Quick-Look Analysis
A Study of INTEGRAL’s Automated Source Identification Algorithm

Petter Oscar Nikula
2015

Supervisor: Jukka Nevalainen
Examiners: Alexis Finoguenov
Juhani Huovelin

UNIVERSITY OF HELSINKI
DEPARTMENT OF PHYSICS

PL 64 (Gustaf Hallströmin katu 2)
00014 Helsingin yliopisto
This thesis investigates the automated near real time science analysis performed at the INTEGRAL Science Data Centre. The structure of the Quick-Look Analysis pipeline and individual analysis stages are detailed. The stage performing pattern recognition for two-dimensional coordinate lists, i.e. source identification, is tested in-depth. The lists contain sources located in a randomly selected 9° by 9° area of the sky. Using the current live version and default parameters; a simulated new source was correctly identified 98% of the time, fields with no new sources produced false detections 8% of the time. The testing reveals two separate flaws; a code error and a methodological error. The sensitivity of recognizing that a new source has been detected is reduced because of the code error. The methodological error causes the algorithm to report the detection of previously unknown sources where none exists. A possible solution is presented. New source detection was improved to well above 99% and false detections reduced below 2% with the new solution. A second methodological error causes the algorithm used to correct for the pointing error of the instrument to produce unreliable results. Fortuitously this problem is serious only for small pointing errors where the source matching algorithm is able to compensate for it.
## Contents

1 Introduction 7  
1.1 Context ........................................ 7  
1.2 This Thesis .................................... 7  
1.3 The Quick-Look Analysis ....................... 8  

2 Concepts and Theory 9  
2.1 Choosing a Language ............................ 9  
2.2 ISDC System Architecture ...................... 10  
2.3 INTEGRAL Science Data ........................ 12  
2.4 ISDC Support Libraries ........................ 14  

3 The Code 19  
3.1 QLA Processing .................................. 19  
3.2 Q_build_scw_list .................................. 20  
3.3 Q_flag_srcs ..................................... 23  
3.4 Q_set_src_fluxes ................................ 24  
3.5 Q_identify_srcs .................................. 27  
3.6 Q_match_src_fluxes ............................... 31  

4 Testing QLA Source Identification 33  
4.1 Test Setup and the Definition of Success .......... 33  
4.2 Input Parameters and Source Data ................ 37  
4.3 Source Identification Testing .................... 43  
4.4 Grid Search Testing .............................. 53  

5 Summary 61  
5.1 Concluding Remarks ............................. 62  

Appendix I 67  
Appendix II 73  
Appendix III 77  
Appendix IV 81  
Appendix V 91  
Appendix VI 111
Abbreviations

API Application Programming Interface
ARF Ancillary Response File
CTS Coding and Testing Standards
DAL Data Access Layer
DAL3CAT Data Access Layer 3 Catalogues
DOL Data Object Locator
ESA European Space Agency
FITS Flexible Image Transport System
GRB Gamma-Ray Burst
HEASARC High Energy Astrophysics Science Archive Research Center
IBIS Imager on-Board the INTEGRAL Satellite
INTEGRAL INTErnational Gamma-Ray Astrophysics Laboratory
IRAF Image Reduction and Analysis Facility
ISDC INTEGRAL Science Data Centre
ISGRI INTEGRAL Soft Gamma-Ray Imager
ISSW Instrument Specific SoftWare
JEM-X Joint European X-Ray Monitor
NASA National Aeronautics and Space Administration
OBT OnBoard Time
OMC Optical Monitoring Camera
PICsIT Pixilated Caesium-Iodide Telescope
PIL Parameter Interface Layer
PSLA Point Source Location Accuracy
QLA Quick-Look Analysis
RIL Reporting Interface Layer
RMF Redistribution Matrix File
SPI SPectrometer for INTEGRAL
TOO Target Of Opportunity
XSPEC An X-Ray Spectral Fitting Package
1

Introduction

“Home is behind, the world ahead, and there are many paths to tread through shadows to the edge of night, until the stars are all alight.”

– J.R.R. Tolkien, The Fellowship of the Ring

1.1 Context

INTEGRAL

The International Gamma-Ray Astrophysics Laboratory (INTEGRAL) mission was proposed to the European Space Agency (ESA) in 1989 by an international consortium of high energy astrophysicists [1]. In June 1993 INTEGRAL was adopted as a medium-sized science mission in ESA’s “Horizon 2000” programme. The satellite is dedicated to fine imaging and spectroscopy in the 15 keV to 10 MeV energy range with concurrent source monitoring in X-ray\(^1\) and optical\(^2\) ranges. It was launched on October the 17th 2002 from Baikonur Cosmodrome in Kazakhstan. The original mission was planned to last 2.2 years but has since been extended several times. The latest extension is until December 2016. Estimates of the remaining fuel indicates that science operations should be possible beyond 2020.

ISDC

It was recognized that the mission would produce vast quantities of data that requires complex analysis. In recognition of this it was decided to establish a center where the data would be received, analyzed, archived and distributed. The INTEGRAL Science Data Centre\(^3\) (ISDC) was formed by a consortium of European and US institutes who responded to ESA’s call to develop and operate the facility [2].

1.2 This Thesis

My aim is to describe, discuss and test a set of software tools running at ISDC called the Quick-Look Analysis (QLA). The QLA is but a small part of the entire INTEGRAL software suite. I

---

\(^1\) 3-35 keV

\(^2\) V-band, 550nm

\(^3\) In Versoix, under the auspices of the Astronomy Department of the University of Geneva
choose to limit my scope to that set of tools because I wrote the initial versions of them during two years (2000-2001), giving me subject matter expertise in this field. Some of the tools have subsequently been upgraded by other developers. I will describe the latest version of the tools which are in use at ISDC.

I will introduce the QLA in this Chapter. The second Chapter (see page 9) depicts the choices made about the ISDC environment and the technical solutions that lead to. The Chapter also serves as a guide to the concepts and terminology to be used in later Chapters. The third Chapter (see page 19) describes how the QLA tools work. I will test one of the QLA tools, using simulated data, in the fourth Chapter (see page 33). The fifth, and final, Chapter (see page 61) sums up the work. Additionally there are six appendices describing the QLA code in detail.

1.3 The Quick-Look Analysis

The QLA is a fully automated scientific analysis that takes place in near real time\(^4\). The core functionality consists of identifying the recently found sources\(^5\), trying to match them with existing catalogue sources\(^6\), and detecting unexpected changes in their fluxes. Of particular interest are found sources that cannot be matched with catalogue sources, i.e. new sources, as well as sources with large changes in their fluxes.

In gamma-ray astronomy there is a natural focus on rapidly variable sources. The timescale for which a system can change is proportional to the size of the system because information cannot be exchanged between individual members of the system faster than the speed of light allows. Many types of gamma-ray emitting processes take place in conjunction with compact objects\(^7\) and are therefore inherently capable of rapid variability. In addition, several gamma-ray emitting processes do not repeat themselves, they are transient\(^8\). Unexpected variability of a source is always of interest to astronomers but presents a challenge for observations, especially short duration transients. The quicker the detection of, and information about, such an event can be disseminated to the community of astronomers the richer the data obtainable are as there is more time for follow up observations by different instruments.

The detection process is carried out by the operations team at ISDC during working hours. The scientist on duty receives images as they become available and alerts for interesting sources generated by the QLA. In case a new source is very bright, with a detection significance > 30\(\sigma\), an automated SMS text message is sent to the operations coordinators mobile phone to allow for an instant response outside working hours. Thanks to this system it is possible to report about interesting new sources within just a few hours. The majority of the circulars\(^9\) for new and transient detections have come about by using the QLA\(^3\).

---

\(^4\) The “near” label expresses that data from INTEGRAL are not instantaneously available at ISDC for analysis. The QLA processing itself takes place in real time, as soon as input data become available.

\(^5\) I will use this term to indicate the fresh observational data from astrophysical sources coming in from the spacecraft.

\(^6\) This indicates astrophysical sources that have been previously identified and for which data exist in the ISDC archives.

\(^7\) Supermassive black holes (0.001-400 AU \(\propto\) 1s-days), stellar black holes (\(\sim\) 30km \(\propto\) 0.1ms) and neutron stars (\(\sim\) 10km \(\propto\) 30\(\mu\)s)

\(^8\) Supernovae and Gamma-Ray Bursts (GRB) are examples of transients with the shortest duration GRB’s lasting just 0.2 seconds

\(^9\) http://isdc.unige.ch/integral/science/circulars
2

Concepts and Theory

“Short cuts make long delays.”
– J.R.R. Tolkien, The Fellowship of the Ring

In this Chapter I will explain the fundamental choices made about the structure of the work with creating the scientific analysis software for INTEGRAL. These choices were based on a number of underlying drivers. The knowledge base of the people working on the project led to selecting two procedural languages for code development. The characteristics of the supported programming languages, the distributed nature of code development and a previous working example (the image reduction and analysis facility, IRAF\textsuperscript{1}) led to the adoption of a batch sequential architectural style. The need to transfer ownership of code components, as well as a general need to keep the code clean and understandable, led to the adoption of a set of rules for everyone to follow, the Coding and Testing Standards (CTS).

Moving on, from the human side of the equation, to the characteristics of the INTEGRAL instruments we have other important drivers. Multiple images have to be interpolated in order to get the best possible science results. The interpolation together with the nearly universal adoption of the FITS file format within high energy astronomy led to an update of the FITS specifications to allow for additional layers of metadata to be added. And finally there was a need to separate the code developers from the underlying information infrastructure. This stems from the metadata/data construction being fragile in use as well as uncertainty about the final layout of the hardware and software handling the infrastructure. This separation takes the form of a number of support libraries that the developers call on instead of directly manipulating the underlying files.

This is by no means a complete set of drivers or decisions made at the inception of the INTEGRAL analysis software project but they are the most important ones for this work. The second purpose of this chapter is to introduce the terminology and concepts necessary for treating the issues in the following chapters.

2.1 Choosing a Language

The Fortran and C programming languages, chosen for writing the INTEGRAL software, are so called procedural languages. The choice was made in order to accommodate the astronomers.

\textsuperscript{1} IRAF is a general purpose software system for reduction and analysis of scientific data. IRAF is written and supported by the IRAF programming group at the National Optical Astronomy Observatories in Tucson, Arizona.
involved in the project who were familiar with the selected languages but not necessarily with later generations of languages. Choosing to use procedural languages to develop the INTEGRAL code impacts the level of abstraction it is possible to achieve. The level of abstraction sets a size limit on a “unit” of code that still remains understandable to the humans reading it. This, in turn, affects the general structure of the code that was chosen by ISDC.

**First generation languages** Any software could, in theory, be written as a (very) long list of instructions that are carried out in sequence. This was also how things looked like early on in software development. With every bit of the program in the same listing such monolithic code becomes unreadable and untestable quite quickly as the length and complexity of the code grows.

**Procedural languages** A powerful method to make the code more readable is to split it into smaller segments which are small enough to remain understandable by a human. These bits of code became procedures\(^2\). The idea is to split out specific tasks from the bulk of the code, making them re-usable and independent of external variables in the process. But as the length and complexity of the code increases it fragments into more and more procedures. The problem can be mitigated by adding new levels of procedure hierarchies. Such hierarchies eventually also become difficult to understand as their complexity grows. Of particular concern is the flow of data between procedures. All communication between procedures have to pass through the procedures above them in the hierarchy. This data flow eventually becomes hard for the developer to manage.

**Object-oriented languages** The problems inherent with procedural languages can be mitigated by placing all related procedures\(^3\) inside a “container” that contains the data they need, an object. Because objects are self contained they can be treated much like any other piece of data inside a program. One can hold them in arrays, sort them, build other objects that contain them, etc. This allows for increasing abstraction in a manner that is much harder to do in the procedural case.

### 2.2 ISDC System Architecture

The design of the ISDC system conforms to a batch sequential architectural style. In this architecture data flows between independent processing stages connected to each other by connectors. A stage is a unit of software that does not rely on any other part of the code for its functionality. The stage begins its operation by reading input data after which it can enrich, refine or transform the data. Once the stage has finished all its tasks it will store the results. A connector transports the data from one stage to the next stage. In the batch sequential architectural style a connector is usually an abstract concept as the stages often read their input data and store their output data directly as files on disc. But more concrete connectors, containing internal processing and data transportation logic, are certainly possible.

The stages in a batch-sequential design are often connected into a simple, linear, network. But

---

\(^2\) An alternative name for a function.

\(^3\) Now typically called methods.
one can also use more complex configurations, directed acyclic graphs\(^4\) (figure 2.1 on page 11). The key advantages of the batch sequential architectural style are that the stages can easily be modified, replaced or reused without changing other stages \([4]\). This allows the system to be broken into components that can be developed independently of each other. This type of divisibility could have been achieved by specifying suitable interfaces, if an object-oriented language had been used, without using this particular architectural style. But the choice of programming languages made the choice of this architecture, or one closely related to it, a necessity.

### 2.2.1 Connectors

Different segments of computer code can, in general, communicate with each other through two mechanisms. Either they are able to access each other directly\(^5\) or indirectly through message passing. As indicated above the architectural choice for the ISDC system was the latter. In this system there are several parallel message passing mechanisms. The science data are stored and transported in files conforming to the flexible image transport system\(^6\) (FITS) standard. Parameters\(^7\) are stored in files conforming to the IRAF parameter file\(^8\) format and messages intended for the human operators are stored in various log files.

Data transportation between stages in the ISDC system follows the passive approach of writing files to disc which are then read by the next stage. Treatment of control information\(^9\) is more complex. The ISDC system utilizes code written in a scripting language as well as the OPUS \([5]\) tool to tie together the processing stages. Therefore the concept of connectors do have a more than theoretical meaning for the system.

---

4 A set of nodes connected by directed edges where it is not possible to start from any one of the nodes and find a path back to that node through the network.  
5 Accessors to objects, some times called methods, is one such form of direct access. Other examples include function calls.  
6 The FITS file standard was developed with the aim of storing high energy astronomy data. In particular to be able to contain data, from many old missions, that had been stored in a variety of different file formats originally. Developed and maintained by the National Aeronautics and Space Administration’s (NASA) High Energy Astrophysics Science Archive Research Center (HEASARC).  
7 Control information guiding executable function. One example is where the data files the executable is supposed to read are located.  
8 A file containing the inputs needed from the system in order for a component to perform its function.  
9 All sorts of information that affects and directs processing in a stage.
2.2.2 Stages

The batch sequential architectural style provides processing stages with exceptionally clean boundaries. The stage can run to completion regardless of the presence of any other stage as long as input data for the stage exist. Apart from enhancing the modifiability of the system it also makes it easier to test in segments. Each stage can be tested entirely in isolation from the rest of the system as long as suitable input files exist\(^\text{10}\). For an external developer the project looks like an interaction with no-one but ISDC. He is provided with instructions on the desired functionality of the stage he is about to create. He writes the necessary code and tests the finished stage against test input simulating real data. Finally he delivers the code to ISDC. Although not called stages by ISDC their executables\(^\text{11}\) system building blocks conform very closely to the idealized architectural concept.

2.2.3 ISDC Architectural Terminology

The entire ISDC system could be described using the two concepts mentioned previously, connectors and stages. ISDC, however, chooses to define their architectural building blocks differently. They use four different types of components in their terminology. 1) Executables - A program whose execution is completely defined by the input data and the parameters. 2) Scripts - A set of executables, or lower level scripts, tied together with flow control instructions in a scripting language. Essentially a batch-sequential sub-system. 3) Components - A generic term for elements of software within the ISDC system; executables, scripts or pipelines. 4) Pipelines - A set of components, typically scripts, chained together via the OPUS\(^\text{12}\) system. Pipelines control the usage of system resources and the overall data flow. Using the architectural terms “stage” and “connector” gives a good degree of clarity when discussing the structure of a batch sequential system. As ISDC chooses not to use these concepts I will also have to follow suite when describing the operation of the system.

2.3 INTEGRAL Science Data

From an analysis software point of view the science instrument is (merely) a source of data. In the bigger picture the instrument and its characteristics determine what the data, and thus the software, will look like. To understand the data we need to begin by looking at the instruments on-board INTEGRAL, how they make observations, how data from those observations are structured and how data are stored and accessed in the ISDC system.

2.3.1 INTEGRALs Instruments

The two main gamma-ray instruments on-board INTEGRAL are 1) the spectrometer for INTEGRAL (SPI)\(^\text{13}\) \([6]\) and 2) the imager on-board the INTEGRAL satellite (IBIS)\(^\text{14}\) \([7]\). The IBIS instrument has two detectors; a low energy 128 \(\times\) 128 matrix, large area (\(\sim\) 2600cm\(^2\)), multilayer

\(^{10}\) The emphasis on “suitable” is strong. I will return to this topic in Chapters four and five. \\
\(^{11}\) An executable is the lowest level unit defined in the architectural design of the ISDC system. \\
\(^{12}\) A software environment developed by the Space Telescope Institute tasked with running automatic pipelines. OPUS was originally developed to handle Hubble Space Telescope science data but is also used by other instruments. \([5]\) \\
\(^{13}\) Optimized for high resolution gamma-ray line spectroscopy at 20keV - 8MeV. \\
\(^{14}\) Optimized for high angular resolution gamma-ray imaging at 15keV - 10MeV.
CdTe detector (ISGRI) and a high energy 64 × 64 matrix of caesium iodide elements with a sensitive area of 2890cm² (PICsIT). In addition the craft observes X-ray emissions\textsuperscript{15} with the two joint European X-ray monitors (JEM-X) \cite{8}. All three instruments mentioned are coded aperture mask instruments\textsuperscript{16}. Optical observations are provided by the optical monitoring camera (OMC) \cite{10}. The anti-coincidence shield protecting SPI and IBIS from background radiation has also been put into use as a gamma-ray burst monitor \cite{11}. \cite{1}

2.3.2 INTEGRALs Observation Strategy

It is not possible to get good resolution imaging from a single exposure, called a \textit{science window}\textsuperscript{17}, because the number of detectors in the detecting array is small (especially for SPI \cite{6}). For this reason INTEGRAL employs a “dithering” strategy for SPI and IBIS observations where the spacecraft does an exposure\textsuperscript{18} of the target and then moves 2°, does another exposure, moves, etc. The observation usually proceeds in either a 7-point hexagonal pattern or a 25 point rectangular pattern centered on the main target of observation. Observations of the galactic plane can use other observation patterns \cite{2}. In either case the final, full resolution, image is obtained by combining all the separate exposures.

2.3.3 Data Structure

A set of science windows that are nearby in time and space creates a logical unit called an \textit{observation}\textsuperscript{19}. Each science window can be used to improve the resolution of the area where the science windows overlap. It is worth noting that a science window does not have to be logically connected to just one observation, it can also be a part of a nearby observation. Because of this there can be many-to-many dependencies between science windows and observations. One way to store such data would be to copy a full set of related science windows into the data structure making up each observation but the data storage required for this would quickly grow large. A more elegant solution is employed by ISDC; science windows are stored once and the data describing an observation is a metadata construct. The metadata contains a description linking the science windows to the observation and notes where in the file system they can be located.

\textsuperscript{15} 3-35keV

\textsuperscript{16} At short wavelengths it is no longer possible to use lenses and mirrors to form an image because the photons would pass through such devices without being deflected. The simplest alternative is a pinhole camera but such a device blocks most of the light falling onto it. To improve on the throughput one can instead use a mask with a carefully selected pattern of holes and blocking elements. The pattern is selected in such a way that each source in the field of view will create a unique pattern of shadows and light (a \textit{shadowgram}) on the detector so sources can be distinguished from each other. One of the advantages of a coded aperture mask detector is that one obtains simultaneous measurements of the background flux (corresponding to the blocking elements) and the source plus background flux (corresponding to the holes). The disadvantages of a coded aperture mask detector are that the image has to be reconstructed (deconvolved) by applying mathematical algorithms dependent on the mask pattern to the measured data and that the throughput is still reduced compared to an instrument with no blocking elements. \cite{9}

\textsuperscript{17} This can either be a \textit{pointing} or a slew. In the first case it is a single exposure in a fixed position and in the second case it is an exposure carried out while the telescope is moving between two pointings. Usually for a duration of 2000s.

\textsuperscript{18} Henceforth the word observation/s (but not “observational” or “observed”) is taken to have this meaning.
2.3.4 Data Storage and Access

The flip side of the data storage scheme is that the logical framework surrounding observations can become quite complex. An observation can easily be part of another observation and so on. To insulate the users of the system from this complexity ISDC chose to deploy a three-layered approach; data access layers (DAL).

All science data are stored in files conforming to the FITS standard. The lowest level, 1, of DAL consists of the CFITSIO library\(^\text{20,21}\) that was expanded to cater to the needs of ISDC. DAL level 1 utilizes the FITS Hierarchical Grouping Convention \([12]\) that makes it possible to logically link many files into a common structure. Using the CFITSIO library is the first step of insulating the user from the particulars of the file structures. The application programming interface (API\(^\text{22}\)) provided by the library can handle all file access tasks.

The second level of DAL solves the data structure complexity problem by providing the abstractions data objects and data elements. A data object is a data structure that contains data elements. A data element in turn can either be another data object or a base element\(^\text{23}\).

In figure 2.2 on page 15 we can see how the correspondence between actual data and the logical structures was solved. The FITS header data units\(^\text{24}\) correspond to DAL data elements and the FITS group defined in the group header data units corresponds to data objects. One could easily imagine files 2 & 4 containing observational data and files 1 & 3 additional information for the whole that is an observation. Files 2 & 4 could now just as easily be re-used in another observation with a different logical structure.

2.4 ISDC Support Libraries

In section 2.3.4 I first mentioned the three data access layers, providing some detail about the data storage scheme employed by ISDC. I've also touched on the parameter interface layer and reporting interface layer in section 2.2.1. So far I have treated these entities only on the conceptual level, as a buffer between the technical details surrounding information transportation (through connectors) and the developer using the system (to write stages). Each of these are a collection of functions to perform tasks like; opening and closing files, reading from files, writing to files, etc. Such a collection of functions with a common theme, like file access, are called software libraries. In this section I will provide general information about several libraries that ISDC provides to support software development for their system.

\(^{20}\) A library of functions or subroutines to be called by executables or applications.

\(^{21}\) Developed by Dr. William Pence to read and write FITS-files. The library and file format are maintained by NASA’s HEASARC.

\(^{22}\) A specification of how software components interact with each other. An API usually takes the form of a software library containing functions that the user can call to perform some task. For instance, draw an object on the screen or open a file for access.

\(^{23}\) The base elements ISDC provides are atomic data structures that come in four flavors; arrays (n-dimensional sets of homogenous data), tables (tabular data structure with rows in a column), virtual tables (a combination of columns from multiple tables in a temporary table) and information sets (human readable and/or bulk data, e.g. programs, text, GIF images, postscript output). In addition to the atomic data structures there is a fifth base element, groups, that contains compound data structures.

\(^{24}\) A FITS file is comprised of segments; header data units, where the first header data unit is called the “Primary header data unit”, or “Primary Array”. The primary data array can contain a 1-999 dimensional array of 1, 2 or 4 byte integers or 4 or 8 byte floating point numbers using IEEE representations. A typical primary array could contain a 1-D spectrum, a 2-D image, or a 3-D data cube.
2.4.1 Data Access Layer (DAL)

DAL is a software library that provides access to ISDC science data files. It contains functions for all necessary file opening, access and closing tasks. It also provides tools for the developer to locate desired data in the logical tree structures of data objects. Using the second or third layer of DAL for file access is mandatory for developers in the ISDC system in order to maintain file integrity.

The term DAL indicates DAL layer 2 if referred to without specifying the layer. Layer 1 is identical to the CFITSIO library which provides functions that safely interacts directly with FITS-format files. Layer 2 is built on top of this, using CFITSIO functions, to add a layer of abstraction as described in section 2.3.4. Layer 2 provides the user with tools for manipulating data objects and data elements instead of the underlying files.

When accessing files through DAL the user provides a data object locator (DOL) instead of a normal path to the file. The DOL is not intended as an abstraction of the path, in fact it contains the path information. But DOLs allow for more generic pathing including the possibility that the file resides on a remote host, among other things [13].

Below we have a typical DAL function call in the C/C++ language. The function creates a new data object. From top to bottom the call specifies; the name of the object to be created, the access type for the object, the template from which the object will be created, a pointer in

---

25 Maintaining the file header structure intact, among other things.
26 An abstract pointer towards the physical location of a data object that takes the form of a Uniform Resource Locator; accessType://fileLocation[HDUlocation].
27 “inputParams” is a convention used by the developer indicating that the value, in the variable “OUTPUT_NAME”, has been read from the parameter file for this executable.
28 The keyword indicates that the new data object will be created on disc. Other possibilities include memory or shared memory.
29 Templates are recipes defining the structure of new data objects. The user could also create an empty data object and build up the structure manually afterward. The name of the variable, “TEMPLATE_DOL”,...
which the location of the new object will be placed\textsuperscript{30} and the current status of the system\textsuperscript{31}.

\begin{verbatim}
status = DALobjectCreate( inputParams.OUTPUT_NAME,
                       DAL_DISK,
                       inputParams.TEMPLATE_DOL,
                       &((*fileInfo).OUTPUT_FP),
                       status);
\end{verbatim}

\subsection*{2.4.2 Parameter Interface Layer (PIL)}

Besides science data files there is also a need to transmit control information to the components. Such information is used to affect processing and is of very diverse nature, largely depending on the specific component in question. Thus the components cannot exist in total isolation from each other, as in the ideal architectural model. An upstream component\textsuperscript{32} may need to transmit control information downstream which necessitates a certain degree of parallel development. Modifiability is maintained through detailed specifications of the components which predates development.

ISDC has chosen to communicate control information in parameter files. These are text files with one piece of control information (parameter) on each line. The parameter files were chosen to be IRAF compatible. A software library was also created to handle access to the parameter files. The functions PIL provide are concerned with; opening, closing, reading from and writing to parameter files [14].

\begin{verbatim}
status = PILGetString( "outputTemplate",
                     (char *) &((*inputParams).TEMPLATE_DOL));
\end{verbatim}

In the code example for DAL we saw a DOL for a template file being used as an input for the sample function. Above we can see how that DOL is read from the parameter file. The function call specifies; the name of the parameter to be read and a pointer to a character array\textsuperscript{33} where the resulting string will be placed.

\subsection*{2.4.3 Reporting Interface Layer (RIL)}

A component might need to send messages to the human operators of the system in addition to communicating with other components. This communication is handled through the API supplied by the RIL library. The exact form this communication takes, from the human perspective, is opaque to the user of RIL. But in practice we are talking about log files and messages on computer screens, as the situation warrants.

One of the main functions of RIL, from an architectural point of view, is to provide something similar to a test driven approach to software engineering. In the section about DAL (2.4.1) we

\textsuperscript{30}A pointer indicates a memory location where some data resides. The function call here tells the called function where to write the location of the new object so the calling function can retrieve the information later on.

\textsuperscript{31}Every function call in the ISDC system shall return either the value defined as ISDC_OK, if all is well, or some other value if an error has occurred.

\textsuperscript{32}Or it could be the system communicating with a component.

\textsuperscript{33}Residing inside a structure (STRUCT) in this case.
saw the concept of status being mentioned. Every function\textsuperscript{34} in the ISDC system is required to return a status code. This requirement also includes the requirement that functions are not allowed to crash uncontrollably but must be able to abort execution and return a status code even when they malfunction. The default status code is contained in the global variable ISDC\textunderscore OK\textsuperscript{35}. Any other, but typically negative, value indicates that an error of some type has occurred. These error codes are defined by the author of each component to be indicative of what type of error has taken place. Each and every time the status is able to change, and otherwise when the situation warrants it, a component is required to log a message using the RIL API. This creates a message trail throughout the execution of each component that resembles a set of automated tests on the code. This sort of messaging is typically not logged during normal operations, due to the large amount of messages it produces, but the option to do so exists.

RIL provides for sending 4 types of messages with 4 different severity levels each. These are; errors, warnings, alerts and log events. Log events and errors were discussed above. Warnings can be issued when something less than an error has taken place. Alerts are aimed purely at the human users of the system. An alert would, for instance, be issued as a wake up call to the operators of the system in case a GRB is detected so that he can take immediate follow-up action. The range of event severity is; 0 - Low interest event that does not get logged centrally, 1 - Events that are not serious but need to be logged for traceability, 2 - A serious event that does not require an immediate response, 3 - The event requires immediate human intervention \cite{15}.

\begin{verbatim}
status = PILGetString( "outputTemplate",
                     (char *) &(*inputParams).TEMPLATE_DOL);
if(status != ISDC_OK) continue;
strcpy(prText, " *outputTemplate parameter read.");
if ( chatty > Q_CHATTY_VERY_VERBOSE ) {
    RILlogMessage(NULL,Log_0, prText);
}
\end{verbatim}

Expanding on the code example for PIL we can see how the status is immediately checked after the PIL function call and execution aborted if the status is not ISDC\textunderscore OK. A message string is created if no error has taken place. The condition for the if-clause is a check to see if the desired level of logging\textsuperscript{36} includes the following RIL function call or not. If this (very low) level of log event is desired then a call to RIL to create a message is made. The parameters sent to the function are; file reference\textsuperscript{37}, the type and severity of the message\textsuperscript{38} and the text string that forms the message. Being at the lowest possible level in the status reporting hierarchy, the RILlogMessage function does not in itself return a status code.

\subsection{2.4.4 Data Access Layer, Level 3}

The highest level of DAL is built using the DAL level 2 API\textsuperscript{39}. DAL level 3 operates on data structures that are specifically defined for ISDC internal use. The third layer of DAL

\textsuperscript{34} Which includes executables as well as all their constituent elements.
\textsuperscript{35} Which contains the integer value zero.
\textsuperscript{36} As read from the parameter file.
\textsuperscript{37} Not required in this case.
\textsuperscript{38} Log event, lowest severity.
\textsuperscript{39} “Data Access Layer (DAL)” on page 15
actually consists of several software libraries tailored towards different uses. There is a general library, DAL3GEN, that largely replicates the DAL API [16]. Another DAL3 library used in this work is DAL3CAT which contains functions for manipulating catalogues and source lists [17]. Important tools when handling catalogues and source lists are masks. These are one-dimensional truth-value arrays with a length corresponding to the size of the catalogue or source list. They can be combined using any logical operation and are used to select sources. Using DAL3 libraries is not mandatory within the ISDC system but provides another layer of safety when it comes to maintaining the integrity of data structures. The user of a DAL3 library can be sure that the operation he uses will maintain not just the general data element / data object structure but also the format of the specific data structures used in the ISDC system.

```c
for(a=0; a<fileInfo.INPUT_ROWS; a++) {
    status = DAL3CATsrcGet( fileInfo.INPUT_FP,
                            a+1,
                            &qlaData[a],
                            status);

    if(status != ISDC_OK) break;
}
```

The code example I have chosen loops over a catalogue file and reads the content into a data structure residing in memory. Each DAL3 function call in the loop is followed by a status check, as required. The input parameters of the function are; A variable containing the location (DOL) of the catalogue file, the loop counter increased by one, the data structure into which the catalogue data will be read and the status.

---

40 Data structures that contain a list of celestial sources and their relevant characteristics.
41 A specialized form of a catalogue. A source list is typically a subset of a catalogue and may contain some additional data used for analysis.
42 Not to be confused with the coded aperture masks mentioned previously. The context indicates which one is in question.
3

The Code

“There is nothing like looking, if you want to find something.
You certainly usually find something, if you look,
but it is not always quite the something you were after.”

– J.R.R. Tolkien, The Hobbit

In this Chapter I describe the two QLA pipelines; what they are and how they differ from each other. This sets the context within which the QLA components exist. This is followed by a description of each of the five QLA components from the perspective of how the calculations embodied in the software are done. If the reader is interested in a description that is closer to how the functionality of the components is implemented in code then he can turn to the Appendices at the end of this thesis. The Appendices may also serve as useful companion literature to this Chapter.

3.1 QLA Processing

The QLA subsystem is a part of the near real time data processing system. It can be divided into two components; Offline Scientific Analysis and QLA processing. The former is a set of tools providing a human user with the ability to analyze any data he might be interested in. The latter is an automated analysis of the science data and the subject of this thesis. It breaks down into two main paths; the ScW QLA Pipeline\textsuperscript{1} and the OBS QLA Pipeline\textsuperscript{2}. This split is due to the different timescales involved. Whereas a new science window can be obtained from a roughly 30 minutes long exposure one has to wait several hours, or even days, to obtain enough science windows to form an observation. One wishes to analyze individual science windows in the ScW QLA Pipeline in order to detect new transient sources as quickly as possible. The OBS QLA Pipeline is used for SPI data analysis and detecting faint sources where overlapping multiple science windows is necessary.

3.1.1 The ScW QLA Pipeline

This pipeline starts as soon as a new science window has been received by the mission operations centre\textsuperscript{3}, pre-processed, and run through the NRT ScW Pipeline\textsuperscript{4}. A schematic of the ScW QLA

\textsuperscript{1}“Science window QLA Pipeline”. An automated science analysis performed on individual science windows.

\textsuperscript{2}“Observation QLA Pipeline”. An automated science analysis performed on a group of science windows.

\textsuperscript{3}Located at ESOC in Darmstadt, Germany.

\textsuperscript{4}Near Real Time Science window Pipeline.
Pipeline can be seen below (figure 3.1 on page 21). The drawing convention used is the stage - connector architecture. Scripts are also indicated in the drawing.

The first step is to create an observation group\(^5\) that consists of a single science window through the generic QLA tool/script \texttt{q\_build\_og}. The DOL to this observation group is then passed to three instrument specific scripts. One for each of the three instruments where analysis of a single science window is possible.

The scripts for IBIS\(^6\) and JEM-X\(^7\) further breaks down into two parts; either \texttt{ibis\_science\_analysis} or \texttt{jemx\_scw\_analysis}\(^8\) which are followed by the generic QLA tool/script \texttt{q\_check\_results}. Both ISSW analysis scripts contain the generic QLA tool/script \texttt{cat\_select}. The OMC script\(^9\) differs from the previous two because there is no good way to search for new sources with the OMC\(^10\). Instead a standard data reduction is performed followed by a check to see if there are significant brightness variations among the known sources.

In the IBIS and JEM-X processing the raw data is reduced\(^11\), with the help of \texttt{cat\_select}, in accordance with the specifics of the instruments. \texttt{Cat\_select} will pick out the sources that are either in the field of view of the instruments, or just outside it, to indicate which sources should be seen or might affect the exposure. The reduced science data will then be compared with catalogue data by \texttt{q\_check\_results} to see if new sources have been found or if a known source shows signs of variability in flux or hardness ratio\(^12\).

### 3.1.2 The OBS QLA Pipeline

The JEM-X instruments would not gain much in accuracy by merging raw data from multiple science windows\(^18\) so the observation pipeline is only run for the major coded mask instruments; IBIS and SPI. The pipeline is launched when a new observation group containing multiple science windows arrives. Execution is very similar to the ScW QLA Pipeline (figure 3.2 on page 22) with the main difference being that the ISSW is now able to use multiple science windows when calculating source fluxes and reconstructing the images. An additional stage has been added to create a mosaic of the images.

### 3.2 \texttt{Q\_build\_scw\_list}

*Function*: Build a group of science windows for QLA data, based on a selection by spatial and temporal proximity." \(^{[19]}\)

---

5 The logical construct containing an observation.

6 \texttt{Q\_scw\_ibis}.

7 \texttt{Q\_scw\_jemx}. Executed twice, once for each of the two detectors.

8 These scripts are not specifically indicated in the figure. They begin and end with the generic instrument specific software (ISSW) stages in the figure which signify one or many stages not described in this thesis.

9 \texttt{Q\_scw\_omc}.

10 The OMC monitors up to 100 11X11 pixel windows preselected on Earth to cover sources of interest. This limitation is due to lack of available bandwidth needed to transmit the full data. \(^{[10]}\)

11 Prepare the raw data; check it for anomalies due to malfunctions; calculate time-dependent detector variables; remove detector fingerprints; create shadowgrams from the corrected data; determine the sky background; deconvolve the shadowgrams; find bright X-ray sources; extract spectra, light curves and flux estimates; check the instruments performance; assemble a background images catalogue; monitor raw and corrected data in order to update detector fingerprint tables if needed. \(^{[18]}\)

12 The normalized difference of the exposure corrected counts in two energy bands. E.g. if we have a high energy band (H) and a low energy band (L) then one way to calculate the hardness ratio is: \(HR = \frac{H - L}{H + L}\).
Figure 3.1: The $q_{\text{scw}}$ pipeline. Boxes represents executables and areas surrounded by broken lines represents scripts. The name of the scripts are in boxes with a curled lower edge. Executables surrounded by a broken line are not described in this thesis. Those with whole lines, and bold typeface names, are. [19]
Figure 3.2: The \textit{q\_obs\_pipeline}. Boxes represents executables and areas surrounded by broken lines represents scripts. The name of the scripts are in boxes with a curled lower edge. Executables surrounded by a broken line are not described in this thesis. Those with whole lines, and bold typeface names, are. [19]
The `q_build_scw_list` executable runs three primary loops (figure 3.3 on page 24). The first loop compares the celestial coordinates of the center point of a reference science window \((\alpha_a, \delta_a)\)\(^{13}\) against the coordinates of the center points of all other science windows \((\alpha_{b|i}, \delta_{b|i})\)\(^{14}\). The reference science window and the list of other science windows are given in the parameter information of the executable. It uses the SLALIB\(^{15}\) function slaDsep (equations 3.1 and 3.2) to calculate the angular distance \((\Delta \sigma)\) between the two locations and flag all science windows where the distance is less than a chosen value.

The second loop calculates the difference in start time \((\Delta t_{\text{start}})\) of the reference \((t_{a,\text{start}})\) and input \((t_{b|i,\text{start}})\) exposures as well as the difference in end times \((\Delta t_{\text{end}})\) of the exposures (equation 3.3). The science window in question is unflagged if either \(\Delta t_{\text{start}}\) or \(\Delta t_{\text{end}}\) exceeds a chosen limit.

The third loop cycles through all flagged science windows and attaches them to the output observation group produced by the executable, assuming there are at least as many flagged science windows as the parameters require.

\[
\begin{align*}
\vec{a}_i &= \cos \alpha_a \cos \delta_a, & \vec{b}_i &= \cos \alpha_b \cos \delta_b \\
\vec{a}_j &= \sin \alpha_a \cos \delta_a, & \vec{b}_j &= \sin \alpha_b \cos \delta_b \\
\vec{a}_k &= \sin \delta_a, & \vec{b}_k &= \sin \delta_b
\end{align*}
\]

\[
\Delta \sigma = \arctan \left( \frac{\vec{a} \times \vec{b}}{\vec{a} \cdot \vec{b}} \right)
\]

\[
\begin{align*}
\Delta t_{\text{start}} &= \left| t_{a,\text{start}} - t_{b|i,\text{start}} \right| \\
\Delta t_{\text{end}} &= \left| t_{a,\text{end}} - t_{b|i,\text{end}} \right|
\end{align*}
\]

### 3.3 Q_flag.srcs

“**Function:** Flag those sources in the list of relevant sources, that also show up in a second catalogue.” [19]

This executable populates a number of masks, performs logical operations on them and finally combines the mask containing the final result with the source list it writes as its output.

The first phase of the execution (figure 3.4 on page 25) is to compare the names of found sources with the catalogue sources. Where the names match the corresponding entry in the “name mask”\(^{16}\) is set to true.

In the second phase the executable retrieves the celestial coordinates for each catalogue source in turn. A DAL3CAT function is used to locate all found sources that happen to be nearby\(^{17}\) those coordinates. These sources are set to true in a temporary “position mask” and the temporary

---

\(^{13}\) \(\alpha_a\) is the right ascension and \(\delta_a\) the declination of the reference science window.

\(^{14}\) \(\alpha_{b|i}\) is the right ascension and \(\delta_{b|i}\) the declination of the \(i\)th science window being compared against the reference science window.

\(^{15}\) An ANSI C library developed by Dr. Patrick Wallace for HEASARC that contains functions for positional-astronomy applications.

\(^{16}\) All masks in this executable operate on the output source list which contains found sources.

\(^{17}\) Within a certain radius that is specified by the parameters that corresponds with the source position error of the instrument in question.
mask is added\textsuperscript{18} to the final “position mask”. In this way a mask flagging all found sources that could possibly be a previously known catalogue source, based on position, is created step by step. The “position” and “name” masks are combined using a logic specified by the input parameters into the “result mask”. The final mask is then saved into the list of found sources.

3.4 \textbf{Q\_set\_src\_fluxes}

\textit{Function: Fill flux information vector columns in list of relevant sources for later comparison with analysis results.}\textsuperscript{19}

As the above quote from the QLA architectural design document states: this executable is used to record or create information about the expected photon fluxes or count rates in the detector energy channels of the INTEGRAL instruments for a list of sources. This can be accomplished in one of two ways; either the source has been observed by INTEGRAL previously and the historic data are used or one can create faux data based on the spectral model and detector responses. This data, regardless of how it was produced, is then stored in the catalogue output file to be compared against new observational data.

3.4.1 \textbf{Historical Data}

The easier case is when the catalogue source has been observed previously by INTEGRAL and historical data are available. The function can then extract the previously observed fluxes in each detector channel and write these values into the “RATE” or “FLUX” column of the catalogue file.

\textsuperscript{18} Logical OR.

\textsuperscript{19}
Figure 3.4: Q_Flag_Sources
it is working on. Some additional data are also stored; the upper and lower energy boundaries for each detector energy channel and the JEM-X renormalization factor\(^\text{19}\). Default data are used if energy boundary information is unavailable.

### 3.4.2 Simulated Fluxes

Expected detector energy channel fluxes are calculated based on the spectral model and detector responses, stored for each catalogue source, if historical data are unavailable. This model can consist of a number of elemental models\(^\text{20}\) and their related best fit model constants. The elemental models are some of those used by the XSPEC software\(^\text{21}\). The \texttt{q-set-src-fluxes} executable is able to recognize five different elemental models out of the more than 140 currently existing XSPEC models. These, together with their constants, are;

**Broken power law (bknpower)**

\[
flux(E) = \begin{cases} 
KE^{-\Gamma_1}, & E < E_{\text{break}} \\
KE_{\text{break}}^{\Gamma_2-\Gamma_1} \times (\frac{E}{1\text{keV}})^{-\Gamma_2}, & E \geq E_{\text{break}}
\end{cases}
\]  

\((3.4)\)

- \(\Gamma_1\): power law photon index for \(E < E_{\text{break}}\)
- \(\Gamma_2\): power law photon index for \(E \geq E_{\text{break}}\)
- \(E_{\text{break}}\): break point for the energy in keV
- \(K\): photons keV\(^{-1}\)cm\(^{-2}\)s\(^{-1}\) at 1 keV

**Power law, high energy exponential cutoff (cutoffpl)**

\[
flux(E) = KE^{-\alpha} e^{-\frac{E}{\beta}}
\]  

\((3.5)\)

- \(\alpha\): power law photon index
- \(\beta\): e-folding energy of exponential rolloff (in keV)
- \(K\): photons keV\(^{-1}\)cm\(^{-2}\)s\(^{-1}\) at 1 keV

**High-energy cutoff (highecut)**

\[
flux(E) = \begin{cases} 
\frac{E_c-E}{E_c}, & E > E_c \\
1.0, & E \leq E_c
\end{cases}
\]  

\((3.6)\)

- \(E_c\): cutoff energy in keV
- \(E_f\): e-folding energy in keV

\(^{19}\) To bring the JEM-X data to the same level as the other instruments.

\(^{20}\) Up to a theoretical maximum of 15.

\(^{21}\) Originally developed by Rick Shafer in 1983 and now maintained by K. A. Arnaud under the auspices of HEASARC. The XSPEC software package provides for an instrument independent x-ray spectral fitting software. The other most notable feature of XSPEC is the ability to add new spectral models as needed \([20]\).
Power law photon spectrum (powerlaw)

$$\text{flux}(E) = KE^{-\alpha}$$ (3.7)

$\alpha$  photon index of power law (dimensionless)
$K$  photons keV$^{-1}$ cm$^{-2}$ s$^{-1}$ at 1 keV

Photoelectric absorption, Wisconsin cross-sections (wabs)[21]

$$\text{flux}(E) = e^{-n_H \sigma(E)}, \sigma = \frac{c_0 + c_1 E + c_2 E^2}{E^3}$$ (3.8)

$n_H$  equivalent hydrogen column (in units of $10^{22}$ atoms cm$^{-2}$)

The elemental models and their related constants are read from the catalogue file and a photon spectrum is calculated by adding$^{22}$ or multiplying$^{23}$ together the fluxes of the elemental models in each energy bin. Because we are dealing with bins of varying width ($\Delta E$) the software actually calculates the flux at the low and high energy limit of each bin and averages these two values. The average flux is furthermore multiplied by the $\Delta E$ of the channel. The resulting photon spectrum mimics the photon energy distribution one might assume enters the detectors. In addition the detectors will have some certain quantum efficiency ($0 \leq QE \leq 1$) when converting photons into detections. The quantum efficiency of a detector is a function of photon energy. This relationship has been measured before the instrument is launched to be used in the data reduction$^{24}$. Each energy channel of the photon spectrum is multiplied by the average quantum efficiency of that channel to obtain the flux potentially recorded by the detector. Finally, there is a non-zero chance that a photon that belongs in one energy channel will end up in another one. This chance is measured before flight and is represented by a channel $\times$ channel redistribution matrix$^{25}$. For each detector energy channel we therefore have to sum the chance, over all photon spectrum channels, that a photon in that particular channel ends up in the detector channel of interest. The simulated detector channel fluxes are then stored in the catalogue file in the same manner as historical flux data.

3.5 Q_identify_srcs

"Function: Identify sources in the lists returned from ISSW, insofar as this has not already happened." [19]

This executable attempts to match found sources with catalogue sources, in the best possible manner, based on their celestial coordinates. The processing is divided into two functions. $Q_{\text{identify\_srcs\_match}}$ (“$Q_{\text{identify\_srcs\_match}}$” on page 28) does the actual matching. It produces a list that links the matched found sources and catalogue sources as well as the information of how many matches were made in total. $Q_{\text{identify\_srcs\_identify}}$ carries out a translation grid search (“Grid Search” on page 30). The grid search slightly adjusts the coordinates of the found $^{22}$ powerlaw, bknpower or cutoffpl.
$^{23}$ highcut or wabs.
$^{24}$ Stored in the ancillary response file (ARF) for XSPEC.
$^{25}$ The XSPEC RMF file.
sources for each iteration and then calls \texttt{q\_identify\_srcs\_match} to find out how many matches were made. The coordinate adjustment that gives the highest number of matches is then used to call \texttt{q\_identify\_srcs\_match} once more to create the final list linking the catalogue sources with the found sources. This information is stored with the found sources and any new or missing sources will result in alerts.

\subsection*{3.5.1 Q\_identify\_srcs\_match}

The work this function carries out can be divided into three steps; calculating relative distances between catalogue sources and found sources, ranking all found sources according to proximity to each catalogue source (and vice versa) and finally determine the quality of matches between catalogue sources and found sources.

**Relative distance** The function creates an $m \times n$ matrix containing the relative distance between each catalogue source ($s_c[i], [i \in 1, 2, \ldots, n]$) and each found source ($s_f[j], [j \in 1, 2, \ldots, m]$) (equation 3.9). This distance is derived from, but not equal to, the angular distance ($\Delta \sigma_{i,j}$, “Q\_build\_scw\_list” on page 20). The measurement used instead is given in units of the combined source position error$^{26}$ of the sources. Effects like rotation and skewing are not accounted for.

\begin{equation}
D_{m,n} = \begin{pmatrix}
\Delta \sigma_{rel}|1,1 & \Delta \sigma_{rel}|1,2 & \cdots & \Delta \sigma_{rel}|1,n \\
\Delta \sigma_{rel}|2,1 & \Delta \sigma_{rel}|2,2 & \cdots & \Delta \sigma_{rel}|2,n \\
\vdots & \vdots & \ddots & \vdots \\
\Delta \sigma_{rel}|m,1 & \Delta \sigma_{rel}|m,2 & \cdots & \Delta \sigma_{rel}|m,n
\end{pmatrix}
\end{equation}

Let us call the source position error for catalogue sources $\rho_c$ and for found sources $\rho_f$. It then follows that the combined source position error (equation 3.10) is $\rho_{tot}$. Furthermore we have two user defined parameters; $\rho_{def}$ and $\Delta \sigma_{max}$. $\rho_{def}$ is a default value for the combined source position error in case $\rho_{tot}$ cannot be calculated$^{27}$. $\Delta \sigma_{max}$ represents the largest distance between a catalogue source-found source pair that is still allowed to be called a match. With these we can calculate the relative distance ($\Delta \sigma_{rel}$) between two sources (equation 3.11).

\begin{equation}
\rho_{tot} = \sqrt{\rho_c^2 + \rho_f^2}
\end{equation}

\begin{equation}
\Delta \sigma_{rel} = \begin{cases}
\frac{\Delta \sigma_{i,j}}{\rho_{tot}}, & \Delta \sigma_{max} \geq 0 \land \rho_{tot} > 0 \\
\frac{\Delta \sigma_{i,j}}{\rho_{def}}, & \Delta \sigma_{max} \geq 0 \land \rho_{tot} = 0 \\
\Delta \sigma_{i,j}, & \Delta \sigma_{max} < 0
\end{cases}
\end{equation}

In this case $\Delta \sigma_{max}$ acts as a toggle. A negative value will make the function use $\Delta \sigma_{i,j}$ directly while a positive value will use $\Delta \sigma_{i,j}$ in units of the combined source position error as the value for $\Delta \sigma_{rel}$.

------

$^{26}$ The difference between the true position of the source and the position measured by the detector.

$^{27}$ If there are no values for $\rho_c$ and/or $\rho_f$
Ranking A user defined parameter is introduced to allow for some degree of uncertainty in the ranking: $\sigma_{fuzz}$. A test distance matrix is created using this value, $T_{i,j} = D_{m,n} \times (\sigma_{fuzz} + 1)$. The test distance matrix is analyzed in order to rank the catalogue sources on a scale of best to worst match\(^{28}\) for that particular found source. The results are stored in a new $m \times n$ matrix, catRank ($R_{m,n}[\text{cat}]$). Ranking is then also carried out for each column of the test distance matrix to rank how well each found source matches a particular catalogue source and the results are stored in another $m \times n$ matrix, foundRank ($R_{m,n}[\text{found}]$). These ranking matrices are initialized so that every entry is the same as the total number of found sources, i.e. the maximum possible rank. The ranking proceeds according to a certain set of rules (equation 3.12). These rules are used for calculating catRank. The index rules are swapped for foundRank ($n = j, m \neq i$).

$$R_{m,n} = \begin{cases} R_{m,n} - 1, & D_{m,n} < T_{i,j}[m = i, n \neq j], R_{m,n} > 1 \\ 0, & D_{m,n} > \Delta \sigma_{\text{max}} \end{cases}$$ (3.12)

The closest source will have a relative distance that is smaller than all the test distances and the rules indicate that its rank is reduced $(n - 1)$-times, where $n$ is the total number of found sources. The rank cannot be reduced to less than 1. Sources further away will get increasingly larger ranks (they are reduced fewer times) up to the cutoff distance $\Delta \sigma_{\text{max}}$. Any source further away than the cutoff distance will get no rank at all (0). The introduction of the $\sigma_{fuzz}$ parameter means that sources that are close to each other get the same rank. You can have shared first, second, etc rankings. The rankings are read along the rows for catRank and along the columns for foundRank (e.g. figures 3.5 and 3.6).

Matching The two ranking matrices need to be unified in order to have a single measurement of matching qualities. The function analyzes the catRank and foundRank matrices comparing the $R_{m,n}[\text{cat}]$ and $R_{m,n}[\text{found}]$ values against each other. In the easiest case, a “perfect match”, $R_{m,n}[\text{cat}] = R_{m,n}[\text{found}] = 1^{29}$. These matches are stored and the corresponding row/column

\(^{28}\) The best match indicated by a rank of 1, the second best by a rank of 2, etc.

\(^{29}\) $(m, n) \in \{(1, 3), (2, 1), (4, 4)\}$ in the example matrices.
is eliminated from further inspection. In the next phase $R_{m,n}\{\text{cat}\} = 1, R_{m,n}\{\text{found}\} = 2$ results ("confused match")$^{30}$ and $R_{m,n}\{\text{cat}\} = 2, R_{m,n}\{\text{found}\} = 1$ ("multi match")$^{31}$ are stored. The results from the previous step are then checked through. A unified list of matchings, including the quality of the match, is built. There are three quality identifiers; "good" indicating there is a 1-to-1 fit, "confused source" where more than one found source matches the same catalogue source equally well or "multiple sources" if more than one catalogue source fits the same found source equally well. This determination is done by proceeding along the row/column from the match made in the previous step and checking for further matches$^{32}$. The unified match list and the information about the total number of matches made are then returned to $q\_identify\_srs\_identify$.

### 3.5.2 $q\_identify\_srs\_identify$

The $q\_identify\_srs\_identify$ function utilizes $q\_identify\_srs\_match$ to find the best possible solution to a potential spacecraft pointing error$^{33}$ by conducting a grid search. It adjusts for the pointing error by testing different pointings to find the best correction. It then recalculates the coordinates of the found sources. Alerts are created for any significant differences between the catalogue and the found sources.

**Grid Search** The pointing error of the spacecraft can be thought of as an error radius, painting a circle on the sky, within which the true pointing lies. It may become impossible to match found sources with catalogue sources if the true pointing is sufficiently offset from the center of this circle. The grid search aims to avoid this problem by testing a number of pointings within, or nearby, the circle to see which test pointing gives the largest number of matches. The best test pointing is considered to be the true pointing of the spacecraft. The method of selecting these test pointings$^{34}$ is to impose a square grid of test pointings on the circle. In my example (figure 3.7 on page 31) the search would proceed by taking up to two steps left/right and up/down from the default assumption, for a total of 25 test pointings. Each such step would have the length $l = \frac{1}{\sqrt{n}} \cdot r_{\text{error}}$ where the maximum number of steps $n = 2$ in this case. The adjustments made to the coordinates of the found sources before attempting to match them against the catalogue would then be $(\Delta \alpha, \Delta \delta) = (a \cdot l, b \cdot l)$ where $a, b \in \{2, 1, 0, -1, -2\}$. Typically $n \approx 10$ during normal analysis. The values of $n$ and $r_{\text{error}}$ are given to the executable as parameters.

**Creating the final output** An alert is created if another pointing than the default one matches the data best. The function also recalculates the celestial coordinates of the found sources if it needs to adjust for such an offset. The function calls on $q\_identify\_srs\_match$ one final time to obtain a list of the best possible source matchings. The list is looped over to check the match quality of each found source. Alerts are created if a source is new or part of a confused/multiple source constellation. The final results are then written to the output file.

---

$^{30}$ $(m,n) = (5,2)$.

$^{31}$ $(m,n) = (3,5)$.

$^{32}$ $(m,n) \in \{(4,4), (4,5)\}$ in the example matrices constitutes a multiple source case.

$^{33}$ The difference between the true pointing of the spacecraft and our measurement of that pointing.

$^{34}$ And the reason it is called a “grid search”. 

---

30
3.6 Q_match_src_fluxes

"Function: Compare the source fluxes and hardness ratios derived during the QLA with the values expected from catalogue data. Raise alerts in case of deviations larger than a parametrized value. This tool has also simplified functionalities of q_set_fluxes and q_identify_srcs (calculate expected source fluxes and compare catalog and detected source positions), so that calling only this tool may suffice the minimum requirements of QLA (an option for the initial test phase of the mission).

As of August 2002, the AO1 TOO target names and positions are implemented (hard coded in the header file), so that alerts are issued if TOO targets are detected. The current method of implementation may be later replaced by a more generic method (e.g., put TOO sources in a FITS catalog format)." [19]

3.6.1 The Overall Structure

The core functionality of q_match_src_fluxes consists of three nested loops. The outermost loop browses a source list containing found sources. The second loop browses an extract of the catalogue and the innermost loop goes through the flux bins of the catalogue source being analyzed.

3.6.2 Criteria for Further Analysis

A set of checks is carried out, before the two inner loops are activated, to determine if the found source should be further analyzed or not. The significance of the detection, as determined by the ISSW, is compared against a parametrized limit. The found source is rejected, and the loop proceeds, if the significance is too low and the significance criteria is active. The angular distance ("Q_build_scw_list" on page 20) between the found source and the center of the image is calculated. The source is analyzed only if it is within a certain distance\textsuperscript{35} of the center. The number of observation groups or science windows also need to be above zero to proceed.

\textsuperscript{35} This value is set by the input parameters.
3.6.3 Matching Sources

A catalogue source matching the found source will be looked for if the previous checks are passed. It was decided to include a very rudimentary version of *q_identify_sources* as one matching option ("Q_identify_srcs" on page 27)\(^{36}\). This functionality checks if the catalogue source is within a certain minimum angular distance of the found source. This is now redundant and the component just compares the source ID’s of the found sources and the catalogue sources to find a match.

3.6.4 Comparing Fluxes

The component will compare the catalogue source with the found source, flux bin by flux bin, in the innermost loop. Catalogue flux data can either be read directly from the catalogue or it can be estimated based on the X-ray spectral model for the source. The latter works out the same way as it does in *q_set_src_fluxes* and the *q_match_src_fluxes_model* function is in fact identical to *q_set_src_fluxes_model* ("Simulated Fluxes" on page 26). The main difference is that the choice between using history data or simulated data is based on an input parameter instead of checking for data availability. If we name the flux in the i'th flux bin of the catalogue source \(f_{\text{cat},i}\), it’s measuring error \(\delta f_{\text{cat},i}\) and similarly for the found source \(f_{\text{res},i}\) and \(\delta f_{\text{res},i}\) then;

\[
\xi = \frac{f_{\text{res},i} - \delta f_{\text{res},i}}{f_{\text{cat},i} + \delta f_{\text{cat},i}}
\]

(3.13)

Alerts of increasing severity are issued if \(\xi\) exceeds certain thresholds\(^ {37}\).

3.6.5 Hardness Ratio

Changes in the shape of the spectrum of a source are also of interest. The component calculates the difference between the hardness ratios of the found source and the catalogue source\(^ {38}\);

\[
\zeta = \frac{|f_{\text{cat},h} - f_{\text{cat},s}|}{f_{\text{cat},h} + f_{\text{cat},s}} - \frac{|f_{\text{res},H} - f_{\text{res},S}|}{f_{\text{res},H} + f_{\text{res},S}}
\]

(3.14)

\(\zeta\) is then compared to four threshold values and an alert of corresponding severity level is issued if a threshold is exceeded.

3.6.6 New and Target of Opportunity Sources

The component carries out two final checks once all catalogue sources have been attempted as a match to a found source. An alert is issued if a found source could not be matched with a catalogue source. The severity of the alert depends on the detection significance for that source. The second check is to find out if the found source is also a target of opportunity (TOO) source. TOO sources are sources of exceptional interest that necessitate changing the prescheduled program of INTEGRAL \(^ {22}\). Typical examples are “new” transient sources like X-ray novae or supernovae. Alerts for the detection of a TOO source are issued if the angular distance between a found source and a source on the TOO source list is less than a parameter defined value.

---

\(^{36}\) This decision was made because it was uncertain if a fully functional version of *q_identify_sources* would be available during early system testing.

\(^{37}\) Specified in the input parameters.

\(^{38}\) The column indexes for the hard and soft columns; h, s, H and S are read from parameters.
4

Testing QLA Source Identification

“There are no safe paths in this part of the world. Remember you are over the Edge of the Wild now, and in for all sorts of fun wherever you go.” — J.R.R. Tolkien, The Hobbit

In this Chapter I will test how well \texttt{q\_identify\_sres} is able to carry out its tasks. The analysis carried out by the component is a two-stepped hierarchy; At the top we have the grid search functionality (“Grid Search Testing” on page 53) that will repeatedly call on the lower step, the source identification. The first three sections of this Chapter deal with source identification functionality. The first section (“Test Setup and the Definition of Success” on page 33) explores how test success is defined and calculated. Additionally some notation will be established and the basic test scheme is described. The second section (“Input Parameters and Source Data” on page 37) describes which inputs the component requires and how these are supplied. Special attention is given to the generation of source data. The source identification testing is carried out in the third section (“Source Identification Testing” on page 43). Results are shown and two problems with the algorithm are discovered. Solutions to these problems are also described. The fourth, and final, section (“Grid Search Testing” on page 53) tests the grid search functionality.

4.1 Test Setup and the Definition of Success

Here I will define what constitutes a success (or failure) when \texttt{q\_identify\_sres} is used to analyze observational data. That definition is used to derive how the success rate is calculated when a large number of observations are analyzed. The first part of the section looks at what happens when the component is doing its work and what types of outcomes are possible (“Source Matching” on page 33). Some useful notation is also established. The second part contains a discussion about what is meant by “success” and the choices one may make to define success in source matching (“Source Matching Success” on page 34). This part ends with a formal definition of the success criteria used during testing in this Chapter. The third part of this section establishes the general manner in which tests are conducted and the way test success ratios are calculated (“Test Setup and Evaluation” on page 36).

4.1.1 Source Matching

The core functionality of \texttt{q\_identify\_sres} is to compare two source lists; one containing previously known sources (the catalogue) and one containing sources that have just been observed by INTEGRAL’s instruments. The shorthand notation used for the former is catalogue sources
and the set of catalogue sources being analyzed is denoted $C$. The second set will be called the found sources and denoted $F$. The set of catalogue sources are an extract of the full INTEGRAL source catalogue (i.e. $C \subseteq \text{CATALOGUE}$) containing all known sources within the field of view of the observing instrument. I will use the notation $s_{c|i}$ to denote the $i$:th catalogue source within $C$. The found sources are a set of sources that have recently been observed by an instrument onboard INTEGRAL. These sources are constrained to reside within some (known) region of space which is the field of view of the instrument in question. I will use the notation $s_{f|j}$ to denote the $j$:th found source within $F$.

The component will try to find pairs of matching catalogue sources-found sources. No (simple) mathematical representation for a match between sources exists as matches depend on complicated processing within the $q_{\text{identify\_srcs}}$ component (“$Q_{\text{identify\_srcs\_match}}$ on page 28). I will use the notation $s_{c|i} \leftrightarrow s_{f|j}$ to indicate a match\(^1\). In a similar fashion I will use the notation $C \leftrightarrow F$ to indicate the process of matching two source sets against each other\(^2\). If $q_{\text{identify\_srcs}}$ finds that $s_{c|i} \leftrightarrow s_{f|j}$ then the $j$:th found source is identified as being identical to the $i$:th catalogue source.

There are five types of outcomes when individual sources are matched. 1) The $i$:th catalogue source is matched to one, and only one, found source (equation 4.1a). This is the normal outcome when the found source is identified as being an already known source. I will call this a “unique match”. 2) No catalogue source matches the $i$:th found source (equation 4.1b). This would typically result in a new source alert. I will call this a “new source”. 3) No found source matches the $i$:th catalogue source (equation 4.1c). This should generate an alert signifying that a catalogue source was lost. This functionality is no longer supported in the code so this result will be disregarded. 4) Several catalogue sources are equally good matches for a found source (equation 4.1d). This generates a multiple source alert. I will call this a “multiple source identification”. 5) Multiple found sources are equally good matches for a catalogue source (equation 4.1e). This generates a source confusion alert. I will call this a “confused source identification”. A single iteration of $C \leftrightarrow F$ can generate several of these match types in any combinations imaginable.

\[
\begin{align*}
s_{c|i} & \leftrightarrow s_{f|j} \\
   & \leftarrow \emptyset & (4.1a) \\
   & \leftarrow \emptyset & (4.1b) \\
   s_{c|i} & \leftarrow \emptyset & (4.1c) \\
   s_{f|j} & \leftarrow s_{c|x} \in \{s_{c|\alpha}, s_{c|\beta}, \ldots\} & (4.1d) \\
   s_{c|i} & \leftarrow s_{f|x} \in \{s_{f|\alpha}, s_{f|\beta}, \ldots\} & (4.1e)
\end{align*}
\]

### 4.1.2 Source Matching Success

One can expect any number of each of the match categories each time a source matching ($C \leftrightarrow F$) is carried out. Some found sources would find unique matches among the the catalogue sources but especially regions of the sky where sources are densely distributed tends to produce many multiple and confused source identifications. Matchings can also be asymmetric; catalogue source

---

\(^1\) Read as: “The $i$:th catalogue source was matched to the $j$:th found source”.

\(^2\) Read as: “The set of catalogue sources are matched with the set of found sources”.

34
s_{c|i} can be multiply matched with found sources s_{f|a} and s_{f|b} while s_{f|b} is uniquely matched with s_{c|i}, or even some entirely different catalogue source. This poses a problem for interpreting the results. We could get dozens, even hundreds, of unique combinations of outcomes\(^3\) when conducting a large number of matchings. Thus I will need to define more easily measurable success criteria. This requires a choice to be made when analyzing the results:  
1) Every single found source is correctly matched to a catalogue source and all alerts issued are correct. 
2) All issued new source alerts are correct but sources don’t always have to be correctly matched. Let us imagine two simple sets of catalogue sources and found sources (equation 4.2).

\[
\begin{cases}
C = \{s_{c|\alpha}, s_{c|\beta}\} \\
F = \{s_{f|\alpha}, s_{f|\beta}, s_{f|\gamma}\}
\end{cases}
\]  
(4.2)

Found sources and catalogue sources sharing the same index are meant to be matched and the extra source in \(F\), \(s_{f|\gamma}\), is meant to produce a new source alert. The difference in approach can then be illustrated by two sample source matching outcomes (equation 4.3 and equation 4.4).

\[
\begin{cases}
s_{c|\alpha} \Rightarrow s_{f|\alpha} \\
s_{c|\beta} \Rightarrow s_{f|\beta} \\
s_{f|\gamma} = \emptyset
\end{cases}
\]  
(4.3)

\[
\begin{cases}
s_{c|\alpha} \Rightarrow s_{f|\beta} \\
s_{c|\beta} \Rightarrow s_{f|\alpha} \\
s_{f|\gamma} = \emptyset
\end{cases}
\]  
(4.4)

In the first case (equation 4.3) both catalogue sources are correctly matched with the found sources, while the new source is not matched and thus an alert is generated. This can be considered as a success if we treat \(q\_\text{identify\_srcs}\) as a source identification engine (the first choice). In the second case (equation 4.4) both found sources are incorrectly identified but fortunately they are matched with the catalogue sources and thus no false alerts are created. Both outcomes are acceptable if we treat \(q\_\text{identify\_srcs}\) like an alert engine (the second choice).

I will select to treat the component as an alert engine during testing because this matches the real use case for the QLA more closely. It is not expected to provide automated source identification but rather bring human attention to interesting data.

The broad strokes of the source matching success criteria could, based on the previous, be stated as follows;  
\(\diamond\) The component should not generate any alerts when the found sources have been observed previously and therefore exists among the catalogue sources and  
\(\diamond\) The component must generate a new source alert when a bona fide new source exists among the found sources.

The third possible criteria, \(\diamond\) The component must generate a missing source alert if a catalogue source is failed to be detected, is discarded because the current version of the component does not support these detections. I can then formally state the first (equation 4.5) and second (equation 4.6) success criterion.

\[
\forall s_{f|i} \exists! s_{c|i} (s_{c|i} \Rightarrow s_{f|i})
\]  
(4.5)

\(^3\) E.g. \([C \equiv F]_1\) might result in 4 matches and 1 new source detection while \([C \equiv F]_2\) might result in 2 matches and 1 source confusion. How do we compare these against each other?
\[ \exists s_f(s_f \not\in \emptyset) \quad (4.6) \]

It is important to note that both criteria cannot be true at once. The first criterion applies only to found source sets where every source has a match among \( C \) and the second criterion applies only when a new source is present in \( F \).

### 4.1.3 Test Setup and Evaluation

I need to run two different matching tests side by side, \( C \equiv F \) and \( C' \equiv F' \), with two success criteria that are simultaneously incompatible (equations 4.5 and 4.6 respectively).

#### Source List Generation

For \( C \equiv F \) I will need a set of catalogue sources (equation 4.7a) and a set of found sources (equation 4.7b) that have an equal amount of members where the members are picked in such a manner that the expected (but not guaranteed) outcome is according to the first success criterion (for more details, please refer to “Source Data” on page 39).

\[
C = \{s_{c|1}, s_{c|2}, \ldots, s_{c|n}\} \quad (4.7a)
\]
\[
F = \{s_{f|1}, s_{f|2}, \ldots, s_{f|n}\} \quad (4.7b)
\]
\[
C \setminus C' = \{s_{c|i}\} \quad (4.7c)
\]
\[
F' = F \quad (4.7d)
\]

The second pair of source sets (\( C' \) and \( F' \)) are constructed by randomly eliminating one catalogue source from \( C \) (equation 4.7c) while keeping the same found sources (equation 4.7d). This results in a “new” source being present in \( F' \).

#### Runs and Tests

A pair of \( C \equiv F \) and \( C' \equiv F' \) will be called a run. In order to gain reliable success statistics I will be conducting a large number of runs with all parameters fixed. I will call such a set of runs for a test. A test might typically consist of \( 10^7 \) runs. New source sets \((C_i, F_i, C'_i, F'_i)\) are generated for every run within a test.

#### Success Rates

Based on the simplification above ("Source Matching Success" on page 34 and equations 4.5 and 4.6 in particular) a source matching \((C \equiv F)\) can have three different outcomes; no alert is issued\(^4\), a new source is detected\(^5\) or some other alert is issued. A run can then have six possible outcomes, three each for \( C \equiv F \) and \( C' \equiv F' \). This allows me to define six ratios quantifying the outcomes of a test; no alert of any type is issued (\( \kappa_{\text{no}} \), equation 4.8), a new source alert is issued (\( \kappa_{\text{new}} \), equation 4.9) and some other alert is issued (\( \kappa_{\text{other}} \), equation 4.10).

\(^4\) A one to one match between all sources in C and F was found.

\(^5\) Or multiple new sources.
\[ \kappa_{\text{no}} = \frac{\sum_{i=1}^{n} \forall s_f \exists! s_c(s_c \Rightarrow s_f), (C_i \rightleftharpoons F_i)}{n} \] (4.8)

\[ \kappa_{\text{new}} = \frac{\sum_{i=1}^{n} \exists s_f (s_f \Rightarrow \emptyset), (C_i \rightleftharpoons F_i)}{n} \] (4.9)

\[ \kappa_{\text{other}} = 1 - (\kappa_{\text{no}} + \kappa_{\text{new}}) \] (4.10)

The equivalent are \( \kappa'_{\text{no}}, \kappa'_{\text{new}} \) and \( \kappa'_{\text{other}} \) for \( C' \rightleftharpoons F' \). As there is no practical difference between the \( \kappa \)'s and \( \kappa' \)'s I will forgo the distinction henceforth and only use the symbols without prims. On occasion I will use these ratios in various plots to illustrate source matching efficiency. With \( C \rightleftharpoons F \) set up to ideally give no alerts and \( C' \rightleftharpoons F' \) only new source alerts the two headline success rates for a test are as follows: (equation 4.11).

\[
\begin{cases}
\Lambda = \kappa_{\text{no}} \\
\Lambda' = \kappa_{\text{new}}
\end{cases}
\] (4.11)

A flawless test would give \( \Lambda = 1.0 \) and \( \Lambda' = 1.0 \). Source confusions, multiple source identifications and false new source alerts tend to drive the value towards zero.

### 4.2 Input Parameters and Source Data

In this section I will examine the numerical values and data inputs required in order for \texttt{q\_identify\_srcs} to perform its tasks. The component has an input parameter file attached that specifies a number of values that have to be supplied. These values are called input parameters. Some of them affect the analysis in various ways that cannot be determined trivially. The testing of \texttt{q\_identify\_srcs} largely involves exploring how the outcome of the analysis changes when these parameters change. The input parameters are presented in the first part of this section ("Input Parameters" on page 37). Two of the input parameters are DOL’s that should point to the location in the ISDC system where the catalogue source set \( (C) \) and found source set \( (F) \) are located. Ideally I would like to use actual data from the live system at ISDC to test the component. Unfortunately such data is not available. The partially analyzed data products flowing through the pipeline into this component would require a significant effort by staff at ISDC to extract from the live process which makes them unfeasible to obtain\(^6\). I will therefore have to create suitable faux data instead. How this data is created is the topic of the second part of this section ("Source Data" on page 39).

#### 4.2.1 Input Parameters

The parameter file associated with \texttt{q\_identify\_srcs} contains a total of 15 entries (table 4.1 on page 38). Most of these parameters define interactions between the ISDC system and the component and are thus not needed for the simulations I intend to run. The remaining four parameters are:

---

\(^6\) Based on private correspondence with Mr Carlo Ferrigno, Coordinator of INTEGRAL operations.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1_cat_dol</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>src1_res_dol</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instrument</td>
<td>Integer</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>unit</td>
<td>Integer</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>alert_new</td>
<td>Integer</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>alert_lost</td>
<td>Integer</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>alert_conf</td>
<td>Integer</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>alert_mult</td>
<td>Integer</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>alert_shift</td>
<td>Integer</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>chat</td>
<td>Integer</td>
<td>0</td>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>relDist</td>
<td>Real</td>
<td>1.5</td>
<td>-100</td>
<td>100</td>
</tr>
<tr>
<td>fluxLimit</td>
<td>Real</td>
<td>0.0</td>
<td>0.0</td>
<td>5000</td>
</tr>
<tr>
<td>searchRad</td>
<td>Real</td>
<td>5.0</td>
<td>0.0</td>
<td>90.0</td>
</tr>
<tr>
<td>gridNum</td>
<td>Integer</td>
<td>10</td>
<td>0.0</td>
<td>1000.0</td>
</tr>
<tr>
<td>distFuzz</td>
<td>Real</td>
<td>0.15</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 4.1: Input parameters for the \textit{q\_identify\_srcs} component. Footnoted (*, ⋆, †, ‡) parameters are not used in the simulations. The table lists the type of the parameter (string, integer, etc), the default value, the smallest and the largest accepted value (for numerical parameters).

\(\Delta \sigma_{\text{max}}\)

This parameter, called “relDist”\(^7\) in the parameter file, sets the upper limit for how large \(\Delta \sigma_{\text{rel}}\) is allowed for a found source to be identified as matching a certain catalogue source (“Ranking” on page 29). Its value is allowed to vary within the range \([-100, 100]\). A key goal of these tests is to see what impact \(\Delta \sigma_{\text{max}}\) has on the results of the source identification. I will use the default value \(\Delta \sigma_{\text{max}} = 1.5\) as a starting point for the tests. Later on I will allow \(\Delta \sigma_{\text{max}}\) to vary within the allowed interval to further explore the behavior of the parameter.

\(^{r}\text{search}\)

This parameter (searchRad in the parameter file) corresponds to the pointing error of the instrument. When the grid search functionality determines the magnitude of the pointing error (“Q\_identify\_srcs\_identify” on page 30) of each observation it will search for the best possible fit...  

\(^7\) A certain degree of confusion in terminology has crept into the parameter file over time. The name of this parameter refers to the relative distance, \(\Delta \sigma_{\text{rel}}\), between sources (“Relative distance” on page 28). But in reality the parameter sets an upper limit for \(\Delta \sigma_{\text{rel}}\) and I will stick to the notation used in the code proper, rather than what the parameter is called in the parameter file.

\(^\ast\) Contains the DOL of the Source Catalogue or the source list containing the found sources being identified. DOL’s are only relevant within the ISDC system context. The necessary source lists are instead produced within the simulation code.

\(^\dagger\) Used only for reading or storing data. The instrument number has no effect on execution and within this simulation I can choose which angular unit (rad or deg) to use without using the parameter.

\(^\ddagger\) These parameters allow the user to adjust the severity of the alerts the component issues type by type. Alert severity is used for data logging purposes at ISDC and has no relevance for the tests I run.

\(^\natural\) Used to set a lower flux limit, for missing catalogue sources, under which no alert is given. The simulation does not suppress any alerts and can thus be seen as always using the default value (0.0) even though this functionality is not included in the simulation code.
within an area specified by \( r_{\text{search}} \). Thus \( r_{\text{search}} \) need to be set slightly larger than the maximum value of the pointing error, \( \rho_{\text{pointing}} \). From literature we find that \( \rho_{\text{pointing}} \leq 5' \) and I will therefore use a fixed value, \( r_{\text{search}} = 6' \), for these simulations [23].

\( n_{\text{grid}} \)

The circular area described by \( r_{\text{search}} \) is placed within a square search area with sides that have a length of \( 2 \times r_{\text{search}} \). This square is then divided into \((n_{\text{grid}} \times 2)^2\) grid steps (“Grid Search” on page 30). My aim with the grid search simulations is to find out what impact this parameter has on the results of the grid search. The parameter is called gridNum in the parameter file.

\( \sigma_{\text{fuzz}} \)

The fractional difference in the relative distances of two found sources that allows them to be considered equally good fits for a catalogue source. The impact of this parameter on the outcome of the source matching is another key goal of the source identification simulations\(^8\). I will use the default value \( \sigma_{\text{fuzz}} = 0.15 \) as a starting point for the tests. Later on I will allow \( \sigma_{\text{fuzz}} \) to vary to further explore the behavior of the parameter. The parameter is called distFuzz in the parameter file.

4.2.2 Source Data

As explained previously (“Test Setup and Evaluation” on page 36) I need source data sets \( C \) (catalogue) and \( F \) (found) as well as the derived \( C' \) and \( F' \) for a run. The found sources can be generated from the catalogue sources by adding the source position error\(^9\). To generate the required data I thus need two things; a set of catalogue sources and an understanding of the source position error due to the detectors onboard INTEGRAL.

Catalogue Sources (\( C \) and \( C' \))

A natural choice for catalogue data is the current INTEGRAL Source Catalogue [24]. It contains a total of 974 sources that have been detected by INTEGRAL so far. An overwhelming majority of which have been seen by the IBIS instrument. Most sources are located along the Galactic plane and particularly towards the center of the Galaxy (figure 4.1 on page 40). The method I will use for generating \( C \) is to choose, at random\(^10\), a \( 9^\circ \times 9^\circ \) field of view\(^11\) and extract the corresponding sources from the INTEGRAL Source Catalogue. The main advantage of using this method is that the catalogue data will match the actual data in the ISDC system fairly closely and will thus be realistic.

---

\(^8\) Exploring the behavior of \( \Delta \sigma_{\text{max}} \) and \( \sigma_{\text{fuzz}} \) is important as their impact on the results cannot be trivially determined.

\(^9\) The error introduced by the uncertainty in determining the source position by the instruments detector.

\(^10\) A large fraction of all sources lie along the Galactic plane, especially towards the center of the Galaxy. The Galactic plane is therefore of particular interest for INTEGRAL observations and the instrument performs regular scans along it. Thus real observation patterns are different than the test approach. This does not, by itself, affect the testing or its results. It is worth bearing in mind, however, that QLA performance in high density regions has an accentuated level of importance for staff monitoring the analysis who have to deal with the alerts the system generates.

\(^11\) Corresponding to the fully coded field of view of the IBIS instrument onboard INTEGRAL.
Figure 4.1: The sky distribution of all sources detected by INTEGRAL.

Figure 4.2: The fraction of $9^\circ \times 9^\circ$ fields in the sky containing the indicated number of sources.
The INTEGRAL Source Catalogue provides the coordinates \((\alpha, \delta)\) and the source position error \((\rho)\) for the sources. \(\alpha\) and \(\delta\) will be used to calculate the angular distance \((\Delta\sigma)\) between sources ("Q_build_scw_list" on page 20) and this together with \(\rho\) is then used to derive the relative distance\(^{12}\) \((\Delta\sigma_{rel})\) equation 3.11 on page 28) between the same.

\(C'\) is then generated by selecting one catalogue source at random and eliminating it from the final source list. Thus one found source will lack a matching catalogue source and be treated as a new source detection. This also means that any initially selected field of view containing less than two sources has to be discarded and a new field is selected instead.

The source density in the generated fields of view will be very unevenly spread. If we choose the number of sources inside a randomly generated \(9^\circ \times 9^\circ\) field of view as the free variable \((x)\) then the number of such fields generated \((y)\) is \(y \propto (\frac{3}{4})^x\) (figure 4.2 on page 40). This introduces a sampling bias that gets more severe towards the highest density end of the tests\(^{13}\). Based on experimentation this sampling bias does not unduly affect the outcomes of the tests and can thus be ignored.

**Found sources \((F'\text{ and } F')\)**

The set of found sources is generated based on the same data that was used to generate \(C\). A found source is created by adding a vector to a catalogue source \(\vec{s}_{f|i} = \vec{s}_{c|i} + \vec{\varrho}_i\), where \(\vec{\varrho}\) is the source position error and \(\vec{s}_{c|i} = [\alpha_{c|i}, \delta_{c|i}]\). The vector \(\vec{\varrho}\) needs to be generated randomly for each \(\vec{s}_{f|i}\) in a manner that reproduces the distribution of the actual source position error faithfully.

To simplify the generation of \(\vec{\varrho}\) I choose to express it in polar coordinates; the radius being \(r\) and azimuth \(\Phi\), \(\vec{\varrho} = [r, \Phi]\). When looking at the individual components of \(\vec{\varrho}\) it is clear that \(\Phi\) has no particular bias and it will be randomized using a continuous uniform distribution. \(r\) has a more complicated behavior. The point source location accuracy (PSLA), which sets an upper boundary for \(r\), depends strongly on two factors: the detection significance and the angle between the axis of the telescope and the source. I will restrict this analysis to sources near the telescopes axis\(^{14}\) in order to reduce the off-axis effect. One can find an expression for the PSLA in literature. S. Scaringi et al. finds that the relationship between the detection significance \((\nu)\) and the offset\(^{15}\) in arcminutes \((f(\nu))\) has the general form expressed below (equation 4.12) [25]. The authors also tabulate the fitting parameters; \(a, b\) and \(c\) (table 4.2 on page 42). I will use the 90% PSL confidence level.

\[
f(\nu) = a\nu^c + b \tag{4.12}
\]

Knowing the range within which (a certain fraction of) \(r\) resides is not enough. I also need to know the shape of the point spread function (PSF) to be able to recreate the distribution. Gros et al. suggests that a Gaussian distribution can be taken as an adequate approximation of the PSF [26]. I turn to the INTEGRAL Source Catalogue in order to obtain the fitting parameters

\(\text{12 This distance is derived from, but not equal to, the angular distance. The measurement used instead is given in units of the combined source position error. The source position error is the difference between the true position of the source and the position measured by the detector.}\)

\(\text{13 We are restricted to essentially just one field of view at the highest density level. The one directly towards the center of the Galaxy.}\)

\(\text{14 Hence the choice of using the } 9^\circ \times 9^\circ \text{ fully coded field of view.}\)

\(\text{15 The highest possible value of } r.\)
Table 4.2: Estimated PSLA fitting parameters for the IBIS fully coded field of view.

<table>
<thead>
<tr>
<th>Confidence level</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>65%</td>
<td>17.65</td>
<td>0.14</td>
<td>-1.19</td>
</tr>
<tr>
<td>90%</td>
<td>31.10</td>
<td>0.23</td>
<td>-1.25</td>
</tr>
<tr>
<td>95%</td>
<td>36.70</td>
<td>0.25</td>
<td>-1.24</td>
</tr>
<tr>
<td>99%</td>
<td>36.40</td>
<td>0.14</td>
<td>-1.04</td>
</tr>
</tbody>
</table>

Table 4.3: Approximate error ranges where each of the four component PSF’s dominate.

<table>
<thead>
<tr>
<th>Error Range</th>
<th>Confidence level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0^\circ \leq r &lt; 0.007^\circ$</td>
<td>65%</td>
</tr>
<tr>
<td>$0.007^\circ \leq r &lt; 0.02^\circ$</td>
<td>90%</td>
</tr>
<tr>
<td>$0.02^\circ \leq r &lt; 0.04^\circ$</td>
<td>95%</td>
</tr>
<tr>
<td>$r \geq 0.04^\circ$</td>
<td>99%</td>
</tr>
</tbody>
</table>

Table 4.4: Fitting parameters for the simulated $r$-distribution.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\eta_i$</th>
<th>$\sigma_i$</th>
<th>$\mu_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.77328</td>
<td>0.00114</td>
<td>0.00000</td>
</tr>
<tr>
<td>2</td>
<td>0.03040</td>
<td>0.00333</td>
<td>0.01324</td>
</tr>
<tr>
<td>3</td>
<td>0.03610</td>
<td>0.00368</td>
<td>0.03254</td>
</tr>
<tr>
<td>4</td>
<td>0.16023</td>
<td>0.01037</td>
<td>0.07492</td>
</tr>
</tbody>
</table>

There is an almost 1:1 correspondence between these sources and those detected only by IBIS.

Furthermore, the distribution of sources detected only by IBIS has a more or less identical distribution compared to the distribution of all sources detected by INTEGRAL.
This equation has a total of twelve free parameters but luckily it is possible to make quite accurate initial estimates for both the means ($\mu_i$) and the standard deviations ($\sigma_i$) by calculating $\mu$ and $\sigma$ within the four ranges mentioned previously. The $\eta_i$ parameters are scaling factors and an initial estimate for them can be gained by calculating the fraction of sources that falls within each range. The final fitting parameters can be seen in table 4.4 on page 42. An overlay of the INTEGRAL Source Catalogue error distribution with the fitted $r$-distribution can be seen in the image mentioned previously (figure 4.3 on page 44).

With the distribution functions of $r$ and $\Phi$ known it becomes possible to generate $\vec{\rho}$ and thus obtain $\vec{s}_{F|i} = [\alpha'_i, \delta'_i]$ for $F$. A value for the source position error ($\rho_f$) is still required. By generating my own sources, instead of observing real ones, I have the advantage of knowing both the true position ($[\alpha_i, \delta_i]$) and the observed position ($[\alpha'_i, \delta'_i]$) of the source. I.e. $\rho_f = |\vec{\rho}|$. In the spirit of keeping these simulations close to reality I will choose a value for $\rho_f$ that is a general estimate of the source position error instead of the accurate source-by-source values. Therefore I will use the boundary within which 90% of all source position errors will fall; $\rho_f \approx 0.081^\circ$. With this information I can generate replacements for the source data and proceed with the tests.

### 4.3 Source Identification Testing

This section recounts my testing of the source identification functionality of `identify_srcs`. In the first part I establish a baseline for the source identification success rates (“Baseline Test” on page 43). In the second part I explore if the success rates can be improved by allowing the input parameter values to change from their defaults (“Parameter Optimization” on page 45). A problem with how one of the parameters affects the source identification is discovered during the parameter optimization. This problem is analyzed in-depth in the third part (“Exploring the Behavior of $\sigma_{fuzz}$” on page 48). The discussion is then expanded beyond testing the current algorithm towards a potential solution to the problem in the fourth part (“Improving the Results” on page 51).

#### 4.3.1 Baseline Test

The first step in investigating the source identification functionality is to establish a baseline against which later test results can be compared. I will do this by conducting a test using the default values for the input parameters and examining the results.

**Baseline**

I performed a test consisting of $10^7$ runs using the values $\Delta \sigma_{\text{max}} = 1.5$ and $\sigma_{fuzz} = 0.15$. The measured total success ratios were $\Lambda \approx 92.4\%$ and $\Lambda' \approx 98.3\%$. Plots are provided for $\kappa_{\text{no}}(\psi)$, $\kappa_{\text{other}}(\psi)$ and $\kappa_{\text{new}}(\psi)$ where $\psi$ is the source density (figure 4.4 on page 45, panels a and b). New source detections in the $C' = F'$ part of the test (panel a) are at a good level but the presence of results without alerts is troubling. The $C = F$ part of the test (panel b) works quite well for low source densities (above 90% with no alerts) but the quality drops rapidly towards higher source densities\(^{17}\). $\Lambda$ and $\Lambda'$ have decent values but there is also room for some improvement.

\(^{17}\) When looking at these results it is worth keeping in mind the source density distribution (“Catalogue Sources ($C$ and $C'$)” on page 39). The vast majority of all outcomes concern just a few sources, even though a large area of the graphs cover results at the higher end of the density spectrum. The sampling
especially with the false new source alerts generated from $C \Leftarrow F$.

**Code Error**

The previous results show that there is a small but significant amount of outcomes of $C' \Leftarrow F'$ giving no alerts at all. This should not be possible. With $F'$ having one source more than $C'$ there should always be some sort of alert even if it is not of the correct type. This prompted me to do a thorough analysis of the source code. This analysis revealed an error in the software run by ISDC.

When the code matches sources it goes through a number of lists entry by entry\(^{18}\). As it does so it checks off each catalogue source that has become matched to a found source. A separate track is supposed to be initiated if another found source is being matched to a catalogue source that has already been checked. The problem is that the search for previously checked sources looks at the wrong list when source confusion matches are processed. The result is that the code erroneously suppresses many confused and new source alerts, producing outcomes with no alerts instead. I have reported this issue to ISDC which is now investigating the matter. I will eliminate the error in my own code for all further tests.

**Code Error Fixed**

The result of the baseline test improves noticeably when rerun after the code error has been eliminated. $\Lambda'$ increases from 98.3% to $\Lambda' \approx 99.96\%$, (figure 4.4 on page 45, panel c). The change for $C \Leftarrow F$ is more subtle (figure 4.4 on page 45, panel d). $\Lambda$ actually decreases somewhat from\(^{18}\) bias also means that the higher density outcomes are less reliable than the low density ones.

Starting with perfect matches, then confused sources, etc towards matches of lower quality.

\[^{18}\] Starting with perfect matches, then confused sources, etc towards matches of lower quality.
Figure 4.4: Baseline test results. \( \kappa_{\text{no}}(\psi) \), \( \kappa_{\text{other}}(\psi) \) and \( \kappa_{\text{new}}(\psi) \) are plotted, \( \psi \) is the source density. Results for \( C' \Rightarrow F' \) (panels a and c) and \( C \Rightarrow F \) (panels b and d) are shown. Plots with (panels a and b) and without (panels c and d) the code error are included. Please note that the curves are stacked.

92.4% to \( \Lambda \approx 92.0\% \) with the fix in place. But \( \kappa_{\text{other}} \) seems to have been transformed mostly into \( \kappa_{\text{new}} \) (false new source alerts) in the process.

### 4.3.2 Parameter Optimization

The next step is to find parameter values that bring down the number of false alerts while maintaining the good level of new source detections. I will begin by looking for an optimal value of \( \Delta \sigma_{\text{max}} \) and then look at \( \sigma_{\text{fuzz}} \). This is analogous to the testing period the analysis suite at ISDC underwent after the launch of INTEGRAL. Optimal values for the various parameters affecting analysis results were obtained when real data became available.

Finding optimal values for the parameters will be done by conducting \( n \) tests and tracking \( \Lambda_\mu \) and \( \Lambda'_\mu \). The lower index, \( \mu \), is used to indicate that \( \Lambda \) and \( \Lambda' \) are now calculated separately for 5 source density bands\(^{19}\) instead of one value for the full range of densities. This allows for a more detailed look at how the parameters affect source matching at different source densities.

The free parameter in a test series is advanced by \( \frac{1}{n} \) of the interval being tested between each test. Running large numbers of tests is computationally intensive and this forces me to keep \( n \), and thus the resolution of the results, fairly low (typically \( n = 100 \)). I vary the ranges of the parameters, starting with large and low resolution, then narrow and better resolution.

\(^{19}\) \( \mu = 3 \) for 3-5 sources, \( \mu = 6 \) for 6-8 sources, \( \mu = 18 \) for 18-23 sources, \( \mu = 25 \) for 25-35 sources and \( \mu = 70 \) for 70+ sources in the field of view. The notations \( \Lambda_\mu \) and \( \Lambda'_\mu \) refers to all 5 bands. I.e. I may use “\( \Lambda_\mu \) tends to...” as a shorthand notation when describing the collective behavior of the 5 curves describing \( \Lambda_\mu(x) \). \( \Lambda_\mu \) acts as a proxy for \( \Lambda \) because \( \sum \Lambda_\mu \times \pi_\mu \approx \Lambda \) where \( \pi \) is a weight factor.
The range allowed, as implemented at ISDC, for the parameter is \(-100 \leq \Delta \sigma_{max} \leq 100\) (table 4.1 on page 38). The first set of tests varies the parameter over the entire allowed range. The chance that a new source is correctly detected is essentially the same, regardless of the value of \(\Delta \sigma_{max}\) (figure 4.5 on page 47, panel a). This can be understood by considering the source matching process. If \(\Delta \sigma_{max}\) is very large then a found source will be attempted to be matched with all possible catalogue sources in the field of view. But as long as there are more found sources than catalogue sources there will always be at least one found source without a match which generates a new source alert. The exception to this is when the new source is too close to some other found source. This is why we see a small fluctuation near the 100% level of accuracy\(^{20}\). If \(\Delta \sigma_{max}\) is small then the new source is most likely located outside the search radius around the catalogue sources and thus it will get flagged as a new source by default.

\(\Lambda_{\mu}\) has a more interesting behavior (figure 4.5 on page 47, panel b). Each source density curve is flat apart from a small region close to \(\Delta \sigma_{max} = 0\). The lower the density, the less false new alerts are generated. The root cause for this behavior will be investigated in the next section. Each curve has its lowest point at \(\Delta \sigma_{max} = 0\) from which they rapidly increase as we move towards the positive or negative end of the scale. A positive value for \(\Delta \sigma_{max}\) signals the software to calculate distances between sources in units of the combined source position error. A negative value for \(\Delta \sigma_{max}\) means that the angular distance is used instead (equation 3.11 on page 28). In either case the outcome is symmetric. It is just the actual distance that a certain (absolute) value of \(\Delta \sigma_{max}\) represents that changes. The increase in false new source alerts close to \(\Delta \sigma_{max} = 0\) takes place because catalogue source-found source pairs fail to become matched when the search radius around the catalogue source is smaller than the source position error. We would see a corresponding increase in catalogue sources with no matching found source in this region.

Based on the previous set of tests one could conclude that any value of \(\Delta \sigma_{max}\) that is sufficiently large will produce the best possible outcome. It could, however, be useful to find out the lowest value of \(\Delta \sigma_{max}\) for which we get those ideal outcomes. In order to do so I need to narrow the range of the parameter to gain sufficient accuracy in the results. The second set of tests repeats the previous set with a narrower range; \(0 \leq \Delta \sigma_{max} \leq 5\) (figure 4.5 on page 47, panels c and d). We can see that the plateau is reached well before \(\Delta \sigma_{max} = 2\) so a third set of tests with \(0.5 \leq \Delta \sigma_{max} \leq 1.5\) is conducted (panels e and f). The plateau is reached at slightly different times at different source densities but a safe value that can be used is \(\Delta \sigma_{max} = 1.25\), slightly below the default value of 1.5. The exact point depends strongly on the shape of the distribution function of the source position error. I will use the best fit value in the following tests.

\(\sigma_{fuzz}\)

The second parameter has an allowed range \(0 \leq \sigma_{fuzz} \leq 1\), as implemented at ISDC (figure 4.6 on page 49, panels a and b). We can see that the various \(\Lambda_{\mu}\) (correct no alert cases) curves generally increase while the \(\Lambda'_{\mu}\) (correct new source alert cases) curves decrease with higher values of \(\sigma_{fuzz}\). The improvement in \(\Lambda_{\mu}\) is quicker than the deterioration of \(\Lambda'_{\mu}\). This suggests that an optimum value for \(\sigma_{fuzz}\) exists somewhere towards larger values. Increasing the value of

---

\(^{20}\) If the new source and the correct found source happen to be within the region specified by \(\sigma_{fuzz}\) then the software will not discriminate between them and the potential new source alert is transformed into a multiple/confused source alert instead.
Figure 4.5: $\Lambda'_\mu$ (left panels) and $\Lambda_\mu$ (right panels) as a function of $\Delta \sigma_{\max}$ for several values of $\mu$ (lines). The different rows of panels show the results for different $\Delta \sigma_{\max}$ ranges.
$\sigma_{\text{fuzz}}$ is also the first thing we have seen (panel b) that noticeably improves the results among the densest fields which have suffered from a very high rate of false new source alerts so far. Based on these tests it would be tempting to explore even higher values of $\sigma_{\text{fuzz}}$.

Ignoring, for the moment, the upper limit of $\sigma_{\text{fuzz}}$ I run a fifth set of tests with $0 \leq \sigma_{\text{fuzz}} \leq 20$. The most severe decline in $\Lambda_\mu'$ is to be found among the densest fields (figure 4.6 on page 49, panel c). $\Lambda_\mu(\sigma_{\text{fuzz}})$ and $\Lambda_\mu'(\sigma_{\text{fuzz}})$ seem to follow (nearly) logarithmic curves. This makes it tempting to extrapolate the optimum value of $\sigma_{\text{fuzz}}$ instead of performing test series after test series to look for it empirically. I use $\Lambda_{70}(\sigma_{\text{fuzz}})$ and $\Lambda_{70}'(\sigma_{\text{fuzz}})$ for this task and extrapolate the best fit logarithmic functions: $\lambda \sim \Lambda_{70}$ and $\lambda' \sim \Lambda_{70}'$ (equation 4.14). I look for the point where $\lambda \approx 1$ which is at $\sigma_{\text{fuzz}} \approx 1100$. A test series which includes $\sigma_{\text{fuzz}} = 1100$ confirms that $\Lambda_{70}$ is indeed very close to 1 at this point (figure 4.6 on page 49, panel f).

\[
\begin{align*}
\lambda(\sigma_{\text{fuzz}}) &\approx 0.1666 + 0.1191 \ln \sigma_{\text{fuzz}} \\
\lambda'(\sigma_{\text{fuzz}}) &\approx 0.9898 - 0.0103 \ln \sigma_{\text{fuzz}}
\end{align*}
\]  
(4.14)

For the sixth set of tests I then let $\Delta \sigma_{\text{max}}$ be the free parameter ($0 \leq \Delta \sigma_{\text{max}} \leq 3$) with $\sigma_{\text{fuzz}} = 1100$ to explore if some further adjustment to the first parameter is needed. We can see that $\Lambda_\mu'$ start to decline rapidly for $\Delta \sigma_{\text{max}} < 1$ (panel e) and that we reach $\Lambda_\mu \approx 1$ at $\Delta \sigma_{\text{max}} > 1$ (panel f).

The testing has revealed that we have a multiple choice situation instead of a clear-cut optimum solution for the parameter values. In low density fields ($< 20$ sources) it makes sense to use $\Delta \sigma_{\text{max}} = 1.25$ and $\sigma_{\text{fuzz}} = 0$. The consequence is that new source detection success rates in the highest density fields will be insignificant. Assuming that the user is willing and able to use an “illegal” value, $\sigma_{\text{fuzz}} = 1100$, then he should use $1.3 < \Delta \sigma_{\text{max}} < 3$. The exact choice depends on his willingness to suffer false new source alerts which increase with $\Delta \sigma_{\text{max}}$. The optimum for maximizing $\Lambda$ and $\Lambda'$ simultaneously is at $\Delta \sigma_{\text{max}} \approx 1.3$ but a higher value increases the sensitivity of detecting new sources, which may be desirable.

### 4.3.3 Exploring the Behavior of $\sigma_{\text{fuzz}}$

The fifth and sixth sets of tests (above) clearly showed that it is possible to dramatically reduce the number of false new source alerts (increase $\Lambda_{70}$ close to 100%) in dense fields of view by increasing the value of $\sigma_{\text{fuzz}}$ enough. Yet we know that the parameter has a maximum value of one in the live software suite. According to the parameter description: “Fractional difference in the relative distance of two found sources that allows them still to be considered equally good fits for a catalogue source”. Clearly it was thought that $\sigma_{\text{fuzz}}$ would always reside within $0 \leq \sigma_{\text{fuzz}} \leq 1$. Additionally there is anecdotal evidence that false alerts are quite common.

The first issue that has to be investigated is why we need a very high value for $\sigma_{\text{fuzz}}$ to effectively eliminate false new source alerts in dense fields of view. The second task is to find out why increasing the value of $\sigma_{\text{fuzz}}$ reduces the accuracy of detecting bona fide new sources ($\Lambda_\mu$).

#### False New Source Alerts

In order to investigate the first issue I extracted runs from the tests where $\sigma_{\text{fuzz}} = 100$ produced a false new source alert whereas $\sigma_{\text{fuzz}} = 1000$ did not. It turns out that these cases share a common geometry. They all involve two catalogue sources that are located very near each other.
Figure 4.6: $\Lambda'_\mu$ (first two panels on the left) and $\Lambda_\mu$ (first two panels on the right) as a function of $\sigma_{fuzz}$ for several values of $\mu$ (lines). The different rows of panels show the results for different $\sigma_{fuzz}$ ranges. The bottom two panels show $\Lambda'_\mu$ (left) and $\Lambda_\mu$ (right) as a function of $\Delta\sigma_{max}$ with $\sigma_{fuzz} = 1100$. 
Figure 4.7: Illustrating the false new alert problem. The test distance (“1/10 Test Distance”) is very small if an incorrect found source is detected nearly on top of a catalogue source. The correct found source will then fall outside the test distance and generates a new source alert (inset). The test distance has to be increased dramatically (“Test Distance”) to incorporate the correct found source and avoid the problem. The incorrect found source is a proper match for another, nearby, catalogue source (not shown).

Let us call these $s_{c|1}$ and $s_{c|2}$ and their matching found sources $s_{f|1}$ and $s_{f|2}$. Factoring in the source position errors of $s_{f|1}$ and $s_{f|2}$ we have a situation where $s_{f|2}$ has been detected almost on top of where $s_{c|1}$ is expected to be found and the correct match, $s_{f|1}$, a bit further away (figure 4.7 on page 50). With two found sources so close to a catalogue source the analysis ought to produce a source confusion alert. The reason why this does not happen, and a false new source alert is produced, is the manner in which the so called test distance is calculated. The test distance is calculated as $T_{i,j} = \sigma_{rel\mid i,j} \times (\sigma_{fuzz} + 1)$. In the example we are looking at the problem surfaces when $T_{1,2}$ (between $s_{c|1}$ and $s_{f|2}$) is used. If another found source, $s_{f|1}$, does not reside within $T_{1,2}$ from $s_{c|1}$ then it is not treated as an equally good match to that catalogue source. In this case the relative distance between $s_{c|1}$ and $s_{f|2}$ is very small and we need a value of $\sigma_{fuzz}$ that begins to approach infinity to include even a nearby source in the circle described by $T_{1,2}$. With the limit $0 \leq \sigma_{fuzz} \leq 1$ we then get instances of “orphaned sources”, especially in the densest fields where nearby pairs of sources are more common. The possibility that the test distance could shrink to such a degree that it becomes an overly stringent discriminator between sources was never considered when the software was designed and none of the test cases I ran at the time revealed this situation. It seemed logical that if $s_{f|2}$ was more than twice as distant from $s_{c|1}$ compared to $s_{f|1}$ then $s_{f|2}$ should not be considered an equally good match. The possibility that $s_{f|2}$ would reside (much) closer to $s_{c|1}$ than $s_{f|1}$ does was simply not considered at the time.

21 Please refer to Chapter three for more details.

22 The term can be defined as meaning a found source, that properly should be matched to a catalogue source, but was instead treated as a new source because another found source was detected closer to the catalogue source.
Loss of Genuine New Source Alerts

The second issue, the loss of real new source detections, is the inverse of the above. It happens when the new source is within the search radius but less than $\sigma_{fuzz} + 1$ times as distant from the catalogue source as another found source. It will then be classified as a source confusion and the new source alert is lost. The problem gets more severe the higher $\sigma_{fuzz}$ is set.

The tests I ran with a very high value for $\sigma_{fuzz}$ effectively turns the entire search radius into an area where no discrimination between found sources takes place. This is why the false new source alerts vanish. But on the flip side it also means that any genuine new source inside the search radius of another source only show up as a source confusion detection. Most new sources will reside outside the search radius of any catalogue source and these will continue to generate new source alerts properly. This is why we have an asymmetry between the increase in $\Lambda_\mu$ and the decrease in $\Lambda_\mu$. The outcome is, however, not satisfactory even though it is possible to locate an optimum value for $\sigma_{fuzz}$.

4.3.4 Improving the Results

Any attempt to further reduce the number of excess false new source alerts in dense fields by necessity involves changing the actual design of the source identification analysis. This is a large topic on its own and lies at the fringes of the scope of this thesis. One possible solution, for instance, would be to replace the current distance-based source matching with a more advanced algorithm. Some practical testing I carried out indicates that a method based on triangle matching [27] would produce superior outcomes. But in order to stay on topic I choose to present only some light modifications to the current algorithm that improves the outcome.

Suggested Modification

The root cause of the false new source alert problem is that the algorithm assigns too much significance to very small differences in distances between sources in some cases. We know that any found source within the search radius could reside somewhere else within that circle. One could then simply declare every found source within the search radius to be of equal significance and generate confused source alerts as necessary. But on the other hand we also know that a found source has a decreasing chance of being the correct match to a catalogue source the larger the distance between the two is. This decrease is also very rapid as we move away from the center of the circle (figure 4.3 on page 44).

Aware of these facts I choose to attempt a solution where a certain fraction, let us call it $\tau$, of the search radius is set as the minimum separation between sources that will be distinguished from each other. I.e. if the relative distance between a catalogue source ($s_{c|1}$) and a found source ($s_{f|1}$) is $\sigma_{rel}(1,1)$ then the relative distance to another found source ($s_{f|2}$) has to be $\sigma_{rel}(1,2) > \tau \times \Delta \sigma_{max}$ for $s_{f|1}$ and $s_{f|2}$ not to share the same matching rank. The new parameter is confined to the interval: $0 \leq \tau \leq 1$. The test distance will then be calculated in a piecemeal fashion (equation 4.15).

$$T_{i,j} = \begin{cases} \tau \times \Delta \sigma_{max} & \text{if } \sigma_{rel}(i,j) < \tau \times \Delta \sigma_{max} \\ \sigma_{rel}(i,j) \times (1 + \sigma_{fuzz}) & \text{if } \sigma_{rel}(i,j) \geq \tau \times \Delta \sigma_{max} \end{cases}$$ (4.15)

Otherwise it would not be included in the matching for the current catalogue source at all.
The seventh set of tests is carried out to see how $\tau$ affects the outcome. The other two parameters are kept at default values ($\Delta \sigma_{\text{max}} = 1.5$ and $\sigma_{\text{fuzz}} = 0.15$). The results are encouraging but clearly involves a trade-off (figure 4.8 on page 52). The denser a field of view becomes the more likely it is that new sources are found within the search radius around a known source (panel a). But as $\tau$ increases we lose the ability to distinguish these new sources from other sources. With $\tau = 1$ we lose $\approx 15\%$ of all new source alerts in the densest fields analyzed ($\Lambda'_70 \approx 0.85$). On the other hand $\Lambda_\mu$ steadily increases with higher $\tau$ values (panel b). Again, the effect is more dramatic in the higher density fields.

### Optimized Identification Test

Picking a suitable value for $\tau$ is a matter of choice. I choose to be conservative and preserve as many genuine new source alerts as possible. The choice will generate excess false new source alerts but hopefully less than with the live version of the analysis running at ISDC.

The most rapid loss of genuine new source alerts takes place for $\tau > 0.65$. Because of this I will use these parameter values$^{24}$ for the final two tests: $\Delta \sigma_{\text{max}} = 1.25$, $\sigma_{\text{fuzz}} = 0.1$ and $\tau = 0.65$. The values for $\Delta \sigma_{\text{max}}$ and $\sigma_{\text{fuzz}}$ are selected based on the parameter optimization carried out previously.

The results show a noticeable improvement in filtering out false new source alerts compared to the baseline tests (figure 4.9 on page 53 panel b). The tradeoff is an increase in other than new source alerts (panel a). The overall success rates are now $\Lambda \approx 98.45\%$ and $\Lambda'_7 \approx 99.83\%$. In addition the chance of being able to match each catalogue source with the correct found source without flaws is 97.19%. A summary of the results compared to the baselines is tabulated (table 4.5 on page 53).

### 4.3.5 Conclusions About the Source Identification Testing

The current version of `identify_srcs` running at ISDC is not very good (figure 4.4 on page 45, panels a and b). The system generates a large amount of false alerts of various types. The problem is especially pronounced in dense fields of view. On the bright side we can note that

$^{24}$ The $\sigma_{\text{fuzz}}$ tests show that new source detections are optimized for when using low values for the parameter (figure 4.6 on page 49).
Figure 4.9: Optimized parameters and $\tau$ introduced. $k_{\text{no}}$, $k_{\text{new}}$, and $k_{\text{other}}$ are plotted as a function of source density. Results for $C' \Rightarrow F'$ (panel a) and $C \Rightarrow F$ (panel b) are shown.

<table>
<thead>
<tr>
<th>Test</th>
<th>$\Lambda'$</th>
<th>$\Lambda$</th>
<th>Perfect matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline Test</td>
<td>98.32%</td>
<td>92.41%</td>
<td>92.06%</td>
</tr>
<tr>
<td>Code Error Fixed</td>
<td>99.96%</td>
<td>92.00%</td>
<td>93.36%</td>
</tr>
<tr>
<td>$\tau$ Introduced</td>
<td>99.83%</td>
<td>98.45%</td>
<td>97.19%</td>
</tr>
</tbody>
</table>

Table 4.5: Summary of the source identification tests. $\Lambda$ = the probability that no (false) alerts are generated when all found sources exist in the source catalogue, $\Lambda'$ = the probability that new sources are detected accurately and Perfect Matches = the algorithm is able to correctly identify all found sources and correctly alert for new sources without generating nuisance alerts.

the algorithm is at least capable of finding most (95%-98%, depending on source density) of the previously unknown sources. The main drawback is that what was intended to be a largely autonomous analysis in fact requires a lot of manual work. There are two causes for these problems;

1) The code has been edited by multiple people over a significant amount of time. At some point a mistake was made. Fixing this coding error improves the results (figure 4.4 on page 45, panels c and d). New source detection is now likely as good as it will get (99.9%) but the rate of false alerts is still alarmingly high.

2) The second cause is a flaw in the theoretical underpinnings of the analysis. The theoretical model was developed largely without any empirical analysis of the results. Such testing would have been possible during code development but at this stage there was no suitable test data available. I've attempted a fix for this problem by introducing the $\tau$ parameter. The improvement over the baseline scenario and the results after the code fix is quite noticeable (figure 4.9 on page 53).

Repairing the code and introducing $\tau$ would clearly be very useful for the QLA analysis. But even with these measures in place the analysis, as a fully automated tool, is only effective at low source densities. At around 10 sources, and above, the rate of false alerts starts to increase and reaches 40% in the densest fields.

4.4 Grid Search Testing

The task of the grid search is to calculate a correction for the pointing error. It accomplishes this task by shifting the found sources, by a small increment each time, and asking the source...
identification analysis to count the number of successful matches. These tests are carried out over the entire area within which the true pointing of the instrument can exist. The offset of the location that results in the highest number of matches is used to correct the pointing error. The next set of tests will look into this process by running a series of simulations where the input parameters are allowed to vary. The resulting corrections are then compared to the proper correction and the behavior of the grid search is examined.

### 4.4.1 Initial Test Setup

The number of parameters in the grid search is five. From the previous sections we know that $\Delta \sigma_{\text{max}}$ and $\sigma_{\text{fuzz}}$ affects source matching in general. The grid search algorithm adds a new parameter: the number of grid cells. I will call that number $n_{\text{grid}}$. The total number of matching source pairs ($N_{\text{tot}}$) affects the grid search and will be used as a parameter. The grid search also requires a maximum search radius ($r_{\text{error}}$) as an input. The latter value can be chosen by the operator of the software but $r_{\text{error}} \propto \rho_{\text{pointing}}$ because the search is most accurate if restricted to roughly the highest possible pointing error of the spacecraft. I find in literature that $\rho_{\text{pointing}} \leq 5'$ [23]. I will use a slightly larger value to be on the safe side, $r_{\text{error}} = 6'$. I therefore end up with four free parameters when I avoid the complication of new found sources and missing catalogue sources.

The initial simulation allows all four parameters to be free in order to give a (very rough) sense of their impact on the grid search. This simulation is quite coarsely grained\(^{25}\), $2 \leq n_{\text{grid}} \leq 15$ (step length = 1), $0.1 \leq \sigma_{\text{fuzz}} \leq 1.0$ (step length = 0.1), $0.5 \leq \Delta \sigma_{\text{max}} \leq 1.5$ (step length = 0.1) and $N_{\text{tot}} \in \{4, 7, 21\}$. The simulated polar coordinates for $\rho_{\text{pointing}}$ were, in all cases, $(4.5', \pi/4)^{26}$. The result of the grid search is the coordinate correction needed to align the found sources with the catalogue sources so the desired result is $(-4.5', 5\pi/4)$.

### 4.4.2 Initial Test Results

Plotting the results of the simulated grid search and comparing them with the desired correction reveals two things (figure 4.10 on page 55). 1) Almost all simulations provide reasonable estimates of the $\phi$-coordinate but a subgroup is slightly better aligned with the target. 2) There is a very broad range of outcomes for the $r$-coordinate.

I calculate two accuracy measurements (equation 4.16) but use the one based on the $r$-coordinate because the variation is larger in that dimension. I then plot each free parameter versus the corresponding $|\gamma|$-value (figure 4.11 on page 55). The number of cells the grid is divided into ($n_{\text{grid}}$) doesn’t seem to correlate at all with the accuracy of the fit. The same is true for $\sigma_{\text{fuzz}}$. The most interesting parameter is perhaps $\Delta \sigma_{\text{max}}$. There is a general loss of accuracy as the parameter increases beyond $\sim 1.0$ but there seems to be a potential for more complex behavior. The plot for $N_{\text{tot}}$ hints at an increase in accuracy with higher $N_{\text{tot}}$ but the scarcity of data points at this point prevents any firm conclusions.

\[^{25}\text{Due to restrictions on computing time.}\]
\[^{26}\alpha, \delta \approx 0.053.\]
Figure 4.10: $\alpha, \delta$ plot of the simulated grid search outcomes when all four parameters are allowed to be free.

Figure 4.11: Correlation between the accuracy of the fit ($|\gamma|$) and the value of the four free parameters.
4.4.3 Detailed Test Setup

With two of the four parameters shown to be more or less irrelevant it then becomes possible to proceed with more fine-grained simulations. I’ll perform 1,000 simulations of the grid search for each of $0.01 \leq \Delta \sigma_{\text{max}} \leq 1.5$ with a step length of 0.01. I will do this for fields with five levels of source density (4, 7, 21, 30 and 80). The fixed parameters will be set to: $\sigma_{\text{fuzz}} = 0.1$ and $n_{\text{grid}} = 10$.

4.4.4 Detailed Test Results

The results reveal a somewhat complex picture (figures 4.12 and 4.13). The accuracy of the fit quickly reaches a very good level ($\gamma \sim 0\%$ and $\vartheta \sim 0\%$) as $\Delta \sigma_{\text{max}}$ grows from its minimum value. $\gamma$ and $\vartheta$ then overshoot the desired level up to, depending on $N_{\text{tot}}$, $\Delta \sigma_{\text{max}} \sim 0.5$ or $\Delta \sigma_{\text{max}} \sim 1.0$. Past this point the accuracy drops rapidly, especially in the $r$-dimension, except for the highest source density simulation. I will follow the 21 source density curve (for the $r$-coordinate) when investigating the causes of this behavior;

$0.01 \leq \Delta \sigma_{\text{max}} < 0.03$

The source position error would have to be trivially small for sources to become matched when $\Delta \sigma_{\text{max}}$ has a very low value. In most cases we don’t get any matches at all because of this. The grid search will then use the default correction $r_{\text{sim}} = 0, \varphi_{\text{sim}} = 0^\circ$ giving $\gamma = -1$ and $\vartheta = -1$. As $\Delta \sigma_{\text{max}}$ increases a little we rapidly get at least a few matches. These matches can only take

---

This number had to be reduced to 100 for the 2 densest setting of $N_{\text{tot}}$ for practical reasons.
Figure 4.13: $\vartheta$ as a function of $\Delta \sigma_{\text{max}}$. $\vartheta = 0$ indicates a correct estimate for $\varphi$, negative values represents too small estimates and positive values too large estimates of $\varphi$.

place when the grid search looks at locations very near the correct coordinates $r_{\text{sim}} \approx r_{\text{real}}$ and $\varphi_{\text{sim}} \approx \varphi_{\text{real}}$. We will get a narrow, well defined, peak in the grid (figure 4.14 on page 58) and $\gamma \sim 0$, $\vartheta \sim 0$.

$0.04 \leq \Delta \sigma_{\text{max}} < 0.9$

It becomes easier to make matches as $\Delta \sigma_{\text{max}}$ increases. This also means that the grid search becomes less sensitive as an increasing number of grid cells near the best possible match “saturates”\textsuperscript{28}. The peak broadens and has become a rather flat plateau towards the end of the interval (figure 4.15 on page 58).

The shape of the resulting $\gamma$- and $\vartheta$-curves in this interval reflects the way the grid search calculates the coordinate correction. 1) It counts the number of matches it can make at the default position ($\Delta \alpha = 0$, $\Delta \delta = 0$) and registers that number as the best one so far. 2) It loops over $\Delta \alpha$ from the most negative correction towards the most positive one. It then loops over $\Delta \delta$ in the same manner (panel a of figure 4.16 on page 59). If a grid cell is encountered which contains more matches than the current best amount then that cell is stored as the best attempt so far. 3) The grid cell stored as containing most matches is used for correcting the position of all sources once the grid has been scanned. The result can be seen in the image (figure 4.16 on page 59). The circles in the images represent the area that has saturated at the maximum number of matches and the arrow with a dashed line indicates the spot that will become selected in each case (panels b, c and d).

I happened to choose the coordinates ($-4.5', 5/4\pi$) as the correct coordinate correction for my testing. This means that the saturated area will be reached from the lower left (panel a of
\textsuperscript{28} Contains as many matches as the best cell.
Figure 4.14: The number of matches found at each $\Delta \alpha, \Delta \delta$ point in the grid search. The result is for a low $\Delta \sigma_{\text{max}}$ value generating a well defined peak. The highest number of matches (2-3) is well below the maximum possible number of matches for this data set (7).

Figure 4.15: The number of matches found at each $\Delta \alpha, \Delta \delta$ point in the grid search. The result is for a high $\Delta \sigma_{\text{max}}$ value generating a broad plateau of best fit grid cells.
Figure 4.16: How the grid search determines the pointing error. The coordinate system sets out the $\Delta \alpha, \Delta \delta$ space around the middle of the observed image. The black arrow indicates the correct pointing error. The circle around the tip of the black arrow is the region of the grid with the highest number of matches. Panel a shows how the grid search, row for row, looks for the highest number of matches with the short arrow indicating the cell it will return as the result. This is simplified in panels b-d with the dashed line arrow directly indicating the point the grid search will end up selecting.
The result is a correction with too high $r$-value. The cell chosen by the grid search will also roughly lie in the $5/4\pi$ direction, explaining the better fit in the $\varphi$-dimension. This coincidence explains why the $\gamma$ curve lies above 0, and increases, when $\Delta\sigma_{\text{max}}$ grows towards 1\textsuperscript{29}. The $\gamma$ curve would have been below 0 if the correct coordinate correction had been in the first quadrant instead of the third one. In that case the grid search would have approached the saturated area from the origin and ended at an $r$ value smaller than the correct one (panel b of figure 4.16 on page 59). Had the correct point been closer to one of the coordinate axes\textsuperscript{30} then we would have had higher uncertainty in the $\varphi$-coordinate and lower uncertainty in the $r$-coordinate compared to my test case (panel c of figure 4.16 on page 59).

$$\Delta\sigma_{\text{max}} \geq 0.9$$

Above some threshold value of $\Delta\sigma_{\text{max}}$, which depends on $N_{\text{tot}}$, the origin becomes incorporated in the saturated area (panel d of figure 4.16 on page 59). The algorithm will always select the default correction (0, 0) if this happens. $\gamma$ and $\vartheta$ will start to drop towards -1. A higher total number of sources gives better statistics and the grid therefore saturates less rapidly when $N_{\text{tot}}$ is higher. This is why the turning point towards minus one is different for the 5 different source densities simulated.

### 4.4.5 Conclusions About the Grid Search Testing

The reasons for the, likely unanticipated, complications following from the saturation phenomenon are twofold. 1) The pointing error of the spacecraft is similar in magnitude to the combined source position error ($\rho_{\text{pointing}} \sim \rho_{\text{tot}}$). 2) The possibility of multiple grid cells sharing the best possible number of matches was not taken into account during design and development. This suggests to me that the expectation was that $\rho_{\text{pointing}} > \rho_{\text{tot}}$. A suitable modification of the grid search would be to look for multiple cells with the highest possible value and return the average coordinate adjustment from these instead of the current methodology. The fact that the tuning parameter, $n_{\text{grid}}$, for the grid search has virtually zero impact on the processing is noteworthy.

\textsuperscript{29} The saturated area expands towards the lower left corner, giving increasingly larger $r$-values.

\textsuperscript{30} A larger $\Delta\alpha$ than $\Delta\delta$ or vice versa.
Summary

“For even the very wise cannot see all ends.”
– J.R.R. Tolkien, The Fellowship of the Ring

Chapter One introduces the INTEGRAL mission and the subject of this thesis, the Quick-Look Analysis. The QLA is placed in its context as a rapid, automated, analysis aimed at detecting new sources and transient events.

Chapter Two explains why certain choices were made in regards to the framework of architecture, programming language and data structures that surrounds the QLA components. Descriptions of these key building blocks are also provided. The Chapter provides the reader with the terminology adopted by ISDC with which the QLA can be described.

The QLA tools are placed in their context within the two QLA pipelines at the start of Chapter Three. These pipelines handle the automated analysis starting from the arrival of new data from INTEGRAL and ending at the point where the processed data is stored in the data archive and possible alerts are issued. Five generic QLA tools are then described in detail. $Q_{\text{build\_scw\_list}}$ forms an observation group from the newly arrived science window/s. That group is passed to the relevant ISSW for data reduction. $Q_{\text{flag\_srcs}}$ then picks out all sources that should have been observed, or might otherwise impact the observation. $Q_{\text{set\_src\_fluxes}}$ finds out what flux levels one should expect from the observed sources. With this information available the ISSW components finish their data reduction work and source identification begins. $Q_{\text{identify\_srcs}}$ matches the found sources, as well as it can, to the catalogue sources. New sources are reported. $Q_{\text{match\_src\_fluxes}}$ finally checks if there have been relevant changes in the flux level or hardness ratio of the sources. Significant flux deviations compared to the catalogue are reported in the form of alerts.

The Fourth Chapter puts $Q_{\text{identify\_srcs}}$ to a series of tests in order to establish its performance. A key finding of this thesis is that a software error in the current implementation prevents the detection of new sources under certain circumstances. Further testing, with the error removed, shows that the QLA is capable of fulfilling its primary goal of reliably identifying the detection of new sources. The second key finding of this thesis is that the design underlying the source identification analysis is flawed. An excess of false new source alerts is the result. A possible software based solution is presented that at least partially remedies the problem. With fixes in place the source identification analysis produces acceptable results in lower density fields where it can be trusted to handle data processing autonomously.

Testing of the grid search component shows that it is unlikely to produce meaningful corrections for the pointing error in most cases. A small spacecraft pointing error, similar in magnitude to
the source position error, does not necessarily need to be corrected for. The source matching algorithm can do the job. But some modifications to the grid search would seem to nevertheless be in order.

5.1 Concluding Remarks

On the one hand most QLA components have proven to be well designed and robust in use with no changes in their code for over a decade. On the other hand this thesis has revealed some fairly significant problems in the source identification component of the QLA. The main difference is that the source identification is the only component that doesn’t have an unambiguous correct / incorrect outcome. If there was an error in the other components then it would be immediately obvious to the users. The source identification analysis is also the most complex component. There are several interacting mechanisms; relative distance calculation, ranking, matching and the grid search. Too much trust was placed in the theoretical design of these mechanisms and too little verification was done. Verifying the design of each one of these on its own would have been doable once real data became available. But when these mechanisms interact it becomes quite difficult just noticing that there is a problem, let alone tracking it down. Functional testing was only performed with data made up by the developers. Rigorous step by step testing using real data, as in this thesis, should have been carried out.

But the problem runs deeper. I briefly mentioned programming language generations in Chapter Two. It was the added level of abstraction achieved by transporting the original C-language code to Java that allowed me to begin pinpointing the problems and explore alternatives. It is noteworthy that the ESA coding and testing standards have developed from discouraging the use of object-oriented languages at the time of INTEGRAL into requiring their use today. From a computer sciences point of view this is not surprising. The development of object-oriented languages happened as a direct response to an increase in software complexity. ESA is, if anything, lagging behind the state of the art.

What is really worrying is that astronomers, as a community, lags even further behind. Software is likely the second most important tool right after the telescopes themselves for the profession. Yet an astronomers education may still contain just a handful of credits in computer sciences and focus exclusively on outdated tools and languages. The Gaia software project had to bring in professional software engineers to write the code. Astronomers were relegated to merely providing the necessary physics models. But what will the astronomers do if the physics model is never properly verified before the software engineers leave the project, like we have seen examples of in this thesis?
Bibliography


Appendix I

General executable design

Figure 5.1: Typical QLA executable design

Overview

All executables described in this thesis follow the same basic design (figure 5.1 on page 67). The underlying principle I have chosen to follow is to keep each identifiable task\(^1\) in its own function to maintain clarity and simplify testing. This leads to an executable layout that contains several functions which are repeated across all the executables; \texttt{q-***_main}, \texttt{q-***_comments}, \texttt{q-***_parameters}, \texttt{q-***_open}, \texttt{q-***_close} and \texttt{q-***_work}. Some departures from this general design have been made over time. The \texttt{q_set_src_fluxes} and \texttt{q_match_src_fluxes} executables now read the ARF and RMF\(^2\) files as part of their \texttt{q-***_main} functions instead of using either the \texttt{q-***_parameters} or \texttt{q-***_open} functions as intended. I will briefly describe each one of these generic functions in this Appendix.

\(^1\) These tasks are quite high level and the readability of the code would have benefited from a more fine grained division.

\(^2\) See Appendix IV and Appendix VI.
The script or pipeline activating the executable will automatically start the q-***_main function. The processing is quite simple (figure 5.2 on page 68); declare a STRUCT\textsuperscript{3} to hold parameter data, call CommonInit\textsuperscript{4}, call q-***_parameters to read the parameters, call q-***_work to carry out the core processing of the executable, process any error codes that have been generated, call CommonExit\textsuperscript{5} with the current status and then finally exit back to the calling script or pipeline.

### Q-***_comments

The ISDC CTS calls for quite liberal logging activity. Each function call or logically divisible execution step should be followed by a RIL call to log events. I chose to place the logging in its own function (figure 5.3 on page 69) to keep all this code overhead neat. The structure of the q-***_comments functions is best illustrated by a short example:

```c
switch(commentNr) {
    case 714:
        strcpy(prText, " A DAL-API in q_build_scw_list_read has ");
        strcat(prText, " returned: \%");
        if ( chatty > Q_CHATTY_VERY_QUIET ) {
            RILlogMessage(NULL,Error_2, prText, status);
        }
        break;
}
```

The q-***_comments functions are mainly very long switch / case constructs allowing the calling function to select the appropriate logging message simply by passing a number to the function. This function is shown separately, without connections, in the overall figure as this support function is called by all other functions within the executable multiple times. Showing these connections explicitly in the figure would have made it overly complicated.

\textsuperscript{3} A C-language memory construct that can contain an assembly of heterogeneous variables. These can be seen as a step towards objects. A fully fledged object differs from a STRUCT by having functions incorporated in the construct.

\textsuperscript{4} A function call required by all ISDC executables. Part of the Common support library.

\textsuperscript{5} Same comment as for CommonInit.
Figure 5.3: $Q_{***\_comments}$ function execution diagram
An important feature of all functions is visible at the start of the next code example. One of the CTS requirements is that every executable closes down in a controlled fashion if an error is encountered. A central part of this error handling is a check of the current status at the start of each function. This check will terminate execution of the function if some error has taken place previously in the stack of function calls. This allows the function to be called, even if something has gone wrong previously, but does not expose it to any potentially problematic by-products from upstream.

The task of the \texttt{q***_parameters} functions is to read the input parameters passed to the executable by the system (figure 5.4 on page 71). The function uses PIL function calls to do this. Input parameter specifications are hard coded because the executable drives what it needs and the system has to supply it. The PIL library provides functions to read or write parameters of all data types. A string type parameter is read in the example below followed by the required logging action. The logic tests indicated in the image represent sanity checks on the input values. These could, for instance, be a check against some high and low limit on the value.

```c
/* Immediate exit if input status is not ok */
if (status != ISDC_OK) {
    q_build_scw_list_comments(200, (*inputParams).CHATTY, status);
    return status;
}
/* <Snip> */
/* Read the DOL for the input index table */
status = PILGetString("inputIndex",
    (char *) &((*inputParams).INPUT_DOL));
if(status != ISDC_OK) continue;
q_build_scw_list_comments(203, (*inputParams).CHATTY, status);
```

\textbf{Q***_open & Q***_close & Q***_work}

The \texttt{q***_open} functions deal with opening files for use by the other functions as well as checking basic file information like listing the science windows attached to an observation group. It also creates, if needed, new files for executable output storage (figure 5.5 on page 71). The \texttt{q***_close} function is the reverse of \texttt{q***_open} and it is also a very simple function. Most often it consists of just two DAL function calls; to close the input and output files. An important feature with this function is that unlike all other functions, it does not exit if an error code was passed to it from the calling function. Open files are closed under all circumstances where the code is still capable of being executed (figure 5.6 on page 72).

The \texttt{q***_work} functions are mainly wrapper functions for \texttt{q***_open}, \texttt{q***_close} and the core functions of the executable. Arguably, this functionality could have been placed in the \texttt{q***_main} functions just as well (figure 5.7 on page 72).
Figure 5.4: $Q_{***}$-parameters function execution diagram

Figure 5.5: $Q_{***}$-open function execution diagram
Figure 5.6: $Q_{***\_close}$ function execution diagram

Figure 5.7: $q_{***\_work}$ function execution diagram
Appendix II

Q_build_scw_list

The core functionality of Q_build_scw_list is contained in two functions; Q_build_scw_list_build and Q_build_scw_list_read (figure 5.8 on page 73).

Q_build_scw_list_build

Memory is allocated to two new memory structures: inputData and resultMatrix (figure 5.9 on page 74). Both are one-dimensional lists. A call is made to the Q_build_scw_list_read function to fill the inputData array (“Q_build_scw_list_read” on page 76). One of the science windows in the inputData list will be the reference science window against which the other science windows are compared. This science window is located based on its ID number which is provided as an input parameter.

Execution enters two loops over the entire inputData array (figure 5.10 on page 75). For each line the right ascension and declination values (α,δ) are read. These coordinates are compared against the coordinates of the reference science window to determine if the angular distance
Figure 5.9: Q_build_scw_list_build execution diagram, main section
Figure 5.10: Q_build_scw_list_build execution diagram, loops
between the two science windows are within limits specified in the input parameters. The corresponding entry in the resultMatrix list is marked if they are close enough to each other.

The second loop reads the values for “Start OBT” and “End OBT” for both the comparison science window and the one being checked. OBT stands for onboard time and is the time measurement used by the satellite itself. These two time stamps marks when a pointing started and when it came to an end. Both the difference between the start times and the end times are compared to a maximum value specified in input parameters. Only those science windows which are close enough to each other in time are grouped.

Execution returns to the previous flow once the flagging and unflagging loops have completed (figure 5.9 on page 74). The total number of science windows that have been flagged are counted. The next step is bypassed if the number does not exceed a threshold value specified in input parameters. Otherwise all flagged science windows in the input file are attached to the output file. Memory allocations are then freed up and the function terminates.

**Q_build_scw_list_read**

This is a file I/O function that was split out of the q_build_scw_list_build function. The way DAL specifies file handling requires that the reading function allocates transfer buffers of the correct length before reading data. Execution begins by analyzing the column structure for the right ascension, declination and science window information to be retrieved (figure 5.11 on page 76). Three buffers of correct size are allocated and the data is read through DAL. The buffers are looped over and the $\alpha$, $\delta$, science window ID and OBT data are copied to the inputData array. A limited check of the data is carried out, ensuring that the OBT times are single timestamps. The allocated memory segments are freed up and execution passes back to the calling function.
Appendix III

$q\_flag\_src$

![Diagram of $q\_flag\_src$](image)

Figure 5.12: $q\_flag\_src$, overview

The main tasks of this executable are carried out in $q\_flag\_src\_compare$ (figure 5.12 on page 77). The purpose of the executable is to allow the user to make various source selections. These selections can be any logic operation listed (AND, OR, ANDNOT, ORNOT, NOT) between two groups. The first group consists of all sources where the catalogue sources and found sources have a matching ID. The second group consists of all found sources that are spatially located within a certain angular distance around the catalogue sources. The tools for making these selections are found in the DAL3CAT library which uses masks for the selection operations.

$Q\_flag\_src\_compare$

This function allows the user to select which logic type will be used when combining the masks (figure 5.13 on page 79). The default is set to AND if no other logic is specified. Memory is allocated to four masks (name, temporary position, position and result) and the masks are all initialized to FALSE. Two memory structures are allocated to hold found source and catalogue source data. The corresponding data is read from file.
A nested loop over both catalogue sources and found sources compares the source ID’s (names) and toggles the flag in the name mask to TRUE if the ID’s match. The next loop walks through the catalogue data source by source and checks which found sources happen to be within the error radius as specified in the input parameters. The reason for using two masks for this task is that the DAL3CAT function, which selects the sources, returns the results in a mask (the temporary mask). Each instance of the temporary mask is summed up with the position mask to create a mask flagging every found source within the error radius of each catalogue source. The name and position masks are combined to create the result mask using the logic selected at the start. All found sources that remain flagged in the result mask have their state set to “required” and the content of the result mask is saved together with the file containing found sources. All dynamically allocated resources are freed up and the function terminates.

---

6 Not to be confused with the source position error. This error radius is a value specified by an input parameter for the component and can be freely set to any value desired.
Figure 5.13: Q_flag_src_compare execution diagram
Appendix IV

\textit{Q\_set\_src\_fluxes}

The core function begins (figure 5.15 on page 82) by determining which instrument has been used\footnote{Since February 2003, QLA analysis is only done for ISGRI and JEM-X data.}. The corresponding file containing flux/rate information is opened. These have been previously measured by INTEGRAL. The source names and their corresponding rates are read from the history file. The upper and lower energy boundaries for each instrument is then read from the catalogue file and a function call to \textit{q\_set\_vector\_energy\_boundaries} is made. The set vector energy boundaries function has just one task; to read the \(E_{\text{min}}\) and \(E_{\text{max}}\) information from the found sources file (figure 5.16 on page 82).

\textit{Q\_set\_src\_fluxes\_flux}

Figure 5.14: \textit{Q\_set\_src\_fluxes}, overview
Figure 5.15: $Q_{set\_src\_fluxes\_flux}$ execution diagram, part 1

Figure 5.16: $Q_{set\_vector\_energy\_boundaries}$ execution diagram
The function applies default values to $E_{\text{min}}$ and $E_{\text{max}}$ in case there are no sources in the results file (figure 5.17 on page 83). These were read previously from the catalogue file.

The final part of $q_{\text{set src fluxes flux}}$ (figure 5.18 on page 84) determines the flux for each bin and then writes the flux data to the output catalogue file. All catalogue sources are looped over. For each source the history data is scanned to see if there is a match. The previously measured count rate data of the source is copied to an array, called flux, in case there is a match.

The second option, if no match can be made, is to calculate the count rates based on the x-ray spectrum model attached to each catalogue source. The function reads the model and the related model constants. It then loops over all flux bins and, after a sanity check on the bin energy boundaries, calls on $q_{\text{set src fluxes model}}$ to calculate the count rate of that bin. The result is multiplied by a renormalization factor if the instrument used is JEM-X.

Once the function has data for all flux bins it writes it to the RATE or FLUX column in the catalogue file. Once all catalogue sources have been processed the function ends.

$Q_{\text{set src fluxes model}}$

The top level of this function has just one task (figure 5.19 on page 85): to identify which instrument was used and then call on the relevant XXX_cntrate function to calculate the bin count rate. A default value is returned if the instrument number parameter is faulty.

The four different cntrate functions are very similar to each other (figure 5.20 on page 85). The main difference is the set of parameters passed downwards to the CalFlux function which is partially instrument specific. Two count rates are calculated for PICS, with different sets of parameters, and the results are summed together.

CalFlux (figure 5.21 on page 86) begins by setting up memory structures. The function then calls the CalPhotonSpec function to calculate the photon spectrum based on the spectral model. The CalPhotonSpec function (figure 5.22 on page 87) calculates the photon spectrum by multiplying the width of the energy channel by the average of the flux at the upper and lower energy boundary (equation 5.1). These fluxes are calculated by the CalcModel function discussed below.
Figure 5.18: $Q_{set_{src\_fluxes\_flux}}$ execution diagram, part 3
Figure 5.19: Q_set_src_fluxes_model execution diagram

Figure 5.20: Execution diagrams for the different cntrate functions
Figure 5.21: CalFlux execution diagram
Figure 5.22: CalPhotonSpec execution diagram

\[(E_{\text{max}} - E_{\text{min}}) \times \frac{\text{flux}\_\text{high} + \text{flux}\_\text{low}}{2} \] 

(5.1)

CalFlux is now able to calculate the expected counts in each detector channel (figure 5.21 on page 86). This is performed inside a loop over the channels. A check is performed to see if the SPI response file is a unit matrix\(^8\). The function copies the related entry from the photon spectrum into the detector channel spectrum if this was the case. The function loops over the photon spectrum and sums up the chance, for each energy bin, that photons would end up in them, if not. The latter is done by multiplying the photon flux in each channel by the quantum efficiency\(^9\) of the detector. This flux will not all end up in the correct energy channel. Some photons will spill over into other, nearby, channels. The function will need to check which fraction of the photons end up in the channel under examination\(^10\). The channel flux rate is stored in the returned memory structure if the upper and lower energy boundaries are within the range accepted in the function parameters.

Model fluxes

CalcModel (mentioned above) calculates the model photon flux for each energy channel of the photon spectrum, before the effects from instrumentation is considered. The model for each source is stored in two data structures. One will list the model elements separated by either a “*” or a “+” symbol indicating multiplicative and additive model elements. The other will list the constants for each model as a list of values. The first part of CalcModel (figure 5.23 on page 88) begins with the usual setup section. It then goes through all the multiplicative elements of the model description and chops out each individual element as its own entry in the eachModel array. The same is done for the additive elements.

The function can now loop over the model elements (figure 5.24 on page 89). It determines which type of element is in the eachModel[i] entry. The number of constants varies by model type. They are read from the second data structure, increasing the counter keeping track of where in the structure execution is progressing for each constant read. CalcModel then calls on

---

\(^8\) I.e. the first entry in the RMF matrix was set to a negative value.

\(^9\) From the ancilliary response file (ARF).

\(^10\) The probability data is located in the redistribution matrix file (RMF).
Figure 5.23: CalcModel execution diagram, part 1
the appropriate model element function to calculate the flux. This value is added or multiplied to the add/multi variables as appropriate. The add and multi values are finally multiplied and the result is returned to the calling function.

The models

The functions encapsulating the different model elements are quite brief (figures 5.25, 5.26, 5.27, 5.28 and 5.29 beginning on page 90). They contain the mathematical expression of the physical model that enables calculating the flux in question.
Figure 5.25: `bknpower` execution diagram

Figure 5.26: `cutoffpl` execution diagram

Figure 5.27: `highecut` execution diagram

Figure 5.28: `powerlaw` execution diagram

Figure 5.29: `wabs` execution diagram
Appendix V

Q\_identify\_srcs

Overview

The responsibility of this executable is to compare new detections with the INTEGRAL catalogue data. A determination is made for each found source: has it been detected previously and can it be uniquely matched with a catalogue source? This process is complicated by the fact that there is a limit to the precision for which the pointing of the spacecraft can be determined. Due to this uncertainty it is possible that the image does not have the same center point as the corresponding view of the catalogue data. If the pointing error is large enough it would cause a number of bad identifications when we try and match found sources with catalogue sources. To correct for this error a so called grid search (“Grid Search” on page 30) is carried out. A schematic of the execution flow (figure 5.30 on page 91) shows that the main tasks are carried out in the q\_identify\_sources\_identify and q\_identify\_sources\_match functions.
Figure 5.31:  *Q_identify_sources_identify*: memory allocations and data input

**Q_identify_sources_identify**

The tasks of *Q_identify_sources_identify* are: to find the best possible correction for the pointing error of the spacecraft, apply this correction to the found sources, call *q_identify_sources_match* to match found sources with catalogue sources, issue alerts for new detections as well as non-detections and finally store all results.

The function initializes a number of memory structures at the beginning of execution (figure 5.31 on page 92). These will hold the name, catalogue ID, declination, right ascension and the error radius of both catalogue sources and found sources. These structures are filled with corresponding data from the input files. The first call of *q_identify_sources_match* is made to find out how many matches between catalogue sources and found sources can be made (*numMatches*) using the default pointing coordinates.

In the next section of the code (figure 5.32 on page 93) the grid search is carried out by looping over the grid in both the declination and right ascension dimensions and then translating the coordinates of the found sources correspondingly. A call is made to *q_identify_sources_match* to determine the number of matches made for each coordinate shift. The shift in coordinates is stored if the current coordinate shift gives a better result than previous attempts.

A check is carried out to see if the default coordinates gave the best match once the grid search
Figure 5.32: $Q_{\text{identify\_sources\_identify}}$: the grid search
finishes (figure 5.33 on page 95). Several changes, including the coordinate changes that gave the best fit, are made to the output file header if the default coordinates were not the best solution. An alert is also raised to inform the users of the system that the default coordinates were replaced. The coordinates of the found sources are recalculated using the shift in coordinates that gave the best fit. A final call to `q_identify_sources_match` is made to obtain its second output: the `matchList` object.

A loop over the found sources is run to create output data. The `matchList` is read for each found source to see; which catalogue source the found source has been paired with, if it is a new source, if multiple catalogue sources match the same found source or if multiple found sources match the same catalogue source. An alert is created unless a one to one match has been achieved. The results are then stored in the output file.

The only task left is to determine if some catalogue source/s has/have not been detected (figure 5.34 on page 96). The execution consists of summing up the total flux for each catalogue source and comparing it with a flux limit given as an input parameter to the executable. The source is added to a list if the catalogue source has a higher flux than this threshold value and no found sources matches the catalogue source. An alert is created to inform the system if there are missing catalogue sources once all sources have been looped over. All dynamically allocated memory resources are freed up and the function exits.

### Q_identify_sources_match

This function has three distinct tasks, visible among the input parameters (figure 5.35 on page 97). It needs to count the number of matches it is possible to make, given the restrictions set by input parameters, between found sources and catalogue sources (`matchNumPtr`). It will provide a list of matches where each catalogue source and found source has been attempted to be matched. For each matched catalogue source - found source pair a code expressing the quality of the match is kept track of (`matchListPtr`). I will treat the `matchListPtr` structure as two separate tables in this text, for ease of understanding, even though it is a single memory construct (Please see Tables 5.6 and 5.7 on page 100). A matrix expressing the distance between matched catalogue source - found source pairs is calculated (`matchDist`).

To achieve these tasks the software executes a number of steps that I will describe below. A number of memory structures are declared to aid in the processing (“Memory allocation” on page 94). This is followed by calculating the relative distance between every possible pairing of catalogue sources and found sources (“Distance determination” on page 97). Each catalogue source is compared to a found source and the goodness of the potential match is ranked (“Ranking” on page 97). This process is repeated for every found source. A number of support tables are populated (“Matching” on page 99) to begin disentangling the question of which ones of all these matches are the best possible ones. E.g. if a certain found source has the highest rank as a match to a particular catalogue source and vice versa then the structure `perfectMatch` is used to note down this pairing. In the final step (“Best matches” on page 100) those structures are analyzed in order to pick out the best possible match that can be made to a particular catalogue source.

### Memory allocation

A large number of memory structures are allocated at the start of `q_identify_sources_match`
Figure 5.33: `Q_identify_sources_identify`: creating alerts
Figure 5.34: Q_identify_sources_identify: identifying missing sources
The function will use these to calculate the distance between each catalogue source and found source. Based on this information it will rank how well the found sources match the catalogue sources.

**Distance determination**

A pair of nested loops will run across all rows (found sources) and every column (catalogue sources) in each row (figure 5.36 on page 98). Coordinates for each source is retrieved from input data (catRA, catDec, foundRA, foundDec) and converted to radians if necessary. Slalib is invoked to find the angular distance on a sphere between the sets of coordinates (D). The function then calculates the relative distance (section 3.5.1 on page 28). The relative distance is stored in the distMatrix.

**Ranking**

The next processing step uses the distance matrix from the previous step to calculate “ranks”. Ranks express the order (from nearest to furthest) in which each found source is compared to each catalogue source and vice versa. The found source nearest the examined catalogue source would get a rank of 1 and sources further away get higher ranks. Two input parameters affects these calculations. maxQ acts as a cutoff distance. Any source pairing with $D_{rel} > maxQ$ get a default rank of 0. Additionally a “fuzziness” parameter ($D_{fuzz}$) is introduced that allows the user to express the degree of certainty desired for a source matching (“Ranking” on page 29).

To begin with the catRank matrix is operated on. Each entry was initialized to the number of catalogue sources in the previous step. (figure 5.36 on page 98). Each row of the distance matrix is analyzed in turn. I’ve created a simplified example row from the distance matrix together
I will use the following parameter values: \( \text{maxQ} = 14.0, \ D_{\text{fuz}} = 0.1 \). Each value on the row is, in turn, compared to the other values modified by \( D_{\text{fuz}} \) (equation 5.2 and table 5.2 on page 98).

Each time when the criterion (equation 5.3) is satisfied the rank value for that source is decreased by one, as long as it is higher than one previously. The first iteration would find that \( D_{\text{rel}}[1] = 5.1 \) is smaller than all \( D_{\text{test}}[m : 2 \rightarrow 5] \) and would decrease the corresponding rank four times by one. We can see the result, for the example row in \( \text{catRank} \), after each iteration step (table 5.3 on page 99).

\[
D_{\text{test}} = (1 + D_{\text{fuz}}) \times D_{\text{rel}}
\]  \hspace{1cm} (5.2)

\[
D_{\text{rel}}[n] < D_{\text{test}}[m], \ m \neq n
\]  \hspace{1cm} (5.3)
Table 5.3: Example of the ranking iteration.

<table>
<thead>
<tr>
<th>catRank/foundRank</th>
<th>CS 1</th>
<th>CS 2</th>
<th>CS 3</th>
<th>...</th>
<th>CS maxC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS 1</td>
<td>R[1,1]</td>
<td>R[2,1]</td>
<td>R[3,1]</td>
<td>...</td>
<td>R[maxC,1]</td>
</tr>
<tr>
<td>FS 2</td>
<td>R[1,2]</td>
<td>R[2,2]</td>
<td>R[3,2]</td>
<td>...</td>
<td>R[maxC,2]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>FS maxF</td>
<td>R[1,maxF]</td>
<td>R[2,maxF]</td>
<td>R[3,maxF]</td>
<td>...</td>
<td>R[maxC,maxF]</td>
</tr>
</tbody>
</table>

Table 5.4: The structure of the catalogue source / found source Rank Matrix. CS - catalogue source, FS - found source, maxC - total number of catalogue sources, maxF - total number of found sources, R - Ranking.

The nearest source has the rank 1 and the second nearest source a rank of 2 once the ranking algorithm has finished. The distances to the two sources ranked 3 are too similar to be discriminated between using this particular value for $D_{fuz}$ and therefore share the same rank. The source furthest away exceeds the distance cutoff, maxQ, and gets a rank of 0.

Figure 5.37 on page 100 displays the logic structure of the ranking code. The next step is to modify the foundRank matrix in the same way as the catRank matrix was modified. Columns are analyzed rather than rows for this iteration. Thus the differences compared to Figure 5.37 are that the outermost loop is run over the number of catalogue sources, the middle loop is run over the number of found sources, the innermost loop is run over the number of found sources and operations are carried out on foundRank instead of catRank. This will give us, for each catalogue source, the distance ranking for every found source (table 5.4 on page 99).

**Matching**

The next step is to use the distance rankings in order to find the best possible found source - catalogue source matches. Six lists are initialized as we loop over the found sources (figure 5.38 on page 101). Each list has the same basic structure apart from matchDist (table 5.5 on page 99). These are used to store matches of increasing complexity. In the highlight of the logics (figure 5.39 on page 102) we can trace out how this is done. If we have a pair of found source and catalogue source [F, C] then a “perfect” match is when C is ranked as closest to F and F is also ranked

<table>
<thead>
<tr>
<th>xxxMatch</th>
<th>FS 1</th>
<th>FS 2</th>
<th>FS 3</th>
<th>...</th>
<th>FS maxF</th>
</tr>
</thead>
</table>
| Catalogue source number | [i]? | [i]? | [k]? | ... | [l]?

Table 5.5: The structure of the various Matching lists. FS - Found source, maxF - total number of found sources. Each list element may either contain the number referring to a catalogue source that matches a certain found source or a code indicating that no match has been made.
closest to C. If C is ranked as closest to F but F is only second closest to C then we have a case of source confusion. In the reverse case, where F is the closest match to C but C is the second closest match to F, multiple catalogue sources are being matched to the same found source. This is carried one step further in the third set of logic checks checking for first vs. third matches. The best match counts, the second and third sets of checks discount catalogue sources that have already been matched.

**Best matches**

A memory structure containing the best possible pairing between found sources and catalogue sources, as well as the quality of the match, is filled out in the final step of the match function. The list with match pairings (Please see Table 5.6) should now display, for each found source, the matching catalogue source number. Empty locations indicates that no match with a catalogue source was possible to make.

The other list (Please see Table 5.7) will display the quality of the match that has been made between the catalogue source and the corresponding found source (Good, Source Confusion, etc.). Empty locations (No ID) would indicate that no match with a found source has been made.
Figure 5.38: $Q_{\text{identify sources match}}$: matching, overview

<table>
<thead>
<tr>
<th>matchListC</th>
<th>CS 1</th>
<th>CS 2</th>
<th>CS 3</th>
<th>...</th>
<th>CS maxC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching Quality</td>
<td>Code</td>
<td>Code</td>
<td>Code</td>
<td>...</td>
<td>Code</td>
</tr>
</tbody>
</table>

Table 5.7: The structure of Match List (catalogue sources). CS - Catalogue source, maxC - total number of catalogue sources.

<table>
<thead>
<tr>
<th>catMatch</th>
<th>CS 1</th>
<th>CS 2</th>
<th>CS 3</th>
<th>...</th>
<th>CS maxC</th>
</tr>
</thead>
</table>
| Found source number | FS [i]? | FS [j]? | FS [k]? | ... | FS [l]?

Table 5.8: The structure of the Catalogue Match list. FS - Found source, CS - Catalogue source, maxC - total number of catalogue sources.
Figure 5.39: Q_identify_sources_match: matching, logics†

† The image contains six logic checks (L1 through L6): catRank[C,F] is notated as C_{c,f} and foundRank[C,F] as F_{c,f};

- **L1**: C_{c,f} = F_{c,f} = 1
- **L2**: C_{c,f} = 1 ∧ F_{c,f} = 2
- **L3**: C_{c,f} = 2 ∧ F_{c,f} = 1
- **L4**: perfectMatch[F] = confuseMatch[F] = multiMatch[F] = NO_ID
- **L5**: C_{c,f} = 1 ∧ F_{c,f} = 3
- **L6**: C_{c,f} = 3 ∧ F_{c,f} = 1

![Diagram of matching logic checks](image-url)
“Perfect” matches Each matching list created in the preceding execution step will be analyzed in turn, from best to worst match types, to locate the best possible match for a catalogue source with a found source. The “perfect” matches\footnote{There is a one to one fit between a catalogue source and a found source.} are looked through initially. If a perfect match exists, and no previous match has been made (figure 5.40 on page 103), a check is carried out to see if it is the only “perfect” match for that source (figure 5.41 on page 104). The relevant tables and a counter are then updated to indicate a match has been made and all possible lower quality matches for the same found source are wiped.

Some additional steps need to be taken in a situation where more than one found source fits the same catalogue source equally well (figure 5.42 on page 104). The other found source gets matched to the same catalogue source in \texttt{matchListF} and its distance to that source is recorded. The corresponding catalogue source has its matching quality flag modified to indicate that a source confusion situation has arisen. All possible matches of lesser quality gets wiped.

“Confused” matches It is time to deal with matches of lesser quality once all perfect matches have been looked at. Next up are confused matches\footnote{More than one catalogue source has been matched to the same found source.} (figure 5.43 on page 105). The execution is nearly identical to the one for perfect matches (figure 5.44 on page 106). If a confused match is encountered then the remaining found sources are looped over to locate possible additional matches of the same quality. The last such multiple match is stored as \texttt{shareMatch}. The match lists are updated together with the match distance and all matches of lower quality are wiped.
Figure 5.41: $Q_{\text{identify sources match}}$: perfect matches, detail one

Figure 5.42: $Q_{\text{identify sources match}}$: perfect matches, detail two
If there were multiple matches then the match lists and match distance are updated to indicate this. Again, all matches of lower quality are wiped (figure 5.45 on page 106).

“Multiple” matches The third round of matching proceeds like the previous two and checks for multiple matches\(^\text{13}\) (figure 5.46 on page 107). The remaining found sources are scanned to look for more matches of the same quality if a multiple match is found. Multiples are stored and relevant tables are updated with matching cross references between the catalogue sources and found sources. Possible multiple matches are stored and all matches of lesser quality are wiped (figure 5.47 on page 108).

“More confused” and “more multiple” -matches The fourth and fifth round of matching is expected to produce matches very infrequently. The lists containing more confused matches\(^\text{14}\) and more multiple matches\(^\text{15}\) are consulted. No check for multiple matches is done due to the rarity of these situations. If a match is found then the catalogue source ID is written to matchListF and the correct matching quality is written to matchListC (like all the previous cases). If we have a more confused match then the corresponding entry for more multiple

---

\(^\text{13}\) More than one found source has been matched to the same catalogue source.

\(^\text{14}\) More than one catalogue source has been matched to the same found source and the best rank for the catalogue sources are 3.

\(^\text{15}\) More than one found source has been matched to the same catalogue source and the best rank for the found sources are 3.
Figure 5.44: $Q_{\text{identify sources match}}$: confused matches, detail one

Figure 5.45: $Q_{\text{identify sources match}}$: confused matches, detail two
Figure 5.46: \textit{Q\textunderscore identify\textunderscore sources\textunderscore match}: multiple matches, overview\textsuperscript{†}

\textsuperscript{†} L1: Multiple match exists AND no previous match with a catalogue source has been made
matches is wiped. In the reversed case there is no further list of lower quality matches left to wipe. The `catMatch` list is then updated with the ID of the found source written to the location corresponding to the matching catalogue source. And finally the `matchDist` list is updated with the distance between the two matched sources taken from the `distMatrix` array (figure 5.48 on page 109).

**Function exit** Just two tasks remains after the matching has been completed; the total number of matches made is written to the pointer the function was handed by the calling function for this purpose and all memory structures allocated by the function are wiped as the function exits.
Figure 5.48: \texttt{Q\_identify\_sources\_match}: final matches\textsuperscript{†} & function exit

\textsuperscript{†} The image contains two logic checks (L1 and L2):
- \textbf{L1}: More confused match exists AND no previous match with a catalogue source has been made
- \textbf{L2}: More multiple match exists AND no previous match with a catalogue source has been made
Appendix VI

\textit{Q\_match\_src\_fluxes}

The executable-specific code is contained in two functions. Execution begins with \texttt{q\_match\_src\_fluxes\_match}. The \texttt{q\_match\_src\_fluxes\_results} function is where the core tasks of \texttt{q\_match\_src\_fluxes\_match} are carried out.

The step by step execution is as follows; The “main logics” make a decision if the current found source should be analyzed or not. The “source matching” determines if we have a catalogue source that matches the current found source. A comparison of found source and catalogue source fluxes takes place in “flux matching” which also raises alerts in case of deviations. The hardness ratio is then determined with the opportunity for additional alerts in case of deviations. The found source is finally checked against a list of TOO sources. A match produces an alert. Each of these steps are described below.
Match function

The \texttt{q\_match\_src\_fluxes\_match} function mainly serves as a wrapper for \texttt{q\_match\_src\_fluxes\_results}. It will read catalogue source and found source data from the files and then hand over execution (figure 5.50 on page 112).

Results function overview

In the third picture (figure 5.51 on page 113) we have an overview of the entire \texttt{q\_match\_src\_fluxes\_results} function. Different sections are only shown as a general flow to maintain readability. The individual sections are described later.

Two values used for alert generation and data array creation are read. The array is filled with TOO source data if the executable has been asked to check for these (“New sources and Targets of Opportunity” on page 116). A logic check is performed on each found source. The source is compared against the catalogue sources if the check is passed (“Main logics” on page 112). The function then checks if the catalogue source and found source match each other (“Matching sources” on page 114). The total flux levels for matching sources are compared (“Comparing Fluxes” on page 114). An alert is issued if the found source flux deviates from the catalogue source flux. Additional analysis of the hardness ratio of the fluxes might be performed (“Calculating the hardness ratio” on page 115). There are checks and opportunities for creating more alerts once all catalogue sources have been looped over. Possibilities includes alerting for new source detections. The additional TOO source check is carried out if mandated. The function ends once all found sources have been checked.

Main logics

Several facts about the found source need to be checked to see if a source flux analysis should be performed. Multiple values are calculated (figure 5.52 on page 114). The significance of the detection, as determined by the detector software, is read. The number of science windows and observation groups is calculated. These numbers are variable for ISGRI but are set to 1 by default for the other instruments. The image middle point coordinates are read from parameters and the angular distance between the source and the center is calculated\textsuperscript{16}.

\textsuperscript{16} Using the \texttt{slaDsep} function detailed previously (section 3.2 on page 20).
Figure 5.51: Function overview
Three logic checks follow; Is the detection significance lower than a lower limit (from parameters)? Is the flag for detection significance set to zero? If either one these criteria is false, the found source is analyzed. Otherwise it is skipped. Only sources closer to the center than a distance limit (from parameters) will be analyzed. And either the number of science windows or observation groups must be above zero to proceed with analysis. The number of flux bins where the $E_{\text{min}}$ value exceeds zero and $E_{\text{max}}$ is larger than $E_{\text{min}}$ is counted. Execution continues by looping over catalogue sources if the checks are passed.

### Matching sources

This function matches found sources to catalogue sources either by comparing source ID or by attempting to match them based on angular distance (figure 5.53 on page 115). The second is an overlap with the functionality of *q_identify_sources* and is no longer in use. The distance between each possible matching of found source and catalogue source is calculated regardless. Input parameters determine if matching is based on distance or source ID comparison. The sources will be considered the same if their ID’s match or if they are near enough to each other. The function either proceeds to detect alert triggers (a match was made) or skips to the next catalogue source (no match could be made).

### Comparing Fluxes

The flux of the found source is compared against the flux of the matched catalogue source (figure 5.54 on page 115). The observed flux and its estimated error is read from the input file. The flux of the catalogue source is read from file, if available. The x-ray emission model of the source is used to estimate the flux if not. The estimation error is zero in the latter case. A call
is made to \texttt{q\_match\_src\_fluxes\_model}\textsuperscript{17} if the flux is estimated using a model.

Two values are calculated; the difference between the observed flux and its error as well as the sum of the catalogue flux and its error. An alert will be given if the first value is larger than the second value multiplied by a constant (read from parameters). The exact level of the alert is determined by how much the first value exceeds the second.

### Calculating the hardness ratio

A flux hardness ratio calculation is carried out for the found source (figure 5.55 on page 116). The flux rates in the hard and soft bins for the found source are read from input data. A call is made to \texttt{q\_match\_src\_fluxes\_model} if the source is flagged to use an x-ray emission model to determine the flux of the catalogue source. Otherwise the appropriate flux levels are read from the catalogue. The hardness ratio for both found source and catalogue source is calculated (equation 5.4).

$$\text{Ratio} = \frac{\text{Hard} - \text{Soft}}{\text{Hard} + \text{Soft}} \quad (5.4)$$

\textsuperscript{17} This model-function is identical to the model-function used by \texttt{q\_set\_src\_fluxes} and is described in Appendix IV.
The absolute value of the difference in hardness ratio is then compared against four limit values that are read from the parameters. Each limit value corresponds to an alert level. Alert texts are prepared and an alert is raised if the difference in hardness ratio is significant enough.

**New sources and Targets of Opportunity**

Two checks are carried out before the function raises an alert for a new source (figure 5.56 on page 116). The first check determines if the source exists in the catalogue. If it does not then the parameter file is consulted. The second check determines if the minimum alert level for new sources is between one and three. An alert is constructed if the source does not exist in the catalogue and the minimum alert level is appropriate. The detection significance is compared against a parameter value to determine the alert level. The alert texts are then constructed with all relevant data and an alert is raised. The spatial distance between the found source and every TOO source is calculated. An alert is created if any one such a distance is small enough.
Glossary

Alert A logging type supported by the Reporting Interface Library (RIL) that aims to get the attention of the human users of the system. The method of doing so depends on the severity of the alert, ranging from writing an entry in a logfile to prompts for immediate attention. Alerts are used in many circumstances but a typical example would be the detection of a new source by INTEGRAL. 8, 17, 28, 30–32, 34–39, 43–46, 48, 50–53, 61, 92, 94, 95, 111, 112, 114–116

Angular distance The great-circle or orthodromic distance between two points on a unit sphere. If the location of the points are expressed using the unit vectors $\vec{a}$ and $\vec{b}$ then the angular distance is $\Delta \sigma = \arctan \left( \frac{|\vec{a} \times \vec{b}|}{\vec{a} \cdot \vec{b}} \right)$. 23, 28, 31, 32, 41, 46, 73, 77, 97, 112, 114

Batch sequential architectural style A software architectural style that consists of connectors and stages, analogous to the filters and pipes in a chemical plant. The connectors transport data as a flow between the stages and the stages enrich, refine or transforms the data. 9–12

Catalogue A data structure employed by ISDC that contains a list of celestial sources and their relevant characteristics. Catalogues are stored as FITS-files. 18, 20, 23, 24, 27, 30–32, 39, 41, 43, 44, 61, 78, 81, 83, 91, 92, 101, 115, 116

Catalogue source Any source that has been previously detected and included in the INTEGRAL master catalogue. These could have been observed by other missions, especially at the beginning of INTEGRALs own mission. Over time it is expected that these will contain mainly sources that INTEGRAL has already observed. 8, 23, 24, 26–39, 41, 44, 46, 48, 50–52, 54, 61, 77, 78, 83, 91, 92, 94, 97–101, 103, 105, 107–109, 111, 112, 114, 115

Component A generic term for elements of software within the ISDC system. 9, 11, 12, 16, 17, 19, 32–35, 37, 38, 61, 62, 78

Connector A software architectural term, related to the batch sequential architectural style. The connector connects two stages, passing output data from the preceding stage to the next stage. Typically in this style, the connector is merely an abstraction and data is passed as a file on disc. 10–12, 14, 20

Data element This can either be a data object or a base element. 14, 15, 18

Data object A composite data type similar to a STRUCT in the C-language. A data object consists of data elements. 14, 15, 18
Executable  A computer program whose execution is completely defined by the input data and parameters it receives. 11, 12, 14, 15, 17, 21–24, 26, 27, 30, 67, 68, 70, 77, 91, 94, 111, 112

Found source A shorthand notation indicating any source that has recently been observed by INTEGRAL and is being analyzed by the scientific analysis software. A found source may, or may not, also be a catalogue source but has not yet been identified as being one. 8, 23, 24, 27–32, 34–39, 41, 44, 46, 48, 50–54, 61, 77, 78, 81, 91, 92, 94, 97, 99–101, 103, 105, 108, 111, 112, 114–116

Library A library of functions or subroutines to be called by executables or applications. 14–16, 18, 23, 70, 77

Mask One-dimensional truth-value arrays with a length corresponding to the size of a catalogue or source list. Masks can be combined using any logical operation and are used to select sources from catalogues or source lists. 18, 23, 24, 77, 78

Observation A set of logically connected pointings by INTEGRAL. 13, 14, 19, 20

Observation Group The data structure that contains an observation. 20, 23, 31, 61, 70, 112, 114

Parameter An input value an executable requires from the system in order to perform its function. This could be, for instance, the location of the data files the executable operates on. 4, 11, 12, 14, 16–18, 23, 24, 28–33, 36–39, 43, 45, 46, 48, 51–56, 60, 68, 70, 73, 76, 78, 83, 87, 94, 97, 98, 112, 114–116

Parameter file The files belonging to an executable that define the inputs needed from the system in order for the executable to perform its function. 11, 15–17, 38, 116

Pipeline A set of components, typically scripts, chained together via the OPUS system. Pipelines control the usage of system resources and the overall data flow. 12, 19, 20, 37, 61, 68

Pointing The direction of the axis of the spacecraft along which the instruments are aligned. The term is also used to indicate a single exposure by INTEGRAL, usually 2000s long, when the spacecraft is held pointed in a single direction. 13, 30, 31, 54, 59, 76, 91, 92

Pointing error The difference between the true pointing of the spacecraft and our knowledge of that pointing. 30, 38, 39, 53, 54, 59–61, 91, 92

Procedural language Procedural programming languages are derived from the concept of the procedure call. A procedure is a subsection of code that performs some developer specified task, preferably quite tight in scope. Other names for procedures are; routines, subroutines, methods or functions. 9, 10

Relative distance Related to the angular distance. This can either be the angular distance in units of the combined positional uncertainty of the two points or the angular distance directly, depending on circumstances and user choice. 28, 29, 38, 39, 41, 48, 50, 51, 62, 94, 97
Run A pair of source matchings between two sets of catalogue sources \( (C \text{ and } C') \) and found sources \( (F \text{ and } F') \) denominated as \( C \equiv F \) and \( C' \equiv F' \). \( C \) and \( C' \) are otherwise identical to each other apart from one source having been removed at random from \( C' \). \( F \) and \( F' \) are identical to each other. For \( C \equiv F \) the interesting measurements are; no alerts were issued vs other results. For \( C' \equiv F' \) the interesting measurements are; a new source alert was issued vs other results. 36, 39, 43, 48

Science Window Data gathered during a single pointing or slew. 13, 19, 20, 23, 31, 61, 70, 73, 76, 112, 114

Script A set of executables, or lower level scripts, tied together with flow control instructions in a scripting language. 12, 20–22, 68

Shadowgram A pattern of light and shadow cast by a source onto the detector of a telescope once it has passed through a coded aperture mask. The logical mask referred to elsewhere in the glossary is not the same as this physical mask. 13, 20

Source list A specialized form of a catalogue. A source list is typically a subset of a catalogue and may contain some additional data for the sources that is useful for analysis. 18, 23, 31–33, 38, 41

Source position error The difference between the true position of a source and the detector’s measurement of that position. 23, 28, 39, 41–44, 46, 50, 56, 60, 62, 78

Stage A software architectural term, related to the batch sequential architectural style. It describes a self-contained set of code that reads input data, enriches/refines/transforms the data and writes the results to its output after finishing all processing. 10–12, 14, 20

Status Merriam-Webster: “state or condition with respect to circumstances <the status of the negotiations>”. The status of a function is either a pre-defined value (ISDC.\_OK = 0) or a negative value in the ISDC context. ISDC.\_OK indicates that no errors have been encountered. A negative value indicates that a previously defined error situation has occurred. 16–18, 68, 70

Support library ANSI C/C++ and Fortran 90 callable software libraries containing functions that acts as a layer between the user and the physical hardware & exact software environment used by ISDC. 9, 68

Test A large number of runs makes up a test. A test will result in two measures; \( \Lambda \) and \( \Lambda' \). \( \Lambda \) is the number of runs producing no alerts divided by the total number of runs. \( \Lambda' \) is the number of runs producing at least one new source alert divided by the total number of runs. 4, 33, 36–38, 41, 43–46, 48, 51–53

Transient Merriam-Webster: “passing especially quickly into and out of existence”. Refers in this text to non-repeating, rapidly evolving, astrophysical processes like supernovae and gamma-ray bursts. 8, 19, 32, 61

119