

---

# **Benchmarking Catalogue Cross Matching**

---

**Robert Power  
Drew Devereux**

**July 2004**

**Version 1.0  
26 July 2004**



1	Introduction .....	3
1.1	Purpose .....	3
1.2	Scope .....	3
1.3	References .....	3
1.4	Overview .....	3
2	Data Preparation .....	4
2.1	Source Catalogues .....	4
2.2	Files .....	4
2.3	Max Errors .....	6
2.4	Id field .....	6
2.5	Reading Test .....	6
3	Catalogue Cross Match .....	8
3.1	1XMM .....	9
3.2	SUMSS .....	9
3.3	Tycho2 .....	9
3.4	2MASS .....	10
3.5	USNO A2 .....	10
3.6	USNO B1 .....	10
4	Nearest Neighbour Matching .....	10
4.1	1XMM .....	11
4.2	SUMSS .....	12
4.3	Tycho2 .....	12
4.4	2MASS .....	12
4.5	USNO A2 .....	13
4.6	USNO B1 .....	13
5	Conclusions .....	13
Appendix A	System Configuration .....	16
Appendix B	Dec Plane Sweep Algorithm .....	17
Appendix C	Cross Match Algorithm .....	19
Appendix D	Nearest Neighbour Algorithm. ....	21
Appendix E	Notes on C++ implementation .....	22

# 1 Introduction

## 1.1 Purpose

The purpose of this report is to compare results achieved in using the plane sweep technique for processing astronomy catalogues. We have focused on catalogue cross matching and determining nearest neighbours and have explored various algorithm alternatives.

## 1.2 Scope

This document records the steps performed to prepare catalogue data and process it for the matching algorithms. The issues encountered and solutions derived in this process are documented. While the full details of the catalogue matching algorithm are detailed elsewhere ([2] and [4]), the appendices include descriptions of some aspects of the implementation we feel worthy of highlighting.

## 1.3 References

- [1] Devereux, D., "Notes on the Implementation of Catalogue Cross Matching" CSIRO ICT Centre Technical Report TR-04/1847.
- [2] Abel, D., Devereux, D., Power, R., Lamb, P. "An  $O(N \log M)$  Algorithm for Catalogue Matching" CSIRO ICT Centre Technical Report TR-04/1846.
- [3] Devereux, D., Power, R. "Plane Sweep Matching Users' Guide" (in preparation/URL?)
- [4] Abel, D., Devereux, D., Power, R., Lamb, P. "An  $O(N \log M)$  Algorithm for Neighbours Evaluation in Astronomical Archives" CSIRO ICT Centre Technical Report (in preparation).
- [5] URL for source distribution.
- [6] XMM Newton: <http://xmm.vilspa.esa.es>
- [7] SUMSS: <http://www.astrop.physics.usyd.edu.au/sumsscat>
- [8] Tycho-2 Catalogue: <http://www.astro.ku.dk/~erik/Tycho-2>
- [9] 2MASS: <http://www.ipac.caltech.edu/2mass> and <http://pegasus.phast.umass.edu>.
- [10] USNO Astrographic Catalogues: <http://ftp.nofs.navy.mil/projects/pmm/catalogs.html>

## 1.4 Overview

This document is organised as follows. Section 2 outlines the catalogues used and how they were prepared for plane sweep matching. Section 3 overviews the catalogue cross matching, the variations of its implementation and summaries the benchmarking performance. Section 4 does the same for nearest neighbour evaluation. We finish with some conclusions in Section 5 and mention possible further work to be done in this area. The appendices provide further details on various aspects of the implementation and benchmarking environment for those interested.

## 2 Data Preparation

The plane sweep matching algorithm requires the input catalogues to be sorted by declination and only needs the spatial description of an object (it's location in right ascension and declination) with associated errors. The maximum errors also needed for the catalogue cross matching, although no error information is needed for determining neighbours. This section reviews the catalogues used and how they were prepared for the purposes of the benchmarking tests.

### 2.1 Source Catalogues

The following is a brief summary of the catalogues used.

Catalogue	# records	Size on disk	# files	Content	Records	Null
1XMM	56,711	215M	2	ASCII	Variable	INDEF
SUMSS	134,870	18M	1	ASCII	Fixed	---
Tycho2	2,539,913	504M	1	ASCII	Fixed	
2MASS	470,992,970	144G	92	ASCII	Variable	\N
USNO A2	526,280,881	6G	24	Binary	Fixed	0
USNO B1	1,045,175,762	78G	1800	Binary	Fixed	0

**Figure 1: Sample catalogue descriptions**

Some catalogues use a fixed record structure where the data for each field is located at specific columns within the record (noted as "Fixed" in the table above). Others use a delimiter character to separate fields and thus have a variable record length ("Variable" above). For example, XMM1 uses a space character to delimit fields and places string data into quotes (so they may contain spaces), while 2MASS uses a '|' as the delimiter character. A field that contains no value (a NULL) may be depicted with a special string or number, as indicated in the last column of the table above, or simply be missing (as is the case with Tycho2).

### 2.2 Files

The catalogue files are processed by C++ programs that read the file contents and write only the necessary data for catalogue cross matching to a file. The format of the output point files is:

```
id    RA    RA_err    Dec    Dec_err
```

The id field is a unique identifier for the record. This is stored as an 8 byte integer (long long in C++) to accommodate large catalogues. For example, the USNO catalogues are provided as a collection of files split into regions partitioned by declination. Two identifiers (record number within the file and region number) can be used to uniquely identify each record in the catalogue. These two numbers are combined into a single unique id.

All output coordinate data is in decimal degrees, both coordinates and their errors.

It is assumed the input catalogues are organised as a sequence of records, each describing an object. Each record is read in turn and only the above information retained. This data is placed into a contiguous array where each element of the array contains the required point information. A consequence is that the total number of records must be known before hand. Once all the data has been read, it is sorted by

declination, using the C standard library's `qsort` function. Then the data is written to a file for later cross match processing. The data may be written as either ASCII text (using C `printf`) or as binary (using C `fwrite`).

Note sorting is done entirely in memory. This is possible since the catalogues chosen are partitioned into regions of disjoint declination strips. The USNO A2 and B1 datasets are provided as 24 and 1800 files respectively, partitioned by equal zones of dec. That is, strips of 7.5 and 0.1 deg wide. This makes the files manageable as far as reading into memory and sorting. After sorting, the files remain separated, although they could have been combined into a single large file. The plane sweep matching code has C++ classes that present a collection of "split files" as a logical single file.

Note also that the 2MASS data is similarly partitioned by declination zones, but the files themselves are *not* disjoint by declination. Whereas the USNO catalogues are strictly partitioned by declination, producing files of differing numbers of records each, the 2MASS files are partitioned so there are a uniform number of records in each (all except the last file). Consequently, the files overlap in declination. This is "fixed" when reading the files by the plane sweep code by always having two files open and returning the next lowest object (by declination).

We may not be so lucky with other catalogues, and other avenues of efficient sorting pursued, for example sort/merge algorithms. There is obviously the potential for parallel processing to be used here.

The three distinct phases of reading, sorting and writing the data are all performed by a single program. A break down of these times for binary and ASCII files are given in the tables below. The times presented are rounded to the nearest unit (m for minutes and s for seconds) and are the same for each row.

Catalogue	Disk	Total		Read		Sort		Write	
		elap	cpu	elap	cpu	elap	cpu	elap	Cpu
1XMM	2M	2.0s	1.4s	1.0	1.2	1.0	0.2	0.0	0.0
SUMSS	5M	2.0s	1.3s	1.0	0.8	1.0	0.5	0.0	0.0
Tycho2	93M	34s	33s	23	22	10	9	1	1
2MASS	18G	106m	95m	70	59	33	32	4	4
USNO A2	20G	45m	43m	8	6	32	32	6	6
USNO B1	39G	95m	80m	26	11	57	57	12	12

**Figure 2: Decoding catalogues, sorting by declination and saving as binary files**

Catalogue	Disk	Total		Read		Sort		Write	
		elap	cpu	elap	cpu	elap	cpu	elap	cpu
1XMM	4M	3.0s	2.3s	2.0	1.7	1.0	0.2	0.5	0.5
SUMSS	8M	3.0s	2.5s	1.0	0.8	0.0	0.6	2.0	1.2
Tycho2	152M	55s	53s	23	21	9	9	23	22
2MASS	32G	172m	166m	65	59	32	32	75	75
USNO A2	33G	121m	120m	5	5	32	32	83	82
USNO B1	63G	244m	232m	22	11	58	57	164	164

**Figure 3: Decoding catalogues, sorting by declination and saving as ASCII files**

The only difference in the above two sets of results is how the files are written to disk, which can be seen by the differences in the write times recorded above.

When creating a file, the maximum right ascension and declination errors are calculated. This number is required as input to the catalogue cross matching algorithm described in [1].

### 2.3 Max Errors

The following table lists the max RA and Dec errors for each of the catalogues in degrees.

Catalogue	Max RA error	Max Dec error
1XMM	0.0156153191	0.0156153191
SUMSS	0.0055833333	0.0061666667
Tycho2	0.0000508333	0.0000511111
2MASS	0.0003361111	0.0003361111
USNO A2	0.0000555556	0.0000555556
USNO B1	0.0002775000	0.0002775000

**Figure 4: Maximum Standard Deviations.**

The following is a summary of the fields containing this information in the tested catalogues.

1XMM has a single field RADEC\_ERR being the statistical  $1\sigma$  error on the source position in arcseconds. This is used for both the RA and Dec error.

SUMMS, Tycho2 and USNO B1.0 all have separate error fields for the RA and Dec.

2MASS has three fields recording the semi-major and semi-minor axis lengths of the one sigma position uncertainty ellipse and the position angle on the sky of the semi-major axis of the error ellipse. We approximate this as a circle, using the semi-major axis length as the radius.

There are no errors supplied with the data for USNO A2.0. The value of 0.2 arc seconds was used for testing purposes and needs to be revised in consultation with someone who knows the data better.

### 2.4 Id field

The output of the catalogue cross matching is a sequence of id pairs, being the unique identifier for the matched objects. The catalogues rarely give a single id as a unique reference for the objects. While this is the case for 1XMM, Tycho2 has three ids combining to form a unique id, 2MASS uses a string encoding the objects position into a unique reference: the others have no identifiers at all.

In order to locate the full object description, the sequence of the object's description in the original source catalogue is used. When the catalogue is provided as a collection of files, a file number is also included. As explained previously, for USNO A2.0, the files are partitioned into zones, and so two ids can be used to locate any object description.

### 2.5 Reading Test

The Plane Sweep Matching library includes classes to read catalogue files having the previously described file structure:

```
id    RA    RA_err    Dec    Dec_err
```

To test these files can be read correctly, and to determine a baseline for reading a catalogue from start to finish, the test program `read_file` can be used. This program simply reads each record one at a time calculating the maximum standard deviation encountered, and checks that the declination of each record is not less than the one before it. The results of running this program on the collection of processed catalogues is:

Catalogue	Num records	Max SD error	CPU time	Elapsed time
1XMM	56711	0.0156153	0.04s	0.0s
SUMSS	134870	0.00616667	0.1s	0.0s
Tycho2	2430468	0.0000511111	1.55s	2.0s
2MASS	470992970	0.000336111	4.9m	5.0m
USNO A2	526280881	0.0000555556	5.3m	5.5m
USNO B1	1045175762	0.0002775	10.3m	10.7m

**Figure 5: Reading binary files.**

Catalogue	Num records	Max SD error	CPU time	Elapsed time
1XMM	56711	0.0156153	0.32s	1.0s
SUMSS	134870	0.00616667	0.83s	1.0s
Tycho2	2430468	0.0000511111	14.17s	14s
2MASS	470992970	0.000336111	50.4m	50.7m
USNO A2	526280881	0.0000555556	53.1m	53.4m
USNO B1	1045175762	0.0002775	108.5m	109.0m

**Figure 6: Reading ASCII files.**

These figures concur with the results previously reported for these catalogues, except for the Tycho2 catalogue. This has some missing values for the mean spatial location of objects and so these records are ignored. The ASCII files take longer to read since they are larger on disk (refer to Figure 2 and Figure 3) and for the string manipulation of their contents. Note the use of ASCII files is a convenience for debugging purposes, it is easier to produce sample data for testing purposes, and wont be considered further in this report.

Note the preparation of a catalogue by reading, sorting, then writing out the relevant information as a separate step prior to performing the plane sweep matching has only been done as a convenience. Since this step needs to be performed before the matching proper, it was easier and quicker for the purposes of benchmarking to do this once at the start. Then the sorted, trimmed catalogues could be used as input for successive tests of the matching procedure using different algorithm parameters. For an end to end analysis of matching, the catalogues could be read into memory, sorted then passed directly to the matching procedure and so avoiding the extra write/read steps we have incurred.

### 3 Catalogue Cross Match

The C++ implementation of plane sweep matching is outlined in [3] and is summarised in Appendix B. Catalogue cross matching is described in detail in [1] and [2] and a fragment of the implementation is presented in Appendix C. The program used to benchmark the catalogue cross matching can be found in the source distribution [5] (the program `cm_file.cpp`). This program has a number of options that control algorithm parameters:

- `z_alpha` value: the value that puts a probability of  $\alpha$  into the tail of a standard normal distribution, thereby putting a probability of  $(1-2*\alpha)$  into the interval from  $-z\_alpha$  to  $z\_alpha$ . We have used a `z_alpha` value of 1.96 which puts a probability of 95% into the central interval. In our problem, this corresponds to a 95% confidence that two objects are not spatially coincident.
- Indexed active list or “simple” list. The indexed active list is indexed on RA to allow fast access in terms of RA. The simple list is a queue maintained by Dec and must be searched in full. The simple active list is only used for debugging purposes, to verify the indexed active list is reporting the same results. We have only used the indexed active list in our benchmarking.
- The refine chain to use. Initial tests were done with a spherical bounding box intersection test. Turns out this identifies few “false drops”. The angular separation refine accepts the candidate pairs whose angular separations are such that the likelihood that the two records represent the same source is greater than some given threshold (defined by the `z_alpha` value). Since the refine steps can be chained, the options here are to use the computationally cheaper bounding box then the angular separation; or alternatively, just use the angular separation. Our testing has shown that it is best to only use the angular separation refine.
- The filter method to use. The generic dec plane sweep algorithm (Appendix B) can be used, or the more specialised cross match one (Appendix C). Again, the use of alternate implementations have been used as a cross check to verify the code is working correctly as well as being an algorithmic alternative with differing computational performance. Our testing has shown the specialised cross match filtering performs better than the generic dec plane sweep. Both methods produce the same results, as can be verified by reader using the supplied code [5].

The tests are performed as a pair wise comparison of the six test catalogues, each prepared by extracting the spatial description of the objects with their associated errors, sorted and written as binary files. This processing has been described in Section 2. The script `cm_all_tests.csh` was used to run the tests on the machine described in Appendix A.

The following sections tabulate the results for the benchmarking of catalogue cross matching the six catalogues against each other, using a 1.96 `z_alpha`, indexed active list, angular separation refine and cross match specific filter. Note that in our testing we only count the number of matches: the details of which objects are matched are not recorded to a file (although this can be easily accommodated in the code).



The tables below record the number of objects that initially fail to be matched using the plane sweep (the "A" catalogue being the one common to all the tests in the table, for example in the first table N/M A refers to the catalogue 1XMM and N/M B the other catalogue it is matched against). The filter candidates are the number of candidate pairs found by the plane sweep filter. The refine drops the number of candidate pairs that fail the refine condition. The refine candidates is the final count of candidate pairs: the number of objects from the two input catalogues that we consider as warranting further investigation, based on the objects spatial location.

The mean and max columns refer to the sizes of the active list when a test object is compared with those in the active list during the plane sweep filter.

### 3.1 1XMM

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	1.0s	1.0s	0	0	667069	606184	60885	81	342
SUMSS	1.0s	0.3s	56619	134807	2120	2028	92	23	305
Tycho2	4.0s	4.2s	56653	2430409	14085	14025	60	22	299
2MASS	13.9m	13.8m	54541	470990423	4000487	3997889	2598	24	300
USNO A2	14.9m	14.8m	54611	526278735	2545774	2543555	2219	23	300
USNO B1	30.9m	30.3m	52103	1045170444	6024872	6019355	5517	23	300

### 3.2 SUMSS

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	1.0s	0.2s	134807	56619	737	645	92	117	221
SUMSS	1.0s	0.5s	0	0	136404	1526	134878	83	164
Tycho2	3.0s	2.5s	134803	2430400	4210	4142	68	57	137
2MASS	8.6m	8.5m	119250	470975187	537318	519533	17785	59	137
USNO A2	10.0m	9.9m	106183	526247587	1109460	1076164	33296	57	137
USNO B1	19.1m	18.6m	94150	1045111603	1902488	1838323	64165	59	137

### 3.3 Tycho2

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	4.0s	3.9s	2430409	56653	253	193	60	502	1206
SUMSS	3.0s	2.6s	2430400	134803	343	275	68	197	476
Tycho2	8.0s	7.7s	0	0	2443550	0	2443550	5	19
2MASS	14.5m	14.4m	559310	469124481	2384282	512964	1871318	14	41
USNO A2	16.0m	15.9m	1499714	525353170	1324503	393747	930756	4	21
USNO B1	32.6m	32.1m	1	1042733504	2462964	7532	2455432	18	46

### 3.4 2MASS

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	67.2m	67.1m	470990423	54541	63982	61384	2598	103184	226766
SUMSS	33.1m	32.9m	470975187	119250	57410	39625	17785	47381	108385
Tycho2	49.6m	49.5m	469124481	559310	2430417	559099	1871318	4278	7429
2MASS	92.4m	92.1m	0	0	477852377	6058045	471794332	5564	9602
USNO A2	90.7m	90.5m	295699768	351016933	325598947	150296000	175302947	4624	7485
USNO B1	148.7m	148.3m	280032290	848377342	482563209	285598003	196965206	5726	9494

### 3.5 USNO A2

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	63.2m	63.1m	526278735	54611	24711	22492	2219	112312	308395
SUMSS	36.3m	36.1m	526247587	106183	104968	71672	33296	57048	136247
Tycho2	24.4m	24.2m	525353170	1499714	1111142	180386	930756	947	2037
2MASS	81.7m	81.4m	351016933	295699768	234859609	59556662	175302947	3480	8516
USNO A2	47.3m	47.1m	0	0	526283471	298	526283173	1244	2339
USNO B1	136.2m	135.8m	110853527	610894351	512665012	76741634	435923378	4206	7922

### 3.6 USNO B1

Catalogue	Time		N/M A	N/M B	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu							
1XMM	252.3m	251.9m	1045170444	52103	89926	84409	5517	223291	468456
SUMSS	108.4m	108.1m	1045111603	94150	200810	136645	64165	100065	252001
Tycho2	199.5m	199.2m	1042733504	1	2473691	18259	2455432	7743	14363
2MASS	238.7m	238.4m	848377342	280032290	445789157	248823951	196965206	10629	20683
USNO A2	200.2m	198.8m	610894351	110853527	582111986	146188608	435923378	8360	14492
USNO B1	281.4m	281.0m	0	0	1339665262	176770886	1162894376	11355	20012

## 4 Nearest Neighbour Matching

Nearest neighbour matching is described in detail in [4] and a code fragment of the C++ implementation is presented in Appendix D. The program used to benchmark the nearest neighbour matching can be found in the source distribution [5] (the program `nn_file.cpp`). This program has a number of options that control algorithm parameters, all of which are the same as for cross matching described above, except for the first listed below:

- The maximum distance, in arc seconds, within which two objects are considered neighbours. The objects spatial location is assumed to be correct, that is, no location errors are considered when determining a match.
- Indexed active list or “simple” list. As for cross matching..
- The filter method to use. As for cross matching.

Note there is no choice of what refine option to use: the great circle distance measured as an angle is compared to the maximum distance provided. A spherical bounding box refine could be used before this, as provided for cross matching, but this has been shown to not be effective and so is not available as an option.

The tests are performed upon a single catalogue. We don't need the spatial location error details, but the files prepared for cross matching have been reused instead of recreating them without this extra information. The script `nn_all_tests.csh` was used to run the tests on the machine described in Appendix A.

The following sections tabulate the results for the benchmarking of nearest neighbours matching for the six catalogues, using max distances of 1, 5, 15, 30, 45 and 60 arcseconds, an indexed active list, and nearest neighbour specific filter. Note that in our testing we again only count the number of matches.

The tables below record the number of objects that fail to be matched using the plane sweep (N/M). The filter candidates are the number of candidate pairs found by the plane sweep filter. The refine drops is the number of candidate pairs that fail the refine condition. The refine candidates is the final count of candidate pairs: the number of object pairs within a catalogue being neighbours within the maximum distance threshold provided.

As before, the mean and max columns refer to the sizes of the active list when a test object is compared with those in the active list during the plane sweep filter.

## 4.1 1XMM

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cand	Mean	Max
	elap	cpu						
1	1s	0.1s	55089	1205	214	991	0	7
5	1s	0.1s	48493	7957	459	7498	1	17
15	0s	0.1s	44920	12162	728	11434	4	33
30	0s	0.2s	35604	29070	5275	23795	9	57
45	1s	0.2s	26630	57910	10538	47372	13	76
60	0s	0.3s	20157	95017	17323	77694	18	99

## 4.2 SUMSS

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cands	Mean	Max
	elap	cpu						
1	0s	0.3s	134870	0	0	0	0	8
5	1s	0.2s	134862	4	0	4	3	17
15	0s	0.2s	134830	26	6	20	11	37
30	0s	0.3s	134698	109	23	86	23	60
45	0s	0.3s	134412	530	298	232	35	78
60	0s	0.3s	132233	2193	850	1343	47	97

## 4.3 Tycho2

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cands	Mean	Max
	elap	cpu						
1	5s	5.2s	2417378	6546	1	6545	4	19
5	6s	5.8s	2415614	7658	228	7430	22	57
15	6s	6.4s	2406586	13613	1594	12019	67	130
30	8s	7.3s	2371751	36512	6612	29900	135	226
45	8s	7.3s	2313678	75963	15306	60657	203	324
60	8s	7.9s	2236328	130358	26915	103443	271	413

## 4.4 2MASS

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cands	Mean	Max
	elap	cpu						
1	32m	31m	469523713	850380	112742	737638	954	1666
5	66m	65m	416405871	45914760	16129812	29784948	4783	7794
15	117m	117m	140114515	666914168	149646880	517267288	14357	22877
30	182m	181m	44102276	2757829704	598319340	2159510364	28712	45640
45	272m	272m	17303150	6240581041	1344614417	4895966624	43068	68233
60	402m	402m	6998825	11112999498	2388833195	8724166303	57424	90649

## 4.5 USNO A2

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cands	Mean	Max
	elap	cpu						
1	37m	37m	526261601	969824	960184	9640	1123	2132
5	78m	77m	469917761	40889182	10063258	30825924	5608	9892
15	128m	128m	177230593	588136111	135112693	453023418	16814	29328
30	188m	188m	43740360	2518709883	552328392	1966381491	33623	58204
45	273m	273m	13632112	5726645794	1238447639	4488198155	50433	87137
60	406m	401m	4303218			8011509260		116087

## 4.6 USNO B1

Max Distance	Time		N/M	Filter cand	Refine drops	Refine cands	Mean	Max
	elap	cpu						
1	91m	90m	994590835	34897623	9036969	25860654	2060	3780
5	214m	214m	651989429	356491410	50120384	306371026	10280	18103
15	339m	338m	190848395	2459715760	512184367	1947531393	30822	53419
30	551m	550m	29386840	9808473524	2101523088	7706950436	61635	106306
45	861m	859m	3893018	21992477605	4700760309	17291717296	92448	159375
60	1284m	1284m	394387	38997682544	8334398845	30663283699	123261	212185

## 5 Conclusions

Cross matching two catalogues consisting of half a billion objects can be achieved in around 4 hours elapsed (2.5 hours to prepare the catalogues and 1.5 to do the matching itself). When one of the catalogues consists of a billion objects the total time taken is about 6 hours elapsed.

Evaluation of nearest neighbours for a catalogue of one billion objects takes 3 to 23 hours, depending on the distance threshold used (1 – 60 arcseconds). For half a billion objects the range is 2 to 8 hours.

These elapsed times can be reduced by preparing the catalogues in parallel or by utilising existing catalogues that can be accessed in declination order. When this is the case, the cross matching can be achieved in 1.5 to 2.5 hours, depending on the sizes of the catalogues used. Similarly reduced times for neighbours evaluation can be seen from the results of Section 4.

We have trialled different algorithm implementations for performing plane sweep matching, most notably filter processing, the type of active list used and various refine chains. We have only documented the fastest options here, but the code distribution can be used to experiment with these alternatives. These different implementations also provide a means of ensuring the code is working correctly:

regardless of the alternatives used, the final results should always be the same, and this has been the case for all the tests we have performed.

There are a couple of sanity checks to be mindful when running the benchmarking tests:

- The final refine results should be independent of the order the catalogues are cross matched. For example, testing Tycho2 with USNO B1.0 should give the same results as testing USNO B1.0 with Tycho2.
- The number of self join matches should always be greater than or equal to the number of objects in catalogue.
- The number of initial filter rejects doing a self join should always be 0.
- The number of filter candidates should equal the sum of the refine candidates and the refine ("false") drops.

While these considerations may seem obvious, they were invaluable when checking the code was producing plausible results. We especially had problems with the commutability criteria: using the optimiser during compilation impacted here as explained in Appendix E.

The reader should be mindful of the following when interpreting our results:

- We have not recorded to disk the final candidate pairs and doing so will incur cost on the times reported. The code supports this functionality, but we did not do so for the benchmarking tests.
- The overall elapsed benchmarking times could be reduced by pre processing the input catalogues and using them in memory as input to the plane sweep matching. We have been mostly concerned with matching performance in the presentation of our results in Sections 3 and 4.
- It is more efficient to pass the smaller catalogue through the active list and use the larger catalogue as the source of test objects to match against the active list. For instance compare the times to cross match the Tycho2 catalogue with the USNO B1 (34 minutes) with the other way around (over 3.5 hours).
- We believe all the catalogues used have object coordinates specified in J2000 at the epoch 2000, except for USNO A2.0 which uses the epoch of the plate. In order to sensibly cross match, for example, USNO A2.0 with B1.0, the star positions stated in B1.0 should have proper motions applied to take its positions to the epoch of the object to which it is being matched. We have not done this. For similar reasons, when astrometric catalogues are used, the matching procedure should take into account the errors in proper motion in conjunction with the stated positional errors when matching objects at different epochs.
- The USNO A2.0 catalogue does not include position errors. We have adopted the value of 0.2 arcseconds for the purposes of cross matching.

There are a number of further avenues that can be explored with this work. The implementation could be parallelised. New matching algorithms applied, for example cluster identification. The pplane sweep performance is sensitive to the size of the

active list. The performance characteristics could be further explored and there is scope for improved implementation in this area. Use other catalogues: we are currently preparing to use a copy of the SuperCOSMOS data. Use catalogues stored in a database environment. Seek collaboration from astronomers to perform further refinement processing to identify pairs of objects based on astronomy concepts, and not just spatial proximity.

As a final note, some catalogues are published with a list of the nearest matches from other catalogues. For example 1XMM includes a list of up to 9 of the closest matches found for the 2MASS and USNO A2.0 catalogues. This can be used as a comparison for our results.

## **6 Acknowledgments**

This work would not have been possible without access to large astronomy catalogues. The authors would specifically like to thank David Monet and his colleagues at the USNO for providing a copy of the USNO B1.0 catalogue and to Nigel Hambly for access to the SuperCOSMOS data (in progress at time of writing).

## Appendix A System Configuration

The plane sweep matching benchmarks have been performed on a machine with the following configuration:

- Dell PowerEdge 2650 server
- Dual Pentium Xeon 2GHz CPUs
- 2Gb main memory
- 5 x RAID5 storage arrays using 10K RPM UltraSCSI 160 disks (maximum transfer rate between the RAID5s to the server is 160MB/s and each RAID box is on its own SCSI channel):
  - 1 x 5-way RAID5, 70GB disks, usable storage ~0.3TB
  - 2 x 14-way RAID5, 70GB disks, usable storage ~0.9TB each
  - 2 x 14-way RAID5, 140GB disks, usable storage ~1.8TB each

The software has been compiled using g++ (version 2.95.4) on Debian Linux 2.4.20.



## Appendix B Dec Plane Sweep Algorithm

The plane sweep matching code was originally written to perform catalogue cross matching. Then we included a nested loop filter as a baseline comparison for debugging purposes. Then neighbours was introduced and we are considering other matching algorithms as well (notably cluster identification).

With this evolution of the code, we decided to separate the filter processing (identifying candidate pairs) from the matching algorithm (the definition of a match). So we have a filter that uses a matcher. There are currently two filters: DecPlaneSweep and NestedLoop. The nested loop should only be used on very small catalogues. The generic plane sweep algorithm in C++ is:

```
Object const * testObject = matcher->nextTestObject();
Object const * activeObject = matcher->nextActiveObject();

/* Loop: for as long as there are more test objects */
while (testObject != 0)
{
    double lowerBound = matcher->getLowerBound(testObject);
    double upperBound = matcher->getUpperBound(testObject);

    // remove from active list objects below lower bound
    matcher->flushActiveObjects(lowerBound);

    // populate the active list with candidates
    while (activeObject != 0 &&
           activeObject->getDec() <= upperBound)
    {
        if (activeObject->getDec() < lowerBound)
        {
            // no need to put it in - it cannot match
            matcher->reportActiveNoMatch(activeObject);
            delete activeObject;
        }
        else
        {
            matcher->addActiveObject(activeObject);
        }
        activeObject = matcher->nextActiveObject();
    }

    matcher->test(testObject);
    testObject = matcher->nextTestObject();
}

// no more test objects, but may still be some in active list
matcher->flushActiveObjects(activeObject);
```

The idea is that the DecPlaneSweepFilter can be used with any matching algorithm. To achieve this, the matching functionality has been abstracted into an interface and a specific matcher (CrossMatch or Neighbours) must provide the implementation for

it. This way the same matcher can be used with different filters to achieve the same result, although the computational cost will vary. This worked well for cross match, but less so for neighbours since there is only a single input catalogue and the generic algorithm assumes two. The neighbours matcher does support the generic interface, but it is less elegant and possibly more difficult to understand.

As well as supporting the generic matcher interface required by the filter, a matcher may implement an optimised filter method. Note that if this method is used, then the generic filter processing (eg plane sweep or nested loop) is *not* used. The matcher specific filter is used for benchmarking, the generic option is for further validation that the matching is working correctly.

The following two appendices detail the cross match and nearest neighbour optimised filter algorithms using C++.

## Appendix C Cross Match Algorithm

The following is the C++ code used to perform a plane sweep specifically for cross matching two catalogues. This is a bit more involved than the generic plane sweep algorithm as it identifies cases where one catalogue can be read ahead in order to "catch up" to the other one.

```
double lowerBound = 0.0;
double upperBound = 0.0;

Object const * activeObject = nextActiveObject();
Object const * testObject = nextTestObject();

if (testObject != 0)
{
    lowerBound = getLowerBound(testObject);
    upperBound = getUpperBound(testObject);
}

while (testObject != 0)
{
    if (activeObject == 0 && activeList->isEmpty())
    {
        /* There's nothing left to test our objects against. */
        raceThroughProducer(testProducer, testObject, uTestConsumer);
        break;
    }

    if (activeObject == 0) // know activeList isn't empty (see above)
    {
        test(testObject, lowerBound, upperBound);
    }
    else
    {
        /* Both object producers have more objects to offer. We need to
        * decide which object to handle next. */
        if (activeObject->getDec() <= upperBound)
        {
            /* The next active object could match the current test object.
            * Handle it first. */
            if (activeObject->getDec() >= lowerBound)
            {
                /* The next test object is reasonably close, so it is possible
                * for this active object to match something.
                * Add it to the active list. */
                activeList->pushBack(activeObject);
                activeObject = nextActiveObject();
            }
            else
            {
                /* The next test object is too far ahead.
                * The next active object cannot match anything, and the objects
                * currently in the active list cannot further match anything.
                * So let's clean up the active list, and then race through the
                * active objects until we reach a object that can match.
                */
                /* Empty the active list. The active list will report its
```

```
    * objects as matched or unmatched as the case may be. */
    activeList->clear(uActiveConsumer);

    /* Now race through the active objects until we reach an object
    * that can match. */
    activeObject = raceThroughProducer(activeProducer, activeObject,
                                       lowerBound, uActiveConsumer);
}
}
else
{
    /* The next active object is too far ahead.
    * Handle the next test object first */
    test(testObject, lowerBound, upperBound);
}
}
}

flushActiveObjects(activeObject);
```

It is hoped the operation of supporting methods may be understood from the comments, otherwise refer to the code in [5].

## Appendix D    Nearest Neighbour Algorithm.

The following is the C++ code used to perform a plane sweep specifically for evaluating the nearest neighbours within a catalogue. This is simpler than the generic plane sweep. If candidate pair (a,b) is reported, we don't subsequently find the candidate pair (b,a). This can trivially be included by reporting (b,a) as a match at the same time as (a,b). Note that this has not been done and the counts reported in Section 4 should be doubled if this is required.

```
Object const * testObject = nextObject(producer);

while (testObject != 0)
{
    double lowerBound = getLowerBound(testObject);

    activeList->deletePriorObjects(lowerBound, uConsumer);
    bool matched = activeList->testObject(testObject, maxDistance, pairCons);
    activeList->pushBack(testObject, matched);

    testObject = nextObject(producer);
}

// no more test objects, but there may still be some in the active list
activeList->clear(uConsumer);
```

## Appendix E Notes on C++ implementation

We have used our own string class as a wrapper to the STL `string` since the `ccmalloc` memory leak tool reports leaks that we believe not to exist. To switch between using the normal STL `string` class and one that should be `ccmalloc` friendly, define `STRING_LEAK_DEBUG` during compilation.

We have used the compiler option `-ffloat-store` when compiling with the optimiser. This is needed to ensure the tests are commutative. Without this defined, the optimiser generates code that reports different results depending if catalogue A is matched with catalogue B or vice versa. When defined, the numbers are the same. See the gcc man page for details.

Initial attempts to open a large file (greater than 2 Gbytes) proved temporarily difficult. In case this is true on other platforms, we have placed all file i/o includes and definitions into the file `FileHeader.h`. This is then included from those classes that need to perform file processing. Refer to the class `FileUtil` for details of opening large files on debian UNIX using the g++ compiler.