections of files. Also, our 2MASS data was provided in multiple files partitioned by declination but not distinctly, so we have provided a class to handle that too. A database could easily be accessed by implementing an `ObjectReader` subclasses to do so. We have done this for an Oracle system and the code is available upon request, but has not been included in this release.

It is unlikely that any of our `ObjectReader` subclasses will be usable on your system. However, when you come to implement your own `ObjectReaders`, we hope that you will find most of the work already done, so that you only have to change the file format or the database query to make things work.

3.4 ObjectConsumers and ObjectPairConsumers

Consumers play the opposite role to Producers: they are invoked with an `Object` or a pair of `Objects`, and they respond by performing some action. There are two types: `ObjectConsumers` and `ObjectPairConsumers`. The filter uses two `ObjectConsumers` to handle those objects that it rejects, and a single `ObjectPairConsumer` to handle the pairs of `Objects` that it accepts as a candidate match; each refine has two `ObjectPairConsumers`: one for accepted pairs, and another for rejected pairs.

The specific details of writing `Objects` is managed by the `ObjectWriter` and `ObjectPairWriter` using `ObjectConsumers` and `ObjectPairConsumers` respectively, similarly as was done for reading objects by an `ObjectProducer` using an `ObjectReader`.

Our implementations of consumers are fairly rudimentary, and are unlikely to be very useful to you, with one exception. The exception is the special case of the `ObjectPairConsumer` that acts on a pair of `Objects` by invoking a downstream refine module. We have implemented this as `ObjectPairRefineConsumer`.

The default behaviour of the `ObjectPairConsumer` is to keep count of the number of `ObjectPairs` it processes and to report the number upon completion.

Note that there is no reason why you cannot use the same consumer as an argument to multiple modules. For example, if you want to keep track of all rejected pairs of objects, you may construct a single consumer to do so, and

use it as the consumer for every refine in the chain. Then each rejected pair of objects will be treated in the same way regardless of which refine module rejected it.

3.5 Implement your own refine

Each refine module must extend the abstract class `Refine`. This is fairly simple to do; the only method that must be implemented is

```
bool refine(const Object *, const Object *);
```

The task of this method is to determine whether or not the two `Objects` provided as arguments match. `Refine` also provides methods `reportMatch` and `reportNoMatch` for your convenience. Each execution of your `refine` method should result in exactly one call to exactly one of these reporting methods.

Your `refine` method should return a boolean value indicating whether or not the object pair has been accepted. If your refine method passes its results further down the chain, then the boolean value returned must take into account the downstream refinements. In this way the return value of `refine` is used to pass the acceptance status of an object pair back up the chain to the filter, where it is used to construct the sets of unmatched objects.

`Refine` also provides a `finished` method. The provided implementation ensures that the downstream consumers have their `finished` methods invoked. There is no need for you to override this method unless your refine method needs to take further action upon running out of data.

Probably the best way to get started on implementing your own refine module is to start with one of the provided refines (`BoundingBoxRefine` or `AngularSeparationRefine`) and use this as a template for developing yours.

3.6 Constructing the Filter Chain

Construction of the filter chain starts at the end of the chain and works back towards the filter. Construction of the filter requires knowledge of its consumers, but construction of the filter's `ObjectPairConsumer` requires knowledge of the downstream refine, which requires knowledge of its own consumers, and so on. Consequently, we start by constructing the final consumers, then construct the final refine, and work our way up the chain. For examples of filter chain construction, see the example mainlines in the `test/` directory.

4 Problems

4.1 Representation of Distributions

Our implementation has assumed that the spatial locations of our objects are normally distributed random variables whose variances can be described by a single standard deviation value. Although this simple specification of variance is valid for some catalogues, many catalogues specify their errors in ways that allow for arbitrary (positive definite) covariance matrices. For example, some catalogues provide error ellipses as the angle and length of the semi-major and semi-minor axes. We have not attempted to support these distributions, but a clever statistician such as yourself will have no trouble untangling the intricacies of multivariate statistics on the surface of a sphere and re-implementing this correctly.

References

- [1] David J. Abel, Drew Devereux, Robert A. Power, and Peter R. Lamb. An $O(N \log M)$ algorithm for catalogue matching. Technical Report TR-04/1846, CSIRO ICT Centre, Canberra, Australia, 2004.
- [2] Drew Devereux, David J. Abel, and Robert A. Power. Notes on the implementation of catalogue cross matching. Technical Report TR-04/1847, CSIRO ICT Centre, Canberra, Australia, 2004.
- [3] Robert Power. Benchmarking catalogue cross matching. Technical Report TR-04/1848, CSIRO ICT Centre, Canberra, Australia, 2004.