Fast distributed cross-matching of big astronomical data and parameter estimation of moving point source model

Zečević, Petar

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Petar Zečević

FAST DISTRIBUTED CROSS-MATCHING OF BIG ASTRONOMICAL DATA AND PARAMETER ESTIMATION OF MOVING POINT SOURCE MODEL

DOCTORAL THESIS

Zagreb, 2023

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Petar Zečević

FAST DISTRIBUTED CROSS-MATCHING OF BIG ASTRONOMICAL DATA AND PARAMETER ESTIMATION OF MOVING POINT SOURCE MODEL

DOCTORAL THESIS

Supervisors: Academician Sven Lončarić, F.C.A.;

Professor Mario Jurić, PhD

Zagreb, 2023

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Petar Zečević

BRZO RASPODIJELJENO UPARIVANJE VELIKIH ASTRONOMSKIH PODATAKA I PROCJENA PARAMETARA MODELA GIBAJUĆEGA TOČKASTOG IZVORA

DOKTORSKI RAD

Mentori: akademik Sven Lončarić, prof. dr. sc. Mario Jurić

Doctoral thesis was written at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Electronic Systems and Information Processing.

Supervisors: Academician Sven Lončarić, F.C.A.; Professor Mario Jurić, PhD

Thesis contains 101 pages

Thesis no.:

ABOUT THE SUPERVISORS

Academician Sven Lončarić, F.C.A. received Diploma of Engineering and Master of Science degrees in electrical engineering from the Faculty of Electrical Engineering and Computing in 1985 and 1989, respectively. He received Ph.D. degree in electrical engineering from University of Cincinnati, USA, in 1994. Since 2011, he has been a tenured full professor in electrical engineering and computer science at FER. He was a project leader on a number of research projects in the area of image processing and computer vision. From 2001-2003, he was an assistant professor at New Jersey Institute of Technology, USA. He founded the Image Processing Laboratory at FER and the Center for Computer Vision at University of Zagreb. Prof. Lončarić has been a co-director of the national Center of Research Excellence in Data Science and Cooperative Systems and the director of the Center for Artificial Intelligence at FER. With his students and collaborators he published more than 250 scientific papers. Prof. Lončarić is a full member of the Croatian Academy of Sciences and Arts. According to a Stanford University study published in 2022 he was ranked in the top 2% of the most cited world scientists in the category artificial intelligence – image processing. For his scientific work he received several awards including the National Science Award.

Professor Mario Jurić, PhD received Diploma of Bachelor of Science in Physics from University of Zagreb in 2002. He received Ph.D. degree in Astrophysical Sciences from Princeton University, USA, in 2006 with the thesis "Long Term Evolution and Stability of Planetary Systems". He was a Postdoctoral Member at the Institute for Advanced Study from 2006 until 2009. He was a Hubble Fellow at Harvard University from 2009 to 2011. From 2012 to 2014 he was Associate Astronomer at Steward Observatory, University of Arizona and LSST Data Management Subsystem Scientist, LSST from 2012 until 2017. He has been serving as a Senior Data Science Fellow at eScience Institute, University of Washington (UW) since 2014. He is also an Associate Professor of Astronomy at UW since 2016, member of Founding Faculty of DIRAC Institute at UW since 2017 and Director of DIRAC Institute at UW since 2020. His interests include computational and data-intensive problems in planetary science and astrophysics; large astronomical surveys; small bodies of the Solar System; galactic structure, formation, and evolution; data science methodology; and dynamics. He has co-authored 69 peer-reviewed articles, 23 conference proceedings, 69 abstracts, 10 preprints, and 9 other products.

O MENTORIMA

Akademik Sven Lončarić diplomirao je i magistrirao u polju elektrotehnike na Fakultetu elektrotehnike i računarstva, 1985. i 1989. godine. Doktorirao je u polju elektrotehnike na Sveučilištu u Cincinnatiju, SAD, 1994. godine. U zvanje redoviti profesor u trajnom zvanju u polju elektrotehnike i polju računarstva na FER-u izabran je 2011. godine. Bio je suradnik ili voditelj na brojnim istraživačkim i razvojnim projektima u području razvoja metoda za obradu slika i računalnog vida. Od 2001. do 2003. bio je Assistant Professor na Sveučilištu New Jersey Institute of Technology, SAD. Voditelj je istraživačkog laboratorija za obradu slike na FER-u. Osnivač je i voditelj Centra izvrsnosti za računalni vid na Sveučilištu u Zagrebu. Suvoditelj je nacionalnog Znanstvenog centra izvrsnosti za znanost o podatcima i kooperativne sustave i voditelj Centra za umjetnu inteligenciju FER-a. Sa svojim studentima i suradnicima publicirao je više od 250 znanstvenih i stručnih radova. Prof. Lončarić redoviti je član Hrvatske akademije znanosti i umjetnosti. Prema studiji Sveučilišta Stanford objavljenoj 2022. godine rangiran je u 2% najutjecajnijih svjetskih znanstvenika u kategoriji umjetna inteligencija i obrada slike. Za svoj znanstveni i stručni rad dobio je više nagrada uključujući Državnu nagradu za znanost.

Prof. dr. sc. Mario Jurić diplomirao je u polju fizike na Zagrebačkom sveučilištu 2002. godine. Doktorirao je u polju astrofizike Sveučilištu Princeton, SAD, 2006. godine s temom "Long Term Evolution and Stability of Planetary Systems". Od 2006. do 2009. godine bio je postdoktorand (Postdoctoral Member) u Institute for Advanced Study. Od 2009. do 2011. godine bio je "Hubble Fellow" na Sveučilištu Harvard. Od 2012. do 2014. godine bio je Associate Astronomer na Steward opservatoriju, na Sveučilištu Arizona te LSST Data Management Subsystem Scientist od 2012. do 2017. godine. Od 2014. godine služi kao Senior Data Science Fellow pri eScience Institutu, na Sveučilištu Washington. Također je izvanredni professor astronomije na Sveučilištu Washington od 2016. godine, član Founding Faculty DIRAC instituta na Sveučilištu Washington od 2017. godine i direktor DIRAC instituta na Sveučilištu Washington od 2020. godine. Među njegovim interesima su računalno i podatkovno intenzivni problemi u planetarnoj znanosti i astrofizici, veliki astronomski pregledi, malena tijela solarnog sustava, galaktička struktura, nastanak i evolucija te metodologija podatkovne znanosti. Koautor je na 69 recenziranih članaka, 23 konferencijska članka, 69 sažetaka i 19 drugih stručnih radova.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisors: Prof. Sven Lončarić who allowed me to dedicate time for work on the PhD by offering me a job at the University and who lead me with his advices until the very end; and Prof. Mario Jurić for his dedication and time he's selflessly spent discussing with me many things regarding problems in Astronomy and Astrostatistics. He's saved me many times from wandering away on the wrong tracks. I also owe my gratitude to Prof. Željko Ivezić from University of Washington for his patient counseling and reviewing of my manuscripts.

My deep gratitude goes to my wife Ksenija, mother Vera, father Ante and sister Ivana without whose love and support this thesis wouldn't have seen the light of day.
I dedicate this work and my whole life to my heavenly mother Mary.

Zagreb, 13th June 2023.

ABSTRACT

Fast distributed cross-matching of big astronomical data and parameter estimation of moving point source model

The thesis develops two original methods: "Distributed Zones Algorithm" and "Online Multifit". "Distributed Zones Algorithm" is a zone-based method for fast distributed cross-matching of big astronomical data. It is developed based on the Zones Algorithm [1] and comprises a distributed data organization scheme and an efficient method of positionally joining data using a moving window. The algorithm is implemented within a system called AXS (Astronomical Extensions for Spark), based on Apache Spark and extended with astronomy-specific functionalities. The system's cross-matching performance has been extensively tested in single-machine and "cloud" environments.

"Online Multifit" is a method for parameter estimation of a moving point-source model based on sequential Bayesian updating. It was developed by extending the Multifit method. Online Multifit utilizes Bayesian statistics to obtain the full posterior probability distribution of moving point source model parameters but it does so by processing one image at a time using "sequential updating" technique, updating the posteriors after each image. The method approximates posteriors as mixtures of Gaussians and the main challenge here is to reduce the errors introduced by these approximations. The final result is that the method uses less computational resources while obtaining full posterior estimates. Furthermore, the estimates can be stored in a compressed form (the means and covariances of Gaussian distributions).

KEYWORDS: astronomy, cross-matching, Apache Spark, Bayesian statistics, sequential updating

SAŽETAK

Brzo raspodijeljeno uparivanje velikih astronomskih podataka i procjena parametara modela gibajućega točkastog izvora

Disertacijom se razvijaju dvije originalne metode: "algoritam baziran na distribuiranim zonama" i "slijedni Multifit".

"Algoritam baziran na distribuiranim zonama" ("ABDZ metoda" u nastavku) je metoda za brzo distribuirano uparivanje velikih astronomskih podataka. Astronomski pregledi su mape regija neba koje tipično nastaju u sklopu astronomskih projekata vezanih uz pojedine teleskope. Astronomski pregledi stvaraju astronomske kataloge, velike baze s različitim mjerenjima svojstava astronomskih objekata. Posljednjih desetljeća svjedočimo sve većem rastu astronomskih pregleda. Tako će, na primjer, Vera C. Rubin Observatory (projekt ranije poznat kao "LSST"), koji će s radom početi 2025. g., prikupiti više od 10 PB podataka o ukupno 20-ak bilijuna objekata.

Podatke iz astronomskih pregleda astronomi tipično analiziraju na način da pristupaju mrežnim sučeljima vezanim za pojedine preglede, dohvate podskup podataka te ga analiziraju na svojim lokalnim mašinama. Međutim, astronome često zanima obrada podataka cijelog neba, a ne samo nekih regija. I često žele uparivati i uspoređivati podatke iz više kataloga u isto vrijeme. Moderni sustavi za analizu i obradu astronomskih podataka bi stoga trebali biti skalabilni i jednostavni za korištenje, podržavati astronomske funkcije, koristiti industrijski standardne biblioteke te omogućiti brzo uparivanje kataloga u stvarnom vremenu. Sustav AXS, razvijen u sklopu ove disertacije, primjer je takvog sustava.

Prostorno uparivanje kataloga jest zapravo pronalaženje parova detekcija iz jednog i drugog kataloga koje su dovoljno prostorno blizu (ovisno o nekom parametru "epsilon"). ABDZ metoda je razvijena na temelju "algoritma baziranog na zonama" [1] koji organizira nebo u horizontalne zone koje služe kao indeksi u podskupe podataka kataloga. To smanjuje količinu podataka koje je potrebno pretraživati kako bi se pronašli parovi. Osnovna ideja ABDZ metode jest organizirati podatke u distribuirane zone (u više datoteka), operaciju uparivanja izraziti kao "epsilon join" upit s funkcijom za izračun udaljenosti te izvršiti taj upit u paraleli povrh distribuiranih podataka.

ABDZ metoda podatke, osim po zonama, dodatno organizira u "kante" (eng. "buckets"), tj. fizičke datoteke, na način da se svi objekti iz pojedine zone spremaju u dediciranu kantu te da se odabir kante za zonu vrši slijedno. Na taj način se automatski smanjuje asimetrija (eng. "skew") u distribuciji podataka (ako su zone dovoljno uske), koja je tako često prisutna u astronomskim podacima i često uzrokuje probleme u obradi podataka jer neke datoteke

sadrže puno podataka, a druge vrlo malo. Kod organizacije podataka primijenjene u ABDZ metodi, podaci se ravnomjerno raspoređuju po "kantama" (datotekama).

Podatke unutar "kanti" je zatim potrebno sortirati po zoni i RA ("right ascention", tj. x) koordinati nakon čega je podatke moguće efikasno uparivati u parovima dvije po dvije "kante" u jednom prolazu kroz podatke na način da se za svaki redak na lijevoj strani na desnoj strani održava pomični prozor koji sadrži samo one retke unutar određene udaljenosti na temelju RA kolone. Samo za te parove redaka je onda potrebno izračunati vrijednost funkcije udaljenosti čime se troši minimalna količina memorije i računalnih resursa.

ABDZ metoda implementirana je unutar sustava AXS (Astronomical Extensions for Spark), temeljenog na Apache Spark sustavu, koji je standardan alat u industriji za obradu velikih količina podataka. Sparkov Python API u AXS-u je nadograđen s funkcijama specifičnim za astronomiju te je njegova "sort-merge join" implementacija nadograđena na način da je u stanju efikasno izvršavati ranije spomenute "epsilon join" upite.

Performanse AXS sustava u uparivanju astronomskih kataloga mjerene su nad Gaia (430 GB), AllWISE (352 GB), ZTF (1,1 TB) i SDSS (66 GB) katalozima na jednoj velikoj mašini. Pri korištenju 28 paralelnih procesa AXS je uparío Gaia i ZTF kataloge za 334 sekunde s praznom predmemorijom ("cold cache") te za 41 sekundu s parcijalno popunjenom predmemorijom ("warm cache"), što je znatno brže od poznatih alternativnih pristupa.

Testirano je i trajanje particioniranja i pripreme podataka u ovisnosti o broju korištenih "kanti" i broju zona. Niti jedno niti drugo nije znatno utjecalo na trajanje pripreme podataka, dok je svako udvostručenje broja zona povećalo veličinu kataloga za otprilike 10%.

Skalabilnost AXS sustava testirana je u okolini "u oblaku", u Kubernetes clusteru na Amazon infrastrukturi. Testovi su pokazali da je sustav linearno skalabilan ako se proporcionalno povećavaju i količina podataka i broj procesa ("weak scaling").

"Slijedni Multifit" je metoda za estimaciju parametara modela gibajućeg točkastog izvora temeljena na slijednom Bayesovom osvježavanju. PSF ("point spread function") je funkcija koja određuje oblik kojeg točkasti izvor svjetla ("point source") proizvodi u fokalnoj ravnini teleskopa te ovisi o položaju izvora, vremenu, stanju instrumenta itd. Kod teleskopa smještenih na Zemlji, glavna komponenta PSF funkcije jest atmosfera koja titra što proizvodi dodatni šum kod astronomskih observacija. Tradicionalan način za smanjenje šuma, tj. povećanje omjera signala i šuma (eng. "S/N ratio"), jest zbrajanje slika (eng. "image coaddition") koje smanjuje šum s korjenom broja slika koje se koriste. Međutim, kod gibajućih izvora, zbrajanje slika proizvodi mutan trag gibanja zbog čega se tako zbrojene slike ne mogu koristiti za pouzdana mjerenja.

Multifit metoda [2][3] povećava omjer signala i šuma korištenjem informacija iz svih dostupnih slika bez zbrajanja slika. Međutim, ona koristi metode tradicionalne statistike, dakle proizvodi samo procjene najvjerojatnijih vrijednosti parametara te zahtijeva obradu svih dostupnih slika odjednom. Slijedni Multifit (eng. "Online Multifit") metoda, razvijena u sklopu ove disertacije, koristi metode Bayesove statistike i proizvodi procjene potpunih posteriornih distribucija vjerojatnosti parametara, te zahtijeva obradu samo posljednje slike u nizu.

Ovdje se za gibanje točkastog izvora svjetla pretpostavlja jednostavan linearan model gdje su brzine u x i y smjerovima konstantne. Pretpostavka je da je izvor svjetla detektiran prilikom automatske obrade u sklopu astronomskog pregleda (npr. "LSST data processing

pipelines") na nekoj poziciji x_E i y_E. No zbog mutnog traga gibanja, ta je inicijalna procjena nepouzdana pa algoritam dopušta korekciju te inicijalne procjene parametrima x_0 i y_0. Osim toga, algoritam nastoji procijeniti i "magnitudu" (mjera svjetline) izvora M počevši od inicijalno procijenjene magnitude M_E. Na početku postupka estimacije dostupne su i slike $I_1 \dots I_N$ dobivene u trenucima $t_1 \dots t_N$, slike s procijenjenim varijancama $V_1 \dots V_N$ i procijenjene PSF funkcije $1 \dots N$.

Šum na astronomskim slikama dobro se opisuje Poissonovom razdiobom jer se radi o diskretnim događajima (udarima fotona na područje nekog piksela). Kod većeg broja događaja Poissonova razdioba može se dobro aproksimirati Gaussovom razdiobom gdje su i sredina i varijanca jednake parametru λ, tj. pretpostavljenom signalu, tj. pretpostavljenom modelu stvaranja slike. Funkcija izglednosti je onda određena Gaussovom razdiobom pa je za njezinu maksimizaciju dovoljno minimizirati tzv. χ^2 statistiku koja se može izračunati oduzimajući modeliranu sliku od promatrane, dijeleći kvadrat dobivene razlike s varijancom, te zbrajajući sve tako dobivene piksele.

χ^2 statistika distriburana je po χ^2 distribuciji s k stupnjeva slobode (gdje je k jednak broju ulaznih uzoraka podataka, minus broj parametara koji se estimiraju). Kod većih stupnjeva slobode k, ta distribucija se također može dobro aproksimirati Gaussovom razdiobom sa srednjom vrijednošću jednakom k te varijancom od $2k$. Ako se ta razdioba zatim podijeli s k, dobije se χ^2 statistika "po stupnju slobode" (eng. "χ^2 per degrees of freedom", ili χ^2_{DoF}) sa središtem u 1 te varijancom $2/k$. χ^2_{DoF} statistika se na taj način može koristiti kao test točnosti estimacije.

Slijedni Multifit metoda razvijana je i testirana na simuliranim podacima jer podaci s dovoljnim brojem pouzdanih mjerenja gibanja nebeskih tijela nisu lako dostupni. Objekti su simulirani na "pravim" astronomskim slikama s teleskopa HSC SSP, konkrektno na RC2 podskupu slika s tog astronomskog pregleda. Odabrana je regija s dobrom pokrivenošću te podijeljena u 304 (16x19) sličica (eng. "cutouts") dimenzija 30x30 piksela te je gibajući izvor simuliran u središtu svake od sličica. Objekti su simulirani s nasumično odabranim brzinama u RA (eng. "right ascention", tj. x) i Dec (eng. "declination", tj. y) smjerovima na način da većina objekata prijeđe 1 piksel i maksimalno 6 piksela u promatranom vremenskom periodu. Magnitude simuliranih objekata su se povećavale od 18 (vrlo sjajno) do 27 (nevidljivo na individualnim slikama) s povećanjem Dec koordinate.

Važno je primijetiti distribuciju vremena kada su slike u odabranom skupu dobivene: prvog dana dobiveno je 9 ekspozicija, a idućih 8 tek nakon 240 dana, i to ponovno na isti dan. Ova situacija nije idealna za procjenu gibanja objekata jer njihovo gibanje nije primjetno unutar jednog dana. Bilo bi puno bolje za primjenu u ovoj disertaciji kada bi ekspozicije bile ravnomjerno vremenski raspodijeljene.

Za obradu slika i simulaciju objekata korišteni su standardni LSST programski "zadaci" (eng. tasks), razvijeni u sklopu Vera C. Rubin Observatory projekta, slijedno pozivani u skladu s preporukama autora: `singleFrame`, `insertImages`, `jointCal`, `makeWarp`, `assembleCoadd`, coadd measurement, forced photometry. Nakon završetka cijelog procesa, u LSST katalogu dostupna su mjerenja nad objektima detektiranim na slikama te ih je moguće dohvatiti korištenjem LSST komandi i upita. Simulirani objekti uspješno su pronađeni u katalogu, osim manjeg broja najmanje sjajnih (magnitude 25, 26 i 27) te objekata simuliranih u blizini vrlo sjajnih zvijezda. Ovi detektirani simulirani objekti su dalje korišteni u razvoju

i testiranju algoritama.

Budući da se za razvoj metoda u ovoj disertaciji koristi predviđajuće modeliranje (eng. "forward modeling"), prvi korak je definiranje modela generiranja slika na temelju pretpostavljenih parametara (pozicije, sjaja i brzine), te definiranje funkcije izglednosti koja će oduzimanjem modeliranih od stvarnih slika, te dijeljem rezultata s varijancom i zbrajanjem svih tako dobivenih piksela dobiti χ^2 vrijednost. Validnost funkcije izglednosti može se provjeriti vizualnom inspekcijom rezultata svakog od koraka, izračunom χ^2_{DoF} vrijednosti ili izračunom njezinih vrijednosti na rešetci parametara (eng. parameter grid), no to vrlo brzo postaje nemoguće s porastom broja parametara.

Multifit metoda je u ovoj disertaciji implementirana na tri načina:

- Kao grupna Multifit metoda korištenjem tradicionalne, frekventističke statistike, na temelju rada "Measuring the undetectable" [3].
- Kao grupna Multifit metoda korištenjem Bayesove statistike
- Kao slijedna Multifit metoda korištenjem Bayesove statistike

"Grupna Multifit metoda", za razliku od "slijedne", znači da se uvijek obrađuju sve dostupne slike, a slijedna Multifit metoda jedan je od doprinosa ove disertacije. Ona u različitim koracima koristi određene aproksimacije te je za očekivati da će njezina točnost biti manja od prve dvije implementacije. Slično je i s usporedbom Bayesove metode s frekventističkom te frekventistička može poslužiti kao osnovna implementacija kojoj se Bayesova treba približiti, a grupna Bayesova metoda kao ideal kojem se treba približiti slijedna Bayesova metoda.

Frekventistički pristup sastoji se u jednostavnoj maksimizaciji metode izglednosti, tj. minimizaciji χ^2 vrijednosti. Postoje različite optimizacijske metode koje mogu biti korištene u tu svrhu i čije su implementacije dostupne u obliku Python paketa. Među njima su Newtonova optimizacijska metoda, Gauss-Newtonov algoritam, Levenberg-Marquardtova metoda, Trust Region Reflective metoda, Dogbox, Powell, Nelder-Mead itd. Devet glavnih optimizacijskih algoritama je uspoređeno po preciznosti estimacije parametara te po brzini rada. Svi algoritmi su rezultirali pogreškom `LinAlgError` u određenom postotku. U tim bi slučajevima algoritam bio ponovno pokrenut s neznatno pomaknutom početnom pozicijom.

Powell metoda je odabrana kao općenito najbolja te je korištena za sva daljnja testiranja. Frekventistička implementacija implementirana je u obliku LSST "zadatka" te je spremna za korištenje u sklopu Vera C. Rubin projekta.

Druga implementacija jest također grupna implementacija, ali je bazirana na Bayesovoj statistici. Bayesova statistika temelji se na Bayesovoj formuli koja kombinira izglednost podataka s obzirom na model i parametre (već definirana funkcija izglednosti), priorne vjerojatnosti samih parametara (npr. ukoliko je poznato da brzine objekata nikada ne prelaze određene vrijednosti) te "dokaz", tj. normalizacijski integral vjerojatnosti podataka i parametara. Integral dokaza je najteži dio formule za izračunati. U rijetkim slučajevima to je moguće analitički, a u ostalima se koriste numeričke metode, najčešće Markov Chain Monte Carlo (MCMC), koja se koristi i u ovoj disertaciji.

Monte Carlo metode koriste slučajne uzorke za procjenu ne-slučajnih varijabli, a Markov Chain Monte Carlo koristi markovljeve lance slučajnih vrijednosti koji opisuju danu funkciju

vjerojatnosti na način da se najviše zadržavaju u najvjerojatnijim područjima. Postoje različite metode uzorkovanja koje je moguće koristiti unutar MCMC postupka. U ovoj disertaciji koristi se grupna (eng. ensemble) metoda čija efikasnost ne ovisi o afinim transformacijama (iskrivljenost distribucije) iz [4]. To je grupna metoda jer koristi set "hodača" (eng. walkers) koji određuju svoju iduću poziciju na temelju pozicija svih ostalih hodača. Autori predlažu razlčite "korake" koji se pri tome mogu koristiti. Tijekom implementacije, "DE korak" i "KDE korak" iz [5] pokazali su se najboljima za ovaj Mulitifit problem.

Rezultat MCMC procedure je histogram povijesnih vrijednosti lanaca (njihovih uzoraka), a pritom vrijedi da što je više uzoraka u nekom intervalu, vjerojatnost tog intervala je veća. Iz tih je histograma potrebno dobiti najvjerojatnije vrijednosti parametara, tj. njihove MAP (eng. maximum a-posteriori) vrijednosti. Također, cilj Bayesovih Multifit metoda jest i opisivanje posteriornih distribucija vjerojatnosti, a ne samo MAP vrijednosti. U tu svrhu potrebno je iz histograma dobiti matematički opis distribucije. Za to je moguće koristiti metodu histograma, no ona ima više nedostataka. Moguće je koristiti i metodu procjene gustoće jezgrama (eng. kernel density estimation), no ona se je pokazala presporom. U ovom radu posteriorne distribucije opisuju se Gausovim smjesama (eng. Gaussian mixtures). Za estimaciju Gaussove smjese iz histograma moguće je koristiti expectation-maximization ili variational inference algoritam. U ovom radu koristi se variational inference jer je u stanju smanjiti težine pojedinih komponenti Gaussove smjese blizu nuli te ih praktično eliminirati.

Histograme koje proizvodi MCMC procedura moguće je vizualizirati i pomoću tzv. "corner plot" grafova koji za svaki par parametara pokazuju konturne grafove te su korisni za istraživanje korelacija među parametrima. Još jedan koristan dijagnostički alat su Q-Q grafovi (eng. quantile-quantile plots) koji daju usporedbu dvije distribucije. Budući da se u ovom radu histogrami aproksimiraju Gaussovim distribucijama, korisno je histograme usporediti s Gaussovom distribucijom. Tu je vidiljivo da je za MCMC proceduru potrebno koristiti dovoljan broj iteracija i "hodača" jer inače oni ne istraže adekvatno posteriornu distribuciju vjerojatnosti već samo djelomično. U takvim slučajevima će i Gaussova aproksimacija unijeti veću pogrešku.

MAP procjene parametara moguće je također dobiti na nekoliko načina. Prvi način je iz moda (najviše točke) procijenjene Gaussove aproksimacije. Ta metoda, međutim, nema veliku točnost. Bolja (i najrobusnija) metoda je izračunom medijana iz histograma uzoraka. No, njezina točnost ovisi o kvaliteti eksploracije distribucije (broju iteracija i "hodača"). U ovom radu MAP procjene se dobivaju dodatnim izvršavanjem Powell (optimizacijskog) algoritma kome je medijan histograma postavljen za početnu vrijednost. Pokazalo se da točnost ove metode ne ovisi o broju iteracija i "hodača". Međutim, kvaliteta aproksimacije Gaussove smjese, koja je važna za slijednu metodu, ovisi o kvaliteti eksploracije distribucije pa se uvijek koristi 400 iteracija, kao kompromis.

Slijedna Multifit metoda temelji se na slijednom Bayesovom osvježavanju gdje se za svaki novi podatak (tj. sliku u ovom slučaju) ažurira posteriorna distribucija vjerojatnosti (na temelju funkcije izglednosti). Ta posteriorna distribucija služi kao prior za idući podatak (tj. sliku). Svaki korak je zasebna Bayes procedura gdje se izvršava MCMC, dobiva histogram te se histogram aproksimira Gaussovom smjesom. Nakon svakog koraka moguće je provesti iste dijagnostičke postupke: analizu corner plotova i Q-Q grafova, a osim toga i analizirati

kako se procijenjene MAP vrijednosti sa svakim korakom približavaju pravim vrijednostima.

Budući da se kod slijedne Multifit metode uvijek obrađuje samo jedna slika, a kod grupne metode sve dostupne slike, procesna kompleksnost slijedne metode je $O(N)$, a grupne je $O(N^2)$. Memorijska kompleksnost slijedne metode je $O(1)$, a grupne $O(N)$.

Sve tri implementacije uspoređene su na temelju slijedećih metrika:

- Sigma udaljenost - udaljenost procijenjenog rješenja od pravog rješenja u 5-D prostoru parametara u jedinicama standardne devijacije. Međutim, različite metode procjenjuju standardnu devijaciju na različite načine i uz velike razlike te se ova metrika nije pokazala previše korisnom.
- Prosječna udaljenost putanje - prosječna udaljenost procijenjenih od stvarnih pozicija objekta u simuliranim trenutcima
- χ^2_{DoF} vrijednost
- Sirova odstupanja vrijednosti parametara od stvarnih (simuliranih) vrijednosti
- Trajanje procedure

Rezultati su pokazali da je frekventistička implementacija na temelju [3] najtočnija i najbrža. Bayesian grupna metoda joj je vrlo blizu po točnosti sve do magnitude od 26 nakon koje se točnost znatno smanjuje. Slijedna Multifit metoda ima veću pogrešku (nakon obrade 15 slika), u prosjeku cca. 0.35 piksela, također do magnitude 26, što može biti prihvatljivo za određene primjene.

Kada se gleda točnost u ovisnosti o koraku (broju obrađenih slika), slijedna metoda pokazuje da se točnost ustrajno popravlja što dokazuje stabilnost cijelog postupka. Pri tome je vidljivo pogoršanje koje počinje oko 10-og koraka, a koje se može objasniti distribucijom vremena kada su slike dobivene: prvih 9 slika dobiveno je na isti dan. Iduće slike donose novu informaciju koja privremeno algoritam skreće na krivi trag, no nakon manjeg broja koraka, algoritam se oporavlja.

Sustav AXS se od 2018. godine koristi u Dirac institutu pri Sveučilištu Washington u razne znanstvene svrhe ([6] i [7]). Također ga je koristilo više istraživačkih grupa diljem svijeta. Trenutno se AXS razvija na Dirac institutu u iduću verziju radnog imena "HiPSCat" koja Parquet format zamjenjuje ekstenzijom IVOA HiPS standarda kojeg je moguće čitati direktno iz Python koda. Plan za ovu novu verziju AXS sustava je da se koristi kao sustav za obradu podataka na Vera C. Rubin projektu te na drugim NASA projektima.

Frekventistička implementacija Multifit algoritma je zapakirana u obliku LSST "zadatka" te ju je moguće koristiti u sklopu Vera C. Rubin projekta. Kako informacije s tog projekta pokazuju, trenutno ne postoji alternativno rješenje za estimaciju parametara gibanja nebeskih tijela te je izgledno korištenje upravo ove implementacije.

Jedno od mogućih budućih pravaca istraživanja su "particle filteri" koji održavaju skup točaka koje u svakom trenutku opisuju posteriornu distribuciju vjerojatnosti. Radi se o vrlo širokom polju različitih pristupa i metoda te bi primjena neke od njih na Multifit problem bila vrlo zanimljiva. Drugi pravac daljnjeg istraživanja moglo bi biti poboljšavanje otpornosti metode na prisutnost sjajnih zvijezda u blizini objekta na način da se održava

nekoliko alternativnih “rješenja” u obliku komponenata Gaussove smjese. Treći pravac mogao bi biti poboljšanje performansi metode korištenjem GPU procesora. Konačno, slijedno osvježavanje nije ograničeno na ovaj problem te ga je moguće primijeniti i na druge probleme u astronomiji.

KLJUČNI POJMOVI: astronomija, uparivanje podataka, Apache Spark, Bayesova statistika, slijedno osvježavanje

CONTENTS

1

Introduction

This chapter gives an overview of the two main topics that this thesis deals with: cross-matching of astronomical catalogs and parameter estimation of a moving point source model.

1.1 Cross-matching of astronomical catalogs

1.1.1 *Background and motivation*

Astronomical surveys and catalogs. Astronomical surveys are maps of regions of the sky typically created as part of an astronomical project, often tied to a particular telescope, producing an *astronomical catalog*. Astronomical catalogs are large databases containing various measurements of astronomical objects' properties. Surveys can be contrasted to observations of a specific target, such as a nebula, galaxy, supernova, etc.

Along with the progress of related technologies, scopes of astronomical surveys and sizes of datasets available in their catalogs are growing at an ever-increasing rate. For example, the 2MASS survey (active 1997-2003) produced a catalog with 470 million sources with the total size of about 40 GB [8]. The latest release (DR14) of Sloan Digital Sky Survey [9] (SDSS; active since 2003) contains 1.2 billion objects with the total data size of about 150 GB. The European space telescope Gaia [10], active since 2016, observing about 1.8 billion stars in our galaxy and aiming to create the largest and most-precise 3-D map of the Milky Way, produced its third data release in 2022 with almost 800 GB in the main catalog (source measurements) and more than 6TB in total (with astrophysical parameters, photometry and spectroscopy measurements etc.) [11].

In the near future, Vera C. Rubin Observatory (formerly known as "Large Synoptic Survey Telescope" or LSST), with its aim of producing the first *video* of the visible universe in history, is expected to acquire about 1000 observations of close to 20 billion objects during the first 10 years of its lifetime (with *catalog* dataset size expected to be larger than 10 PB [12]).

Radio astronomy produces even larger amounts of data so that most of them cannot even be stored in catalogs and need to be filtered out during acquisition. The research presented here, however, is confined exclusively to optical astronomy.

Analyzing astronomical catalogs. Astronomers today access datasets produced by astronomical surveys typically through data archives available online, such as

Strasbourg astronomical Data Centre, or SDC [1]; Mikulski Archive for Space Telescopes, or MAST [2]; and NASA/IPAC Infrared Science Archive, or IRSA [3]. These online interfaces allow for efficient searching, filtering and extraction of catalog data stored in relational databases.

Researchers typically begin their analyses by downloading a subset of the catalog data to their local cluster (or a machine), typically as FITS files ("Flexible Image Transport System" is the standard data format used in astronomy[4]), and then proceed by using custom scripts (usually written in Python) or code organized in Jupyter notebooks [13]. This workflow has been used effectively in the past for producing some of the most impactful results in Astronomy.

CROSS-MATCHING OF ASTRONOMICAL CATALOGS.Beyond working on a single catalog, astronomers often want to positionally cross-match objects from two (or more) surveys in order to link data about the same physical object observed with different telescopes, optical filters or at different times. Positional cross-match, a fundamental operation in analyzing survey data, is a join of two catalogs based on the distance between their objects: finding k nearest neighbors in catalog B of each object in catalog A. For example, accurate maps of distributions of stars and interstellar matter were recently constructed by combining information from the SDSS, Pan-STARRS, and 2MASS catalogs [14]. Furthermore, cross-matching can be done probabilistically if it also considers measurement uncertainties [15].

ANALYZING ENTIRE CATALOGS.Analyses spanning entire astronomical catalogs are becoming increasingly important. Examples of analyses that require whole-catalog processing include explaining the nature of Dark Energy, automatic classification of observed objects and searching for outliers (e.g. [16] and [12]).

ANALYZING TIME-SERIES DATA.Many important astronomical surveys (and some of the previously mentioned ones) are multi-epoch surveys, meaning that they come with a *time domain component*: they repeatedly observe the same objects. These result in catalogs with *time series* of object properties (often called *light curves*). Some of these surveys also produce real-time alerts when they observe changes in object properties (e.g. a supernova explosion).

1.1.2 *Problem statement*

With increases in data volumes of the upcoming astronomical surveys, and with data analysis requirements that were just described, it is increasingly obvious that the RDBMS-based systems and their data downloaded through web interfaces are becoming a bottleneck for scientific research in astronomy. RDBMS-based systems have difficulties supporting algorithms that need to frequently go through entire catalogs. Also, positional cross-matching

[1] `http://cds.u-strasbg.fr/`
[2] `https://archive.stsci.edu/`
[3] `https://irsa.ipac.caltech.edu`
[4] For more information see `https://fits.gsfc.nasa.gov/`

of catalogs is computationally extremely expensive and usually not straightforward to implement with a RDBMS-centric system.

An alternative approach is to migrate RDBMS backends to modern, scalable, industry-standard, distributed data processing systems and store data in columnar file formats, a setup which would allow for efficient data processing and would map well to the requirements posed by the modern astronomy. However, such systems are slow to penetrate into astronomy community mostly because they are difficult to use and deploy and lack astronomy-specific functionality, such as cross-matching of astronomical catalogs, spatial selection or time-series support.

REQUIREMENTS FOR A MODERN ASTRONOMICAL DATA ANALYSIS SYSTEM. The preceding considerations can be summarized into the following list of requirements for a system that can serve astronomers in analyzing future large sets of astronomical data.

1. **Support for astronomy-specific functions**: in order for the data analysis system to be useful to a broad community of astronomers, it needs to support astronomy-specific functions (positional cross-matching and spatial querying being the most important ones),
2. **Efficiency**: the principal operations (positional cross-matching, filtering, and scanning through the entire dataset) need to be as fast as possible,
3. **Light curve functions**: ability to manipulate series of observations of a single object,
4. **Scalability**: ability to handle highly skewed datasets at petabyte scale,
5. **Ease of use**: enable astronomers to perform ad-hoc analyses by freely combining SQL syntax and Python code,
6. **Use of industry-standard frameworks and libraries**: this would make the system more maintainable and allow for reuse of services and code from other areas in the industry.

FAST, ON-LINE CROSS-MATCHING OF CATALOGS. Being a fundamental astronomical operation, as was already stated, efficient cross-matching function is an essential part of a modern astronomical data analysis system. Although conceptually simple, designing and implementing it on an astronomical scale is not trivial. The cross-match operation is often made possible by pre-computing cross-match tables between catalogs, but as the number of catalogs grows this becomes inefficient ($O(N^2)$, where N is the number of catalogs).

An ideal system would offer sufficiently fast spatial cross-matching of catalogs so that it can be done on the fly, as required by the researcher.

1.2 PARAMETER ESTIMATION OF A MOVING POINT SOURCE MODEL

1.2.1 *Background and motivation*

ASTROMETRY, PHOTOMETRY AND SPECTROSCOPY. The immediate goal of optical astronomical surveys is to first detect light sources in the acquired images, associate them

to physical objects (stars and galaxies) and then determine physical attributes of the objects as accurately as possible. Measuring positions of objects in the sky is called *astrometry*. Astronomers are interested in objects' luminosity (total energy output), temperature, color, size, shape, and other properties. These are all based on flux (electromagnetic radiation) measurements, i.e. photometry and spectroscopy.

Photometry measures object's total flux. Spectroscopy measures object's flux as a function of wavelength in order to determine the object's chemical composition. Spectroscopy is much more expensive than photometry and is not possible in low signal-to-noise (S/N) environments. As a less precise, but also a less expensive and hence feasible alternative, astronomers use photometry along with wavelength filters, which amounts to low resolution spectroscopy [17]. Filters let only a discrete range of wavelengths to pass through them. SDSS uses filters denoted as `u` (which comes from "ultraviolet"), `g` ("green"), `r` ("red"), `i` ("infrared"), `z` (beyond infrared). LSST also adds the `y` filter (further beyond the `z` filter).

A *Filter response* diagram shows a system's *transmission*, i.e. the number of electrons produced in the system by each incoming photon. Figure1.1shows the response of LSST's detector system when using different filters.

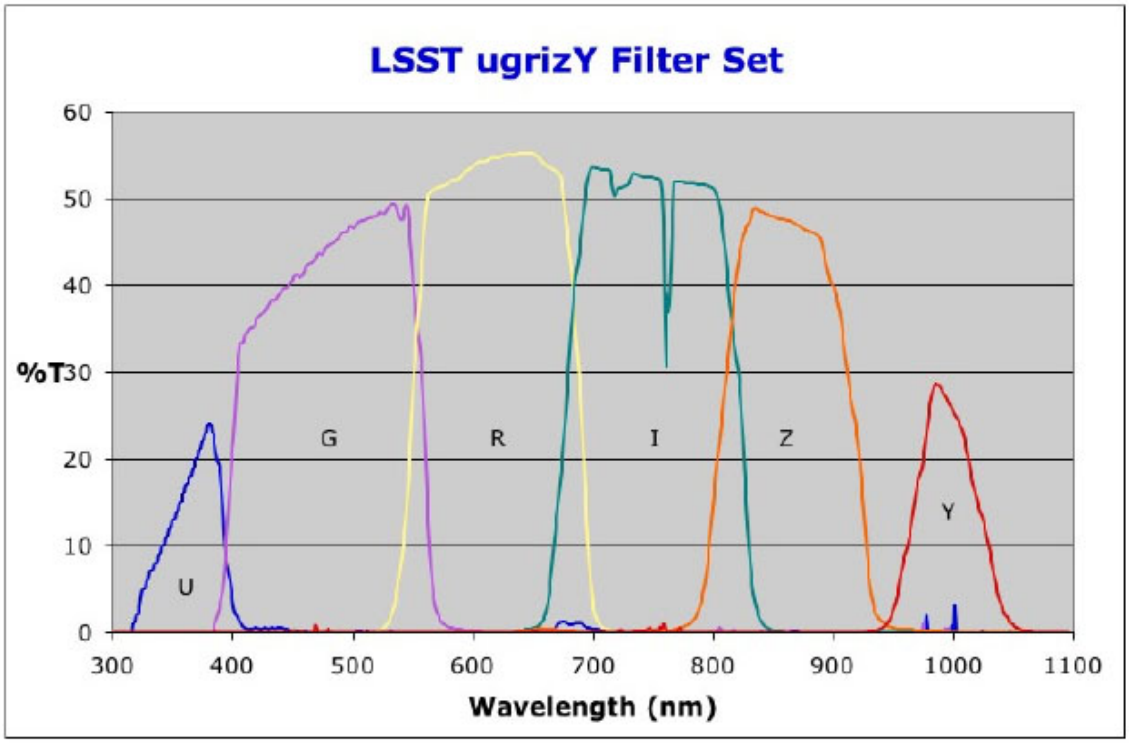

Figure 1.1: Response of LSST's detector system for each of its six filters as a function of wavelength. Filter response is the number of electrons produced in the system by each incoming photon. Image source: https://www.lsst.org/sites/default/files/img/ugrizY.jpg

There are two main methods of photometric measurements: PSF photometry and aperture photometry. PSF photometry (PSF stands for "point-spread function", defined in the next section) measures object's flux by fitting a PSF model to a star (reliable PSF estimation is critical here). Aperture photometry simply sums up pixel values within an *aperture*, a circular area centered on the object. PSF photometry is more accurate and

aperture photometry approximates it well if a star is much brighter than the background, if PSF cannot be estimated reliably, or when measuring non-point sources [18].

ASTRONOMICAL IMAGE RESTORATION AND SEEING.PSF determines the shape a point source produces in the telescope's focal plane. PSF depends on time, instrument state, source position and source color [19]. For *diffraction limited* optical systems, such as space-based telescopes [17], the Airy function was shown to be the best choice of PSF function. Large ground-based telescopes are "seeing limited", "seeing" being defined as full width at half maximum (FWHM) of the PSF [17]. Atmospheric turbulence is the main component of ground-based telescopes' PSF and it changes through time similarly to how air vacillates above a hot road. As a consequence, on longer exposures of several seconds or more, the changing PSF causes light sources to get smeared on the resulting image.

Estimating PSF is a critical problem in ground-based surveys. It is especially important for measuring weak lensing effects [20] where having an accurate PSF model is critical for galaxy shape measurements.

IMAGE COADDITION.Image coaddition is a technique traditionally used in astronomical surveys for obtaining "deeper" images, improving the PSF and removing cosmic rays and other artifacts [18]. Astronomical images are mostly static (galaxies and stars do not move much), so multiple images can be combined to increase S/N. The co-added image ("coadd") has smaller seeing than original images and the PSF of the coadd can approach diffraction limit. For example, in [21] images of the so-called *Stripe 82* (a 2.5 deg wide region repeatedly observed by SDSS) were coadded to obtain images about 2 mag ("magnitude" is a measure of brightness) deeper (meaning "sharper" and "brighter") and with a median seeing of approximately 1.1" (1.1 arc-seconds), compared to 1.4" seeing of the original images. Zackay and Ofek in [22] obtain 0.35" FWHM in co-added images (compared to the diffraction limit of 0.3") for seeing conditions of 1.2"-1.5". Figure1.2shows an example of deeper objects visible on a coadd from the SDSS project.

1.2.2 *Problem statement*

Accurate coaddition, which preserves scientific and statistical information in original images, is not trivial. Different images are typically taken with different PSFs and backgrounds. PSF also varies across the image. Simply averaging pixels of different images is not sufficient as it would result in discontinuities in variation of PSF across the coadd [16]. Some astronomical measurements, such as measurements of galaxies' ellipsicities which are important for weak lensing measurements, rely on accurate PSF model estimation. PSF estimation of the resulting coadded image is the main problem when coadding images.

While image coaddition is extremely useful for detecting faint objects, and while it eases computational requirements of multi-epoch image processing, it results in images where discontinuities in variation of PSF across the coadd (at the borders of individual epoch images) and the correlated noise [2][16] make accurate estimation of the coadd PSF difficult. The optimal coaddition methods previously mentioned ([18][22]) have limitations: they are applicable only to images where background noise dominates or to images without blended

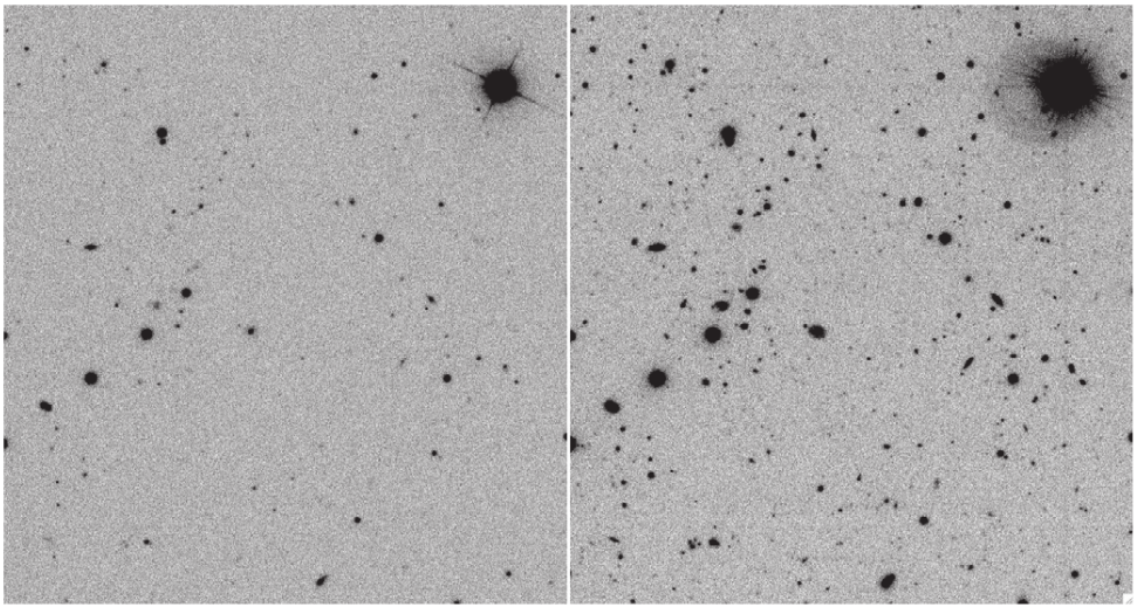

Figure 1.2: SDSS coaddition example ([21]). The original description: "Comparison between single pass (left) and coadd (right) images in the r band for run 206, camcol 3, field 505, centered at R.A. = 15 deg, decl. = 0 deg. Images are shown with the same scale, contrast and stretch. The single pass counterpart (run 5800, camcol 3, field 505) is one out of 28 images used in the coaddition of this particular image. This example illustrates the fact that a large number of objects below the detection threshold of each image can be well detected and measured in the coadd."

(i.e. overlapping) objects. These are important issues in areas where extremely accurate PSF modeling is important, such as weak lensing estimations from galaxy shape measurements.

More fundamentally and most importantly for this thesis, when coadding images of moving sources, the result is a blurred coadd, not appropriate for physical measurements. A method that preserves information from all individual images of an object and uses that information to increase signal-to-noise ratio without depending on coadds is called Multifit ([2][3]). However, that method, because it is based on "traditional" or "frequentist" statistics, provides only point estimates of model parameter values while obtaining a full posterior probability distribution using alternative methods requires much more computing power. This thesis proposes an alternative method called "Online Multifit" which produces estimates of the posteriors but only handling a single image at a time, thus reducing computational requirements. The method is applied to the problem of estimating parameters of a moving point source model, described in section3.2.2.

1.3Scientific contributions of the thesis

The original scientific contributions of the thesis are the following.

Zone-based method for fast distributed cross-matching of big astronomical data An original method called "Distributed Zones Algorithm" is developed based on the Zones Algorithm [1]. It comprises a distributed data organization scheme and an efficient method of positionally joining data using a moving window. The algorithm is implemented within

a system called AXS (Astronomical Extensions for Spark), based on Apache Spark and extended with astronomy-specific functionalities. System's cross-matching performance has been extensively tested in single-machine and "cloud" environments.

Method for parameter estimation of a moving point-source model based on sequential Bayesian updating An original method called "Online Multifit" is developed by extending the Multifit method. Online Multifit utilizes Bayesian statistics to obtain the full posterior probability distribution of moving point source model parameters but it does so by processing one image at a time using "sequential updating" technique, updating the posteriors after each image. The method approximates posteriors as mixtures of Gaussians and the main challenge here is to reduce errors introduced by these approximations. The result is that the method uses less computational resources while obtaining full posterior estimates. Furthermore, the estimates can be stored in a compressed form (the means and covariances of Gaussian distributions).

1.4 THESIS OUTLINE

The thesis is organized as follows:

Ch 2 This chapter deals with the topic of fast cross-matching of large astronomical catalogs. Its first section gives an overview of existing systems and algorithms that tried to solved the problem. The second section formally describes the problem of positional cross-matching of catalogs and gives a detailed description of the "Distributed zones" algorithm, one of the scientific contributions of the thesis. In the third section, a description with details about implementation of the algorithm within the system called "AXS", based on the Apache Spark platform and created during the work on the thesis. The same section gives an overview of AXS' functionalities meant to be used by astronomers. In the last, fourth section ("Experimental results") detailed descriptions of test results of cross-matching well-known astronomical catalogs using AXS conducted on a single machine and "in the cloud" are given.

Ch 3 This chapter deals with the topic of estimating parameters of a moving point source model. In its first section, an overview of development of the "Multifit" idea is given, where instead of using image coaddition, original images are compared to the images generated based on the supposed model (so called "forward modeling"). In the second section, the differences in approach to the Multifit problem depending on whether "traditional" or "Bayes" statistics are used are described, and mathematical foundations of the problem are laid down. In the third section the main idea comprising the core of the second scientific contribution of the thesis is described: application of the method of Bayesian sequential updating of the posterior probability distribution (of model parameters), sequentially with each new image, in order to obtain an estimate of the shape of the posterior distribution with smaller memory cost. The fourth section brings the details and source code for Multifit algorithm implementation using traditional statistical methods in the form of a LSST "task", ready to be used as part of Vera C. Rubin telescope data processing. The fifth section gives the details and source code of Multifit algorithm implementation using methods of Bayesian statistics,

but using all images at once (the so called "batch" approach). The sixth section brings the details and source code of the Online Multifit idea implementation. The seventh section gives the results of accuracy and speed testing of all three implementations.

Ch 4 In this chapter, the conclusions resulting from the conducted doctoral research are presented and potential future directions for further research are given.

2

Fast Cross-Matching of Large Astronomical Catalogs

This chapter handles the theme of fast cross-matching of large astronomical catalogs, which is actually a positional join of astronomical observation data. The positional join is based on object coordinates in an astronomical coordinate system.

This thesis proposes the *distributed zones algorithm* (see section2.2.2), comprising a specific data organization scheme and a version of an *epsilon join*, as a solution for fast, distributed positional cross-matching operation. The proposed data partitioning scheme also elegantly solves data skew issues so often present in astronomical datasets (see section 2.2.2.4).

This thesis also introduces Astronomical eXtensions for Spark (AXS) [23], a system based on Apache Spark [1], enriched with astronomy-specific functions and containing an implementation of the distributed zones algorithm. The goal of AXS is much broader than cross-matching of astronomical catalogs. Its main aim is to offer a viable option for a modern astronomical data analysis system. Main requirements for such a system were listed in section1.1.2.1. AXS functionalities in that regard are described in section2.3.3.

Performance testing described in section2.4was also performed using AXS. The results show that the approach used by AXS is superior performance-wise to other systems solving the same problem, listed in the next section.

AXS has been used since 2018 at Dirac institute[2] at the University of Washington for working on Zwicky Transient Facility (ZTF) data (for example, to "efficiently identify candidate transits of likely white dwarf stars in the large ZTF data set" in [6]) and for Vera C. Rubin Observatory (formerly known as LSST) use cases ([7]), but also by other astronomy research groups around the world.

AXS is currently being evolved by Dirac institute into a system internally called "HiPSCat". The main motivation for this change is replacing the Parquet file storage format with an extension of IVOA HiPS standard[3], which is directly readable with Python. The plan for this next generation of AXS is to be used as the data processing system for Vera C. Rubin Observatory and other NASA projects.

[1] The main web site is at: `https://spark.apache.org`

[2] See https://dirac.astro.washington.edu/ for more information

[3] See `https://www.ivoa.net/documents/HiPS/` for more information

2.1 RELATED WORK

Here, a list of projects with similar goals to AXS and offering catalog cross-matching capabilities is given. Large Survey Database (LSD) is described in more detailed as it was foundational work behind AXS.

2.1.1 *The Large Survey Database*

The Large Survey Database [24, 25] is a computing framework and distributed database management system for querying, cross-matching, and analysis of large survey catalogs. It is highly scalable (can scale to more than 100 nodes) and is optimized for parallel scans of positionally $(longitude, latitude)$ and temporally $(time)$ indexed datasets. It implements a "shared nothing" architecture (i.e. the one where nodes do not share memory or disk resources and use them independently in order to maximize concurrency). The LSD code is available at `http://github.com/mjuric/lsd`.

Google's BigTable [26] distributed database and the MapReduce [27] programming model have influenced LSD's design and some of its terminology.

LSD'S DATA ORGANIZATION. LSD vertically partitions tables into *column groups* containing related data (such as astrometry or photometry) and horizontally into space and time cells of equal sizes in sky pixels [HEALPix; 28] and equal time intervals. The partitioning scheme maps to compressed and checksummed HDF5 files (*tablets*) organized in a distributed directory structure. High performance is achieved by storing related data close together and loading data in large chunks.

LSD'S PROGRAMMING MODEL. LSD implements a subset of SQL DML [29] with syntax extensions that allow for freely mixing Python and SQL. Users can also write computing *kernels* that are applied on partial query results at cell level and transform or aggregate them before further processing. Such granular processing steps are organized in directed acyclic graphs (DAGs) of execution, similarly to Dask [30] and Spark [31], and the framework distributes, schedules and executes the lazily-computed DAGs.

AREAS FOR IMPROVEMENT. While LSD got adopted by a number of research teams as a ready-made solution for querying, sharing and analyzing large datasets, there are still some major areas in need of improvement:

- **Partitioning skew:** Because of its fixed, non-hierarchical partitioning scheme, LSD suffers from significant partitioning skew.
- **Problematic temporal partitioning:** LSD enables fast "time slicing" because its tables are partitioned on time. Most real-world users, however, use queries that request all data for a certain object. The temporal partitioning scheme makes such queries slower than what would otherwise be possible.
- **Not resilient to failures:** LSD is not resilient to failures of individual processes.

- **Legacy code:** LSD has custom implementations of functionalities available today in more mature and widely adopted packages, such as Pandas, AstroPy, scikit-learn, and Apache Spark. It is also written in Python 2.7, which is not supported anymore.

2.1.2 *Other systems*

Work has been done on a number of other systems focused on providing efficient cross-matching and processing of large astronomical catalogs:

- *catsHTM* is a tool for "fast accessing and cross-matching of large astronomical catalogs" [32]. It uses Hierarchical Triangular Mesh (HTM) for partitioning and indexing data and stores them in HDF5 files. The authors report that it takes about 53 minutes to cross-match 2MASS (470 million objects) and WISE (560 million objects) catalogs (without saving the results).

- *ASTROIDE* [33] is a system based on Apache Spark, similarly to the work presented in this thesis. ASTROIDE also offers an API for cross-matching and processing astronomical data. However, the data partitioning scheme used is HEALPix, resembling Large Survey Database's approach. The authors tested their system using 80 CPU cores spread over 6 nodes and report that ASTROIDE needs about 800 seconds for cross-matching 1.2 billion with 2.5 million objects.

- *Gaia survey data pipeline* The cross-match function implemented in the Gaia survey data release pipeline is described in [34]. That algorithm requires the data to be sorted by declination so that the data can only be read once. The algorithm differentiates between good and bad match candidates and stores each in separate tables. The algorithm is implemented in MariaDB, with custom performance optimizations. The authors report that the time required to cross-match Gaia DR1 data set (1.1 billion objects) and SDSS DR9 (470 million objects) is 56 minutes ([34]).

- *Open SkyQuery's* cross-match implementation is described in [35] by Nieto-Santisteban et al. They base their implementation on the *zones* algorithm and implement it on Microsoft SQL Server. Cross-matching SDSS (350 million objects) and 2MASS (28 million objects) catalogs takes about 20 minutes when using 8 machines.

- Authors Jia et al. developed an algorithm [36] for cross-matching catalogs in heterogeneous (CPU-GPU) environments. The data is indexed using HEALPix method. The authors report that it takes 10 minutes to cross-match SDSS data (1.2 billion objects) with itself using multiple nodes with high-end GPUs.

- A probabilistic cross-matching algorithm is implemented in [37] by Dobos et al. The data is partitioned based on zones and each machine contains full copy of the data. Their cross-match is not only based on distances, but on other criteria as well. All of the criteria contribute to the final calculation of the likelihood. Their multi-node SQL queries are orchestrated using a complex workflow framework. The authors do not report any performance numbers.

2.2FAST, ON-THE-FLY, POSITIONAL CROSS-MATCHING

In this section, cross-matching operation is defined more formally and a solution for fast, distributed cross-matching, *the distributed zones algorithm*, is explained in detail: the data partitioning scheme used, the mechanics of the join operation by independent processes and (the lack of) data skew.

2.2.1 *Cross-matching objects in astronomical catalogs*

As was already described in1.1.1.3, researchers in astronomy often want to combine observations from two (or several) catalogs that correspond to the same astronomical objects. The simplest version of this problem comes down to finding all observations, from two or more catalogs, that are less than some angular distance apart.

More formally, if L and R are the left and right relations (catalogs/tables), the result of a cross-matching join operation is a set of pairs of tuples l and r such that the angular distance between them is less than the defined threshold of ϵ:

$$\{(l,r) \mid (l,r) \in L \times R, dist(l,r) \leq \varepsilon\} \tag{2.1}$$

This is graphically illustrated in Figure2.1.

If there were more than one match in the right relation, for a particular item in the left relation, the cross-matching operation thusly defined returns all such matches. A *nearest-neighbor* join is the one where only the match with the minimum distance (which still has to be smaller than the defined ϵ threshold) is included in the result. Finally, the $dist$ function can be differently defined, using different distance measures, taking into account other attributes (such as ellipticity, for example, which can differentiate stars from galaxies), or can even be probabilistic in nature (as was already mentioned in1.1.1.3).

2.2.2 *Distributed zones algorithm*

A critical question with cross-matching is how to organize data so that spatially-related data is stored close together and is available for quick searches. There are known algorithms that use clever combinations of indexing and advanced SQL query optimizer features to achieve the needed functionality.

HEALPix (Hierarchical Equal Area and isoLatitutde Pixelization of the sphere) [28] and HTM (Hierarchical Triangular Mesh) [38] are the two indexing schemes that have traditionally been popular for indexing astronomical data. They both organize the sky into meshes of arbitrary granularity.

The zones algorithm [1] uses a different approach. The basic idea is to divide the sky into horizontal stripes called "zones" which serve as indexes into subsets of catalog data so as to reduce the amount of data that need to be searched for potential matches. In one of the earlier papers [39] Gray et al. compare HTM, which is used in SDSS' SkyServer [40], and the zones approach and find the zones indexing scheme to be more efficient when implemented within relational databases.

In this thesis, the zones algorithm is further developed and adapted for a distributed, shared-nothing architecture. The **Distributed Zones Algorithm** comprises a distributed

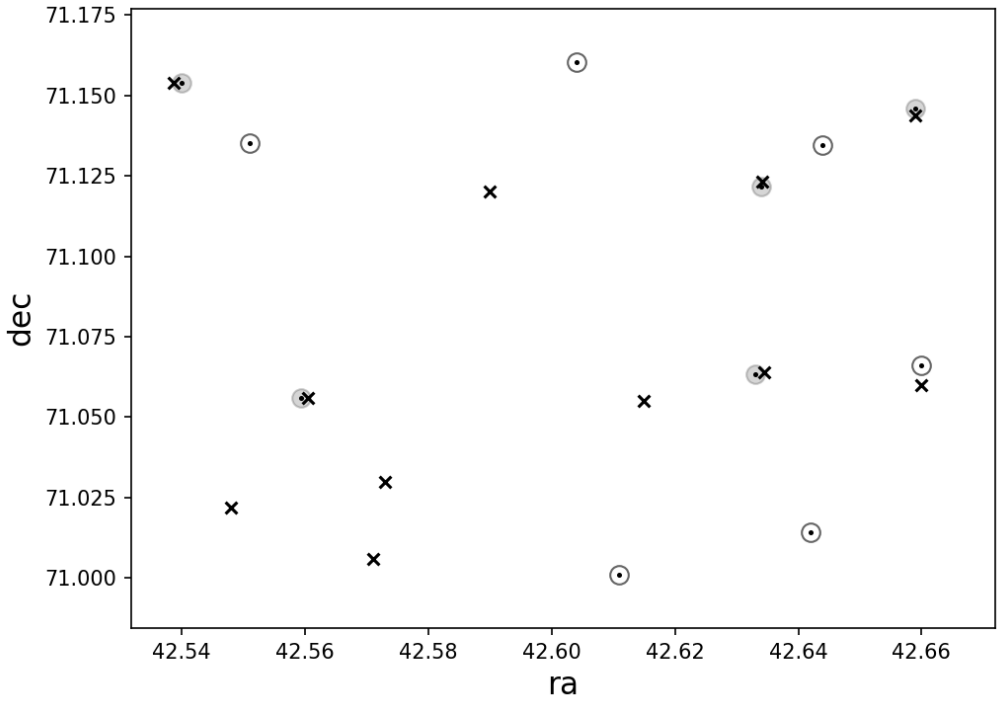

Figure 2.1: An example of cross-matching two catalogs in a 10 arcmin x 10 arcmin region of the sky. Objects of the two catalogs being matched are represented as dots and crosses. The circles show the search region around each object of the first catalog. Circles are filled if their area contains a match.

data organization and an indexing scheme, along with an implementation of an efficient *epsilon join*. These are described in more detail in the following sections.

↬ EPSILON JOIN WITH A DISTANCE FUNCTION.The general idea is to express the cross-join operation as a query of the form shown in Listing2.1and execute it in parallel over distributed data.

Listing 2.1: Example of an epsilon join

```
SELECT * from GAIA g JOIN SDSS s ON g.zone = s.zone
  AND g.ra BETWEEN s.ra - e AND s.ra + e
  AND distance(g.ra, g.dec, s.ra, s.dec) <= e
```

The first two conditions of the query comprise what is known as an *epsilon join* [41]: an equi-join on the primary column (*zone*) and a range condition on the secondary column (*ra* in this case). Epsilon parameter *e* defines the maximum distance between object pairs that will be returned and *distance* is a function which calculates distance between two points defined by their RA (right ascension; *ra* column) and Dec (declination; *dec* column) coordinates.

↬ BUCKETING DATA AND REDUCING DATA SKEW.In order to efficiently cross-match catalogs using the previously described epsilon join, the data need to be organized appropriately. As was already stated, distributed zones algorithm divides the sky into N

horizontal zones but it also enables parallel processing by physically partitioning the data into `B` *buckets*, physical files containing distinct subsets of data, intended to be read by separate processes.

All the objects from the same zone are stored in the same bucket and the zones are placed in buckets sequentially: zone `z` is placed into bucket `b = z % B`. Figure2.2shows an example for `N=16` zones and `B=4` buckets, but in reality, thousands of zones and hundreds of buckets are used.

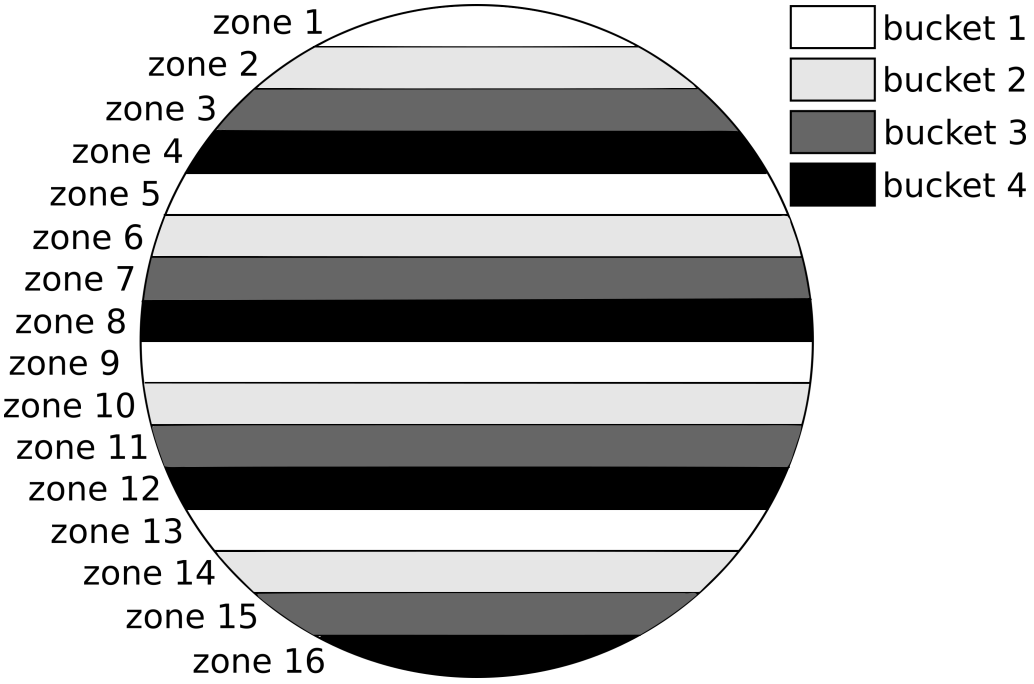

Figure 2.2: An example with the sky (shown as a circle) partitioned into 16 horizontal zones and placed into 4 buckets. Objects from each zone get placed into buckets sequentially. In reality, there are thousands of zones.

SORTING AND JOINING DATA WITHIN BUCKETS.Data within buckets, in the distributed zones algorithm, are sorted by `zone` and `ra` columns (in that order). These two columns serve as indexing columns for the cross-match operation.

With data sorted like this, two *buckets* can then be efficiently joined using an epsilon join (described previously in section2.2.2.1) in *one pass through the data* by maintaining a moving window over the data in the right relation (catalog). This is graphically shown in Figure2.3. The moving window is defined by the equi-join condition on the primary column (`zone`) and the range condition with the epsilon distance on the secondary column (`ra`).

If two catalogs are partitioned in the same way (having the same number of buckets and the same zone heights), objects from the same buckets can then be joined independently of the other buckets. An arbitrary number of processes can then join pairs of buckets in parallel, which makes the whole cross-matching operation scalable.

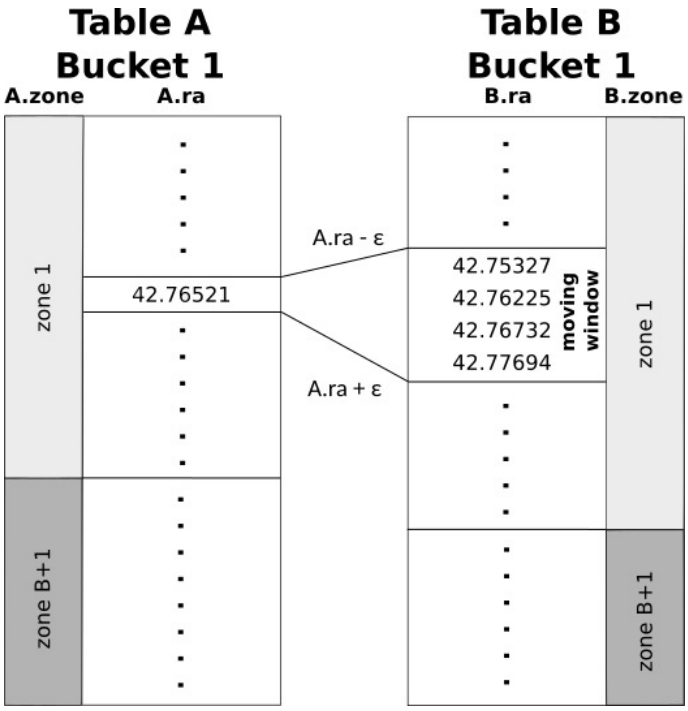

Figure 2.3: Using the "epsilon join" to reduce the number of rows for which distance is calculated. For the match candidate row in the figure, only four distance calculations are performed. (B stands for the number of buckets.)

DATA SKEW CONSIDERATIONS. Astronomical objects are unevenly distributed on the sky. Milky Way, for example, contains much more stars than the rest of the sky. Hence, large survey catalogs often include highly skewed data. If naively partitioned, processing such data in parallel could cause some processes to use large amounts of memory and last much longer than the others, which could considerably degrade the overall performance.

When sequentially placing thin stripes of sky into different buckets, as is done in the distributed zones algorithm, the data gets evenly distributed between the buckets which eliminates data skew.

CORRECTNESS AT ZONE BOUNDARIES. A weakness of the approach described so far is the lack of cross-matching completeness and correctness at zone boundaries. Because objects situated near zone boundaries might have a match in a neighboring zone, the processes strictly joining only pairs of buckets might miss such matches without additional data exchange between processes (without “looking” into the neighboring zone). This data movement at runtime would significantly hurt the performance.

One solution is to place duplicates of objects from the lower “border stripe” of each zone into the zone below it and mark such objects as duplicates. These borders need to be sufficiently wide in order to cover any search radius that might be used in real life (*epsilon* value). *10* arcsec is a reasonable default, capable of handling datasets ranging from SDSS to Gaia scales.

This approach slightly increases the amount of catalog data but enables processes to run independently. At the same time, however, the approach complicates other operations: the duplicated rows need to be excluded from ordinary query results. Additionally, correction

is needed due to the fact that these duplicated objects will sometimes be cross-matched twice (once inside their original zone and once inside the neighboring zone).

ZONE HEIGHT AND NUMBER OF BUCKETS. Number of buckets used in the distributed zones algorithm is a trade-off between the amount of data that need to be processed by a single process (the bucket size) and the maximum possible number of processes (the maximum parallelism). A single bucket can be processed by a single task only and a single task would need to process several buckets one by one (serially).

Zone height is also a trade-off: thinner zones reduce data skew but increase the amount of data that need to be duplicated (because thinner zones means more zones, which means more zone boundaries).

2.3 IMPLEMENTATION OF AN ASTRONOMICAL DATA ANALYSIS SYSTEM ON TOP OF APACHE SPARK

The distributed zones algorithm, described in the previous section, was implemented in a system called AXS [4]. More than that, AXS is an attempt at creating a modern astronomical data analysis system, requirements for which were described in section1.1.2.1, by adapting one of the standard big data analytic tools in industry – Apache Spark.

In this section, AXS and Apache Spark, as its foundation, will be described in more detail along with the implementation specifics of AXS' cross-matching function and other astronomy-related features it provides.

2.3.1 *About Apache Spark*

Apache Spark is a fast and general-purpose engine for big data processing, with built-in modules for SQL, machine learning, near-real-time processing and graph algorithms [31, 42]. Its development started at UC Berkeley in 2009 and it has since become the dominant big data processing engine due to its speed (more than 10x improvement compared to Hadoop [43]), ease of use, broad functionalities, cross-language support, and integration with most of the industry-standard tools and systems in the field.

Spark is scalable and resilient to failures of its components, supports distributed, columnar file formats (such as Parquet [44]) and offers Python interfaces, which are familiar to astronomers. Compared to other similar projects, such as Dask [30], Spark is more mature and more widely used. Spark has been used at many large installations and has connectors to a large number of third-party databases and systems. It is still being actively developed which makes it a safe option to build on.

Spark was also recognized as a suitable platform for astronomical data analysis in [45], for example, where the authors describe *spark-fits* Spark connector "to handle arbitrarily large FITS files".

Spark satisfies most of the requirements for a modern astronomical data analysis system listed in section1.1.2.1. It enables astronomers to combine SQL and Python code, it uses

[4] The code is available at `https://github.com/dirac-institute/AXS` and the documentation at `https://dirac-institute.github.io/AXS/`

industry-standard frameworks and libraries, and it is highly scalable and efficient. AXS adds a few features that Spark itself lacks: an efficient spatial cross-matching operation and astronomy-specific functions. These are described in the sections to follow.

SPARK PROGRAMMING AND OPERATING MODEL.Spark abstracts processing operations through its notion of *resilient distributed datasets*, or RDDs, which are defined by a set of *transformations* that are applied to data *partitions* in parallel and in a fault-tolerant manner. Transformations, such as *map*, *filter* or *groupByKey*[5] create other RDDs. They are applied serially and form DAGs, *directed acyclic graphs*, which can be thought of as programs that are *materialized* once they are executed on a dataset. RDDs are materialized using *actions*, operations that need to evaluate the DAG and return a result. Examples of actions are *count* (returns number of elements in a dataset) and *sum* (returns sum of elements in a numeric RDD).

Spark elegantly abstracts the underlying distributed nature of the data RDDs represent. Working with RDDs seems like working with local data collections. Furthermore, if calculation of a partition fails, Spark can restart the calculation for that partition only, which makes RDDs resilient to failures.

Spark's transformations and actions can be used to express almost any data analysis algorithm and more advanced API's can be built on top of RDDs. An example especially important for this thesis are *Spark SQL's DataFrames*. While RDDs can operate on data in all kinds of formats, DataFrames are designed for handling structured data: tables with typed columns. Users can use the standard SQL or the related Java, Scala or Python DSL (domain-specific language) APIs to write queries that get translated into physical manipulations of data through Spark's Catalyst optimizer, which performs query planning and optimization as well as generates and compiles Java code to be executed during runtime.

On top of all of this are various end-user libraries: Spark MLlib, with implementations of major machine learning algorithms; Spark GraphX, with various graph algorithm implementations; and Spark Streaming, offering real-time processing capabilities. These APIs are available in Java, Scala, R, and Python. Python support, as was already stated, is especially important for analysis of astronomical data. The described layers of Spark architecture are shown graphically in Figure2.4.

2.3.2 *Implementing the Distributed zones algorithm*

As was already explained, the Distributed Zones Algorithm, developed by this thesis, consists of a distributed data organization and indexing scheme, and an implementation of an efficient *epsilon join*.

Parquet [44], a distributed and columnar file format, was chosen for AXS' data storage because it is well supported by Spark and widely used in Spark community, but is also very popular outside it. Parquet files can be distributed over many machines and processes and can also be *bucketed*, which are features that are central to the Distributed zones algorithm, as was explained in the previous sections.

[5] E.g., see `https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations` for explanations and a complete list

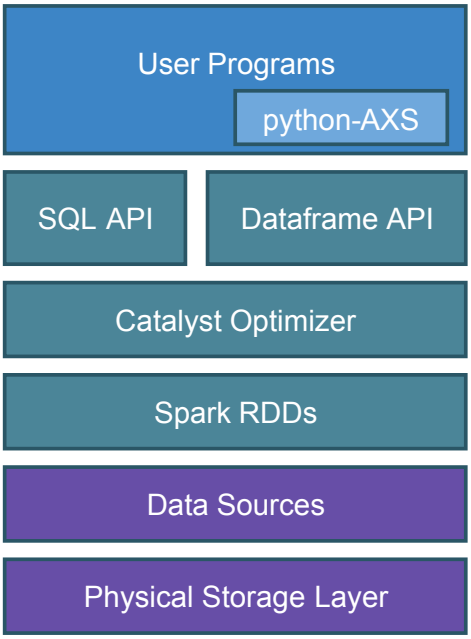

Figure 2.4: Layers in Spark's architecture, including AXS and other user programs.

AXS relies on Spark's DataFrame API for handling Parquet files, their organization and bucketing. The epsilon join implementation, however, is not available in Spark API itself and an extension of Spark's sort-merge join implementation was needed [6]. Without this intervention into Spark's source code, Spark's optimizer and code generator would calculate the distance function for all object pairs from the two zones being merged (determined by the first equi-join condition, i.e. the first part of the epsilon join), essentially doing a cartesian join. Such queries consume too much memory and cannot complete for datasets of realistic size.

The Spark code extension, as implemented in AXS, enables Spark's optimizer to recognize an optimization opportunity in the case of an epsilon join on a dataset sorted in an appropriate way, so that the distance function is calculated only for those row pairs, from the two zones being cross-matched, that match the prior `BETWEEN` condition. The algorithm in this way operates in a moving window, as was explained in section2.2.2.3.

2.3.3 *A walk through AXS Python API*

AXS Python API, as a thin layer on top of Spark's `pyspark`, is the main end-user interface to AXS. It was also developed as part of the work on this thesis. The main idea behind its design was to abstract away the implementation details of cross-matching function and data partitioning details and make it easy to use for astronomers. Main elements of AXS Python API are described in this section and the full documentation is available at `https://axs.readthedocs.io/en/latest/`.

[6] The code and discussion regarding this can be found here: https://github.com/apache/spark/pull/21109

An example of a cross-match using AXS API. AXS users interact with the system primarily through `AxsCatalog` and `AxsFrame` classes. These two are actually extensions of Spark's `Catalog` and `DataFrame` interfaces, respectively.

An instance of `AxsCatalog` is constructed using an instance of `SparkSession`, analogously to how Spark's `Catalog` is created:

```
from axs import AxsCatalog
axs_catalog = AxsCatalog(spark)
```

The `SparkSession` object encapsulates the connection to the Spark metastore database, and enables manipulation of Spark tables. An `AxsCatalog` instance is aware of the tables partitioned according to the distributed zones algorithm and is able to retrieve instances of such tables as shown here:

```
axs_catalog.list_tables()  # output omitted
sdss = axs_catalog.load("sdss")
gaia = axs_catalog.load("gaia")
```

The objects that `AxsCatalog` returns are `AxsFrame` instances. They extend Spark's `DataFrame` and add astronomy-specific methods, such as `crossmatch`, used for cross-matching of two `AxsFrame` tables.

```
from axs import Constants
gaia_sd_cross = gaia.crossmatch(sdss, r=3*Constants.ONE_ASEC,
    return_min=False)
gaia_sd_cross.select("ra", "dec", "g", "phot_g_mean_mag").
    save_axs_table("gaiaSdssMagnitudes")
```

The previous snippet performs positional cross-matching of the `gaia` and `sdss` catalogs. A subset of columns is then retrieved from the resulting catalog, and these are finally saved into a new, zones-partitioned table named `gaiaSdssMagnitudes`. Note that the DAG constructed thusly is executed lazily: only when `save_axs_table` is called. That is because `crossmatch` and `select` are *transformations* while `save_axs_table` is an *action* (already mentioned in section2.3.1.1).

The `crossmatch` function returns all matches within specified radius by default. If the `return_min` flag is set to `True`, `crossmatch` returns only the nearest neighbor.

2.3.4 *Spatial selection support*

Spatial selection is a query of objects in a region of the sky. One of the ways this can be done in AXS is by using *region queries*, which are determined by maximum and minimum RA and Dec angles. Because sky is partitioned by zones and zones are stored in buckets, such queries are executed in AXS as parallel, multi-process searches through the matching bucket files, thanks to the underlying Spark engine. Since AXS catalogs are bucketed by zones, AXS can skip whole files that do not contain the required zones. And since the bucket files are sorted by `zone` and `ra` columns, the search can be fast. The engine can also skip parts of files not containing the required ranges.

AXS user is not aware of these optimizations and region queries in AXS API are as simple as the following:

```
region_df = gaia.region(ra1=40, dec1=15,
                        ra2=41, dec2=16)
```

Another method of spatial selection in AXS is cone search using the `cone` method. It requires a center point and a radius and returns all objects with coordinates within the circle thus defined.

2.3.5 *Time series support*

Large-scale surveys repeatedly observe the same objects over and over again. This produces object's light curves: time series of the objects' flux (number of photons per second per unit area) measurements. The measurements typically include the flux, the timestamp, the error and perhaps some metadata, such as the filter that was used, for example.

Such time series data can be stored in AXS as a set of array columns in the catalog. This approach stores light curve data alongside the main catalog data, which makes it fast to retrieve and search for them. Spark supports array functions well, but it is also flexible enough to allow for other ways of time series data organization, if that is needed by the end user.

AXS provides two helper array functions to support light curve operations: `ARRAY_ALL-POSITIONS` returns an array of indexes of all occurrences of an element; and `ARRAY_SELECT` returns all elements indexed by a provided index array. These two functions can be combined with other built-in Spark SQL functions to further manipulate light curve data.

The following example in Listing2.2illustrates handling of light curves. It will return the number of all observations with *r* filter (the `band` column) in a catalog *ztf*:

Listing 2.2: Example of using `array_allpositions`

```
from pyspark.sql.functions import size, array_allpositions
ztf_rno = ztf.select(size(
    array_allpositions(ztf("band"), "r")))
```

`array_allpositions` searches arrays in the *band* column for *r*-band value and returns an array of indices of such array elements. The Spark's built-in function `size` returns the length of the indices array.

2.3.6 *Fast Histograms*

A common technique of summarizing data both in astronomy and other sciences is building *histograms* of a statistic as a function of some parameters of interest. For example, Hess diagrams contain counts of stars in a galaxy as a function of color and magnitude (brightness). AXS provides two functions for building histograms, implemented as thin wrappers around Spark API, taking advantage of its distributed and fault-tolerant nature.

The two functions are `histogram` and `histogram2d`, both Spark *actions*. When calling the `histogram` method, users provide a column definition and a number of bins into which the data is to be summarized. Analogously, `histogram2d(cond1, cond2, numbins1, numbins2)` summarizes the data in two dimensions, using the two column definitions (condition expressions) and the two number of bins provided.

An example is given in code Listing2.3. The code produces a 2D graph showing the density of differences in *g band* magnitude measurements between SDSS and Gaia catalogs, versus the same differences between WISE and Gaia catalogs. Both differences are binned into 100 bins.

As can be seen, the result from `histogram2d` can be directly forwarded to the `Matplotlib`'s `pcolormesh` plotting function.

Listing 2.3: An example of using `histogram2d`

```
from pyspark.sql.functions import coalesce
import matplotlib.pyplot as plt
cm = gaia.crossmatch(sdss, return_min=True). \
    crossmatch(wise, return_min=True)
(x, y, z) = cm.histogram2d(cm.g - cm.phot_g_mean_mag,
    cm.w1mag - cm.phot_g_mean_mag,
    100, 100)
plt.pcolormesh(x, y, z)
```

2.3.7 *Creating AXS tables*

Spark does not save intermediate results of computations, unless this has been requested explicitly. This saves resources and speeds up processing. However, there are common use cases where saving intermediate results is useful. For example, when two large catalogs are cross-matched and when those results will be repeatedly used in further processing. If those results are to be used in subsequent cross-matching operations with third tables, the results need to be saved in the same distributed zones format and registered in the `AxsCatalog` registry. The same is true for new catalogs that need to be saved in AXS as AXS tables.

AXS provides the `AxsFrame.save_axs_table` method for this common use case. It saves an `AxsFrame`'s data as a new table and partitions the data as was described in section 2.2.2. It assumes `ra` and `zone` columns are already present in the `AxsFrame`, or it can optionally calculate the `zone` column.

Saving AXS tables means that the data need to be repartitioned and sorted, which makes this the most expensive operation. Data partitioning and sorting is a necessary part of the cross-match operation, but it is performed in advance and only once, so that table joins can later be performed on the fly.

The results of experimental measurements of the time AXS needs to partition different catalogs can be found in section2.4.2.

2.3.8 *Support for Python user-defined functions*

Similarly to some relational databases, Spark allows for defining user-defined functions (UDFs) that can be applied on rows or groups of rows during execution of Spark's optimized queries. AXS offers to methods in `AxsFrame` class for this purpose which are thin wrappers around Spark's `pandas_udf` and `udf` functions. These AXS methods are `add_column` and `add_primitive_column`.

Their only purpose is to make it a bit easier for astronomers to run custom data processing functions on a row-by-row basis (i.e. to avoid using `@pandas_udf` and `@udf` annota-

tions) and make their code more readable. They can only be used when processing data row by row, which corresponds to Spark's `udf` function and Spark's `pandas_udf` function of type `PandasUDFType.SCALAR` (they cannot be used for `PandasUDFType.GROUPED_MAP` nor `PandasUDFType.GROUPED_AGG` UDFs).

Both functions accept a name and a type of the column to be added, the function to be used for calculating the column's contents, and names of columns whose contents are to be supplied as input to the provided function. The difference between the two methods is that `add_primitive_column` supports only outputting columns of primitive types, but is significantly faster because it uses Spark's `pandas_udf` support under the hood. `add_column` method uses the scalar `udf` functions, making it slower, but supports columns of complex types. `pandas_udf` is faster because it is able to handle blocks of rows at once by utilizing Python Pandas framework (and its vectorized processing).

2.3.9 *Adding New Data to AXS Catalogs*

In some use cases, new batches of data need to be added to existing AXS tables. This can be done with the method
`add_increment`:

```
add_increment(self, table_name, increment_df,
    rename_to=None, temp_tbl_name=None)
```

The method adds the contents of the `increment_df` Spark DataFrame to the AXS table with the name `table_name`, calculating zones, and bucket and sorting the data appropriately in the process. The name the old table will be renamed to and the name used for the temporary table can be customized with `rename_to` and `temp_tbl_name` parameters, respectively.

This operation is also expensive, similarly to `save_axs_table` (section2.3.7) because on top of partitioning and sorting the data it also needs to merge new and existing tables.

2.4 EXPERIMENTAL RESULTS

In this section results of testing AXS' cross-matching and data partitioning performance are given. The results show that the distributed zones algorithm implemented in AXS easily outperforms other systems that were listed back in section2.1.

2.4.1 *Cross-matching performance*

The astronomical catalogs that were used for testing AXS' cross-matching performance are listed in table2.1. Two scenarios were tested: when only the nearest neighbor match was returned and when all matches were returned by the cross-matching operation. The number of resulting rows for each combination of catalogs for both scenarios is given in table2.2. Furthermore, cross-matching duration for both scenarios was compared in the "warm cache" case (data was partially cached in operating system's memory buffers) and in the "cold cache" case (the OS memory buffers were empty) [7]. This caching mechanism is

[7] OS memory buffers were cleared by writing "3" to the */proc/sys/vm/drop_caches* file

Catalog	Row count	Row count - no duplicates	Size
SDSS	0.8 Bn	0.7 Bn	66 GB
Gaia DR2	1.8 Bn	1.7 Bn	431 GB
AllWISe	0.8 Bn	0.7 Bn	357 GB
ZTF	3.3 Bn	2.9 Bn	1.23 TB

Table 2.1: The catalogs used for cross-matching performance tests, with the number of rows, number of non-duplicated rows and size of compressed data.

Left catalog	Right catalog	Results - all	Results - NN
Gaia DR2	AllWISE	320 M	320 M
Gaia DR2	SDSS	227 M	126 M
ZTF	AllWISE	109 M	109 M
ZTF	SDSS	273 M	168 M
Gaia DR2	ZTF	92 M	49 M
AllWISE	SDSS	235 M	119 M

Table 2.2: List of catalog combinations used for cross-match performance tests, with the numbers of resulting rows when returning all matches or only the first nearest neighbor.

distinct from Spark's own caching mechanism (which was not used during the tests) and works on OS level.

For each scenario and case, the number of Spark executors in the cluster, determining the level of parallelism, was varied going from 1 to 28. Each executor had 12 GB of Java memory heap and 12 GB of off-heap memory at its disposal. Figure2.5shows the results graphically, and the raw numerical results are given in tables2.3and2.4for the first scenario and tables2.5and2.6for the second scenario. Each datum in the figure and in the tables is an average of three tests.

All tests were executed on a single, large machine. There was not much improvement in performance beyond 28 executors (processes) because of constrained resources of a single machine. Data used in the tests were stored on the local file system (hard disks).

The results of tests with cold cache show the cross-match performance that can be expected if there is no memory available for caching (the data needs to be read from disk). The system did not have enough memory for the whole datasets so even during tests with warm cache data had to be partly read from disk.

The results show that AXS outperforms other systems (described in Section2.1) in cross-matching operation, although it is difficult to compare them because of different architectures, datasets and algorithms that are used. It can be noted, though, that the best results presented here are in tens of seconds, while other teams report results in tens of minutes.

2.4.2 *Data partitioning performance*

In order for AXS to be able to efficiently cross-match data, they first need to be appropriately prepared: partitioned and bucketed by zone and sorted. This is an expensive operation that

Execs.	Gaia-AllWISE		Gaia-SDSS		ZTF-AllWISE	
	warm	cold	warm	cold	warm	cold
1	481	2079	363	1143	718	2897
2	287	979	243	665	356	1434
4	156	624	113	355	185	876
8	84	413	58	232	99	566
12	56	332	41	185	68	451
16	49	264	35	156	56	364
20	41	241	30	158	49	327
24	35	226	24	141	41	296
28	34	220	25	136	42	291

Table 2.3: Averaged raw cross-match performance results (in seconds), when returning all matches, for the first three catalog combinations, depending on the number of executors and whether cold or warm OS cache was used.

Execs.	ZTF-SDSS		Gaia-ZTF		AllWISE-SDSS	
	warm	cold	warm	cold	warm	cold
1	788	2401	823	2472	382	1335
2	401	1268	375	1331	185	610
4	216	734	197	969	90	414
8	109	450	99	611	48	256
12	79	370	71	488	35	186
16	62	308	56	422	27	159
20	54	289	49	373	23	140
24	45	257	42	341	20	122
28	45	241	41	334	20	131

Table 2.4: Averaged raw cross-match performance results (in seconds), when returning all matches, for the last three catalog combinations, depending on the number of executors and whether cold or warm OS cache was used.

Execs.	Gaia-AllWISE		Gaia-SDSS		ZTF-AllWISE	
	warm	cold	warm	cold	warm	cold
1	656	1875	491	1067	643	2428
2	327	955	246	542	322	1291
4	165	607	126	314	164	793
8	104	418	70	212	95	563
12	133	331	56	169	89	429
16	132	268	83	157	108	359
20	86	253	80	151	96	320
24	72	230	59	141	66	284
28	70	227	52	137	63	286

Table 2.5: Averaged raw cross-match performance results (in seconds), when returning only the first nearest neighbor, for the first three catalog combinations, depending on the number of executors and whether cold or warm OS cache was used.

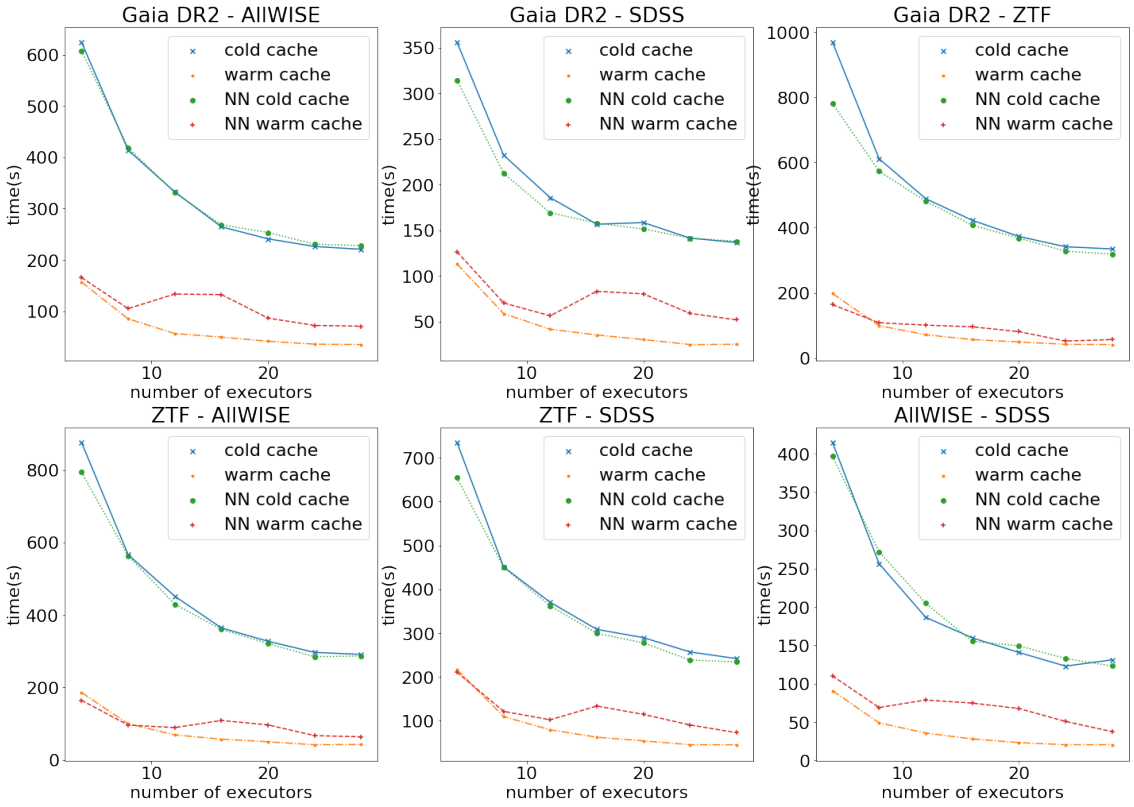

Figure 2.5: Performance tests of cross-matching various catalogs in scenarios with file system buffers empty or full (we used Linux OS-level caching, not Spark caching), when returning all matches or just the first nearest neighbor ("NN" results).

needs to be done once in advance and pays off later during runtime. This section documents performance tests of AXS catalog partitioning. These numbers are intended to be only informational as users are not expected to need to do this on their own, or at least not often.

Table2.7shows the times needed for partitioning various catalogs (in minutes) and the size of the partitioned and compressed Parquet files on disk (in GB), depending on the number of zones used, while using the fixed number of buckets (500, which is the default in AXS). 10800 is the default number of zones in AXS (corresponding to the zone height of one arc-minute), the results for which are shown in the middle columns. The results on the left and right correspond to half and double the number of zones. 28 Spark executors were used for all the tests. Compressed size of partitioned catalogs increases with the number of zones because of increased data duplication, as was explained in section2.2.2.6.

Table2.8shows the same tests, but this time varying the number of buckets used, while keeping the number of zones constant at the default of 10800.

As can be seen from the results, data preparation times depend the most on the total size of the data. Increasing the number of zones also increases the time required to partition the data, while number of buckets has no influence on time required.

2.4.3 *Cross-matching performance depending on the number of zones and buckets*

The effect of number of zones and number of buckets on AXS' cross-matching performance was also investigated. The table2.9shows cross-matching performance results when using

Execs.	ZTF-SDSS		Gaia-ZTF		AllWISE-SDSS	
	warm	cold	warm	cold	warm	cold
1	821	2142	651	2258	431	1141
2	406	1067	323	1241	213	618
4	210	654	163	780	110	396
8	121	449	108	573	68	272
12	102	362	101	481	78	204
16	133	299	95	407	74	155
20	114	277	80	368	67	149
24	90	238	52	327	50	132
28	72	234	56	318	37	123

Table 2.6: Averaged raw cross-match performance results (in seconds), when returning only the first nearest neighbor, for the last three catalog combinations, depending on the number of executors and whether cold or warm OS cache was used.

Catalog	5400 zones		10800 zones		21600 zones	
	size	time (min)	size	time (min)	size	time (min)
SDSS	66	12	71	12	82	12
Gaia	430	89	464	86	532	150
Allwise	352	120	384	119	444	133
ZTF	1124	547	1169	545	1334	523

Table 2.7: Data partitioning: size of the partitioned catalogs (in GB) and time needed to partition the data (in minutes) depending on the number of zones used. All tests shown here used 500 buckets for partitioning data.

different numbers of zones while keeping number of buckets fixed at the default value of 500. Table2.10shows the same when using different numbers of buckets while keeping the number of zones fixed at the default value of 10800. Values in the middle columns in both tables are the same as those in tables2.3and2.4(because those tests used the default values for both the number of zones and the number of buckets). All tests in this section were done using 28 executors and with the queries returning all matching results.

Catalog	250 buckets		500 buckets		1000 buckets	
	size	time (min)	size	time (min)	size	time (min)
SDSS	71	12	71	12	72	10
Gaia	464	88	464	86	464	86
Allwise	384	125	384	119	384	116
ZTF	1169	557	1169	545	1169	514

Table 2.8: Data partitioning: size of the partitioned catalogs (in GB) and time needed to partition the data (in minutes) depending on the number of buckets used. All tests shown here used 10800 zones for partitioning data.

Catalog	5400 zones		10800 zones		21600 zones	
	warm	cold	warm	cold	warm	cold
G - A	32	207	31	226	36	240
G - S	33	128	37	148	36	151
Z - A	47	260	38	296	39	283
Z - S	48	209	47	239	49	227
G - Z	37	271	39	315	44	326
A - S	27	114	29	122	29	130

Table 2.9: Cross-matching duration (in seconds) depending on the number of zones and whether cold or warm OS cache was used while keeping number of buckets fixed at 500 (the default), for each catalog combination (denoted by their first letters), using 28 executors and returning all results.

Catalog	250 buckets		500 buckets		750 buckets	
	warm	cold	warm	cold	warm	cold
G - A	32	211	31	226	32	2314
G - S	33	146	37	148	37	159
Z - A	37	292	38	296	36	289
Z - S	47	237	47	239	47	234
G - Z	39	323	39	315	40	313
A - S	28	119	20	122	28	132

Table 2.10: Cross-matching duration (in seconds) depending on the number of buckets and whether cold or warm OS cache was used, while keeping number of zones fixed at 10800 (the default), for each catalog combination (denoted by their first letters), using 28 executors and returning all results.

2.4.4 *AXS performance in a cloud environment*

AXS' scalability was further tested in a "cloud environment", consisting of a Kubernetes cluster running in Amazon Web Services (AWS) virtual machines and with data stored in Amazon Simple Storage Solution (Amazon S3). The setup and results are described in [46] and summarized in this section.

Kubernetes[8] is an "open-source system for automating deployment, scaling, and management of containerized applications". Containers are an isolation mechanism that allows processes on the same operating system to run with isolated resources and Kubernetes orchestrates starting, stopping, monitoring, scaling, securing and self-healing of applications running inside containers.

Apache Spark supports running its processes inside a Kubernetes cluster, which is a functionality that AXS naturally inherits. Spark provides a script for creating Docker container images that can be used for starting Spark inside Kubernetes.

Kubernetes itself is cloud-agnostic, i.e. it can be run on any cloud provider. In this case it was run on Amazon cloud, but it would work equally well on Google Cloud Platform or Microsoft Azure. Cloud providers offer essentially limitless scalability (in practical terms)

[8] The Kubernetes documentation is available here: https://kubernetes.io/docs/

in CPU and memory resources.

In [46], AXS' "strong" and "weak" scalability were tested using a simple query (doing a sum of the *RA* column) on the ZTF light curve catalog containing $9x10^9$ rows. *Strong scalability* shows how much a query can be sped up by increasing the amount of processing power (number of processor cores) while keeping constant the amount of data being processed. *Weak scalability* shows how processing time changes when both the processing power and amount of data increase. The figure2.6shows the scalability measurement results calculated as *speedup*, for strong scaling, and *scaled speedup*, for weak scaling. *Speedup* is defined as t_{ref}/t_N, where t_{ref} is the query execution time performed with the reference number of cores (virtual CPUs, or vCPUs) and t_N is the query execution time performed with N cores. *Scaled speedup* is defined as $t_{ref}/t_N x P_N/P_{ref}$, meaning that execution query times are scaled by the problem size P_N with respect to the reference problem size P_{ref}. The problem size was scaled proportionally to the number of cores: a query running with 10 cores had to handle the problem size 10 times larger than the problem size used for a 1-core query.

When the reference number of cores was set to one (sequential computing), abnormally high speedups were observed. By adjusting the reference number of cores to 16, more realistic scaling was seen: weak scalability was shown to be linear while strong scalability shows a slow-down with an increase of processing power, which was to be expected, given that with more processes, more computing power needs to be spent on inter-process communication, similarly to what can be seen in Figure2.5.

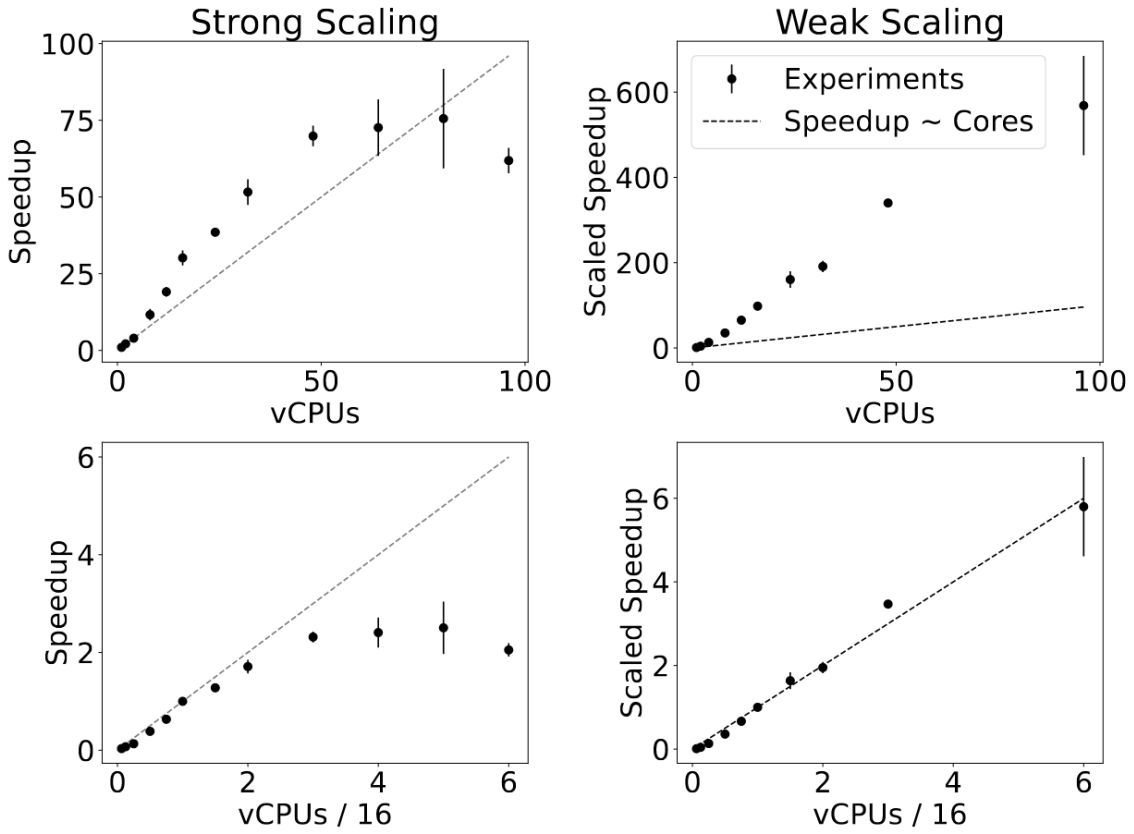

Figure 2.6: Figure 6 from [46]. The figure shows the strong scaling speedup (left column) and weak scaling scaled speedup (right column) observed when running a simple AXS query summing a column of the ZTF catalog containing $3x10^9 rows$. Strong scaling means increasing query speeds by increasing processing power (processor cores) and keeping the amount of data constant. Weak scaling means proportionally increasing both processing power and amount of data. Speedup is calculated as query execution time divided by reference query execution time. Scaled speedup additionally scales the speedup by inverse of the factor of data increase. The reference query for tests in the upper row used a single core, but with such sequential computing as reference, abnormally high speedups were observed. With the reference point set to 16 vCPUs (lower row), more realistic speedups are observed for both strong and weak scaling: weak scaling is linear and strong scaling diminishes for more vCPUs, which was expected.

3

Parameter Estimation of a Moving Point Source Model

3.1 Related work on the multifit algorithm

Image coaddition, a method of increasing signal-to-noise ratio, was described back in section1.2. It was noted there that image coaddition cannot help with estimating properties of moving point sources because such coadds end up containing blurred trails.

Multifit algorithm is a method that preserves information from all individual images of an object and uses that information to increase signal-to-noise ratio without depending on coadds. It does this by using *forward modeling*, i.e. it creates synthetic images, based on the presumed model of the target object, and compares the generated images with the obtained ones to estimate how well the model fits the reality. An optimization method drives the process of changing model parameters so that the difference between the generated and real images is minimized thus obtaining the optimal parameter values.

This "multi-epoch image processing" was called *Multifit* for the first time in [2]. There, the Multifit method was used for finding a single model per object (star or galaxy) based on a set of individual epoch images, each convolved with its own PSF. Several advantages of Multifit were noted [2]:

- If PSF is wrongly estimated for an object in an epoch image, that error will behave as a random error for that particular object and not as a systematic error for all objects on an image (which is the case if objects are modeled from a coadd where spatial correlations are present).
- Pixel interpolation of original images, which can cause noise correlations in coadds, is avoided here altogether.
- There are no fundamental limitations because of blended objects or if background noise is not dominant.
- Accounting for artifacts (cosmic rays, etc.) present in epoch images is more elegant and is taken care of at the model level.
- Finally, and most importantly for this thesis, there are no limitations when applying the method to measuring properties of moving sources.

3.1.1 *Evolution of the Multifit idea*

↬ LENSFIT.Lensfit was the first name used for the Multifit approach, suggested in [47] as a method for estimating ellipsicities of galaxies. The authors found a way to analytically marginalize galaxies' surface brightness and positional parameters. For the rest of the galaxy model parameters they create a 3D mesh of discretized parameter value combinations and for each point in the mesh they simulate a 2D brightness model. The resulting models are transformed using Fourier transformation, multiplied by Fourier transformation of the PSF model and finally compared with an image of a galaxy. The model with the lowest computed χ^2 value has the most likely ellipsicity values.

↬ "MEASURING THE UNDETECTABLE".In [3] the authors demonstrate how Multifit can be used for measuring parallaxes and proper motions of sources that are too faint for detection on individual epoch images. They use coadds for providing a "first-guess" positions of sources. Each epoch image needs to have a "reasonable photometric calibration, noise estimate in each pixel and correct astrometric calibration". They fit three types of models, which predict "every pixel value in every image at every epoch". The three types of models are:

- Moving point source
- Extended galaxy
- General transient or artifact

The different χ^2 values enable hypothesis tests, which give the most likely model. The best-fitting model's parameters constitute the source measurement results. The moving point source model (the one authors are interested in measuring) consists of six features: flux, position (RA, Dec), parallax and a proper motion (in two dimensions).

↬ LSST PROJECT.Running Multifit was also planned on LSST project as part of its pipeline. Fitting four types of models was foreseen [12]:

- Slowly Moving Point Source Model - similar to the model from [3], described above
- Small Object Model - intended for measurement of small galaxies
- Large Object Model - intended for identifying large galaxies
- and Solar System Model - intended for comets and near-Earth objects

↬ JAMES BOSCH.In his doctoral thesis [48], James Bosch investigates which galaxy models to use and how to efficiently characterize marginal posterior probabilities of their parameters. He recommends a model based on "multi-scale elliptical shapelet" functions and usage of "adaptive/iterative importance sampling algorithm", a version of a MCMC algorithm.

LENSFIT IMPLEMENTATION. The first implementation of the Lensfit algorithm is described in [49] for the CFHTLenS (Canada-France-Hawai Telescope Lensing) project. A few innovations are introduced compared to the original algorithm: a more complex galaxy model is used, PSF function is modeled at pixel level and an "adaptive" algorithm is used for sampling from the probability distribution, where a mesh of the possible values changes depending on the current algorithm step.

DARK ENERGY SURVEY. Dark Energy Survey uses Multifit [50] for galaxy shape and shear estimation. They also base their approach on Lensfit from [47] and call their algorithm implementation `NGMIX`.

3.2 MULTIFIT USING FREQUENTIST AND BAYESIAN STATISTICS

There are two main, competing views of statistical inference: frequentist and Bayesian (a great overview can be found in [51]). Frequentist paradigm argues that probability of an event can be objectively estimated by repeating an experiment enough times and calculating the frequency at which the event occurs. Any parameter that influences the event's probability is a fixed constant and attaching probabilities to it makes no sense.

Bayesian statistics, on the other hand, allows for assignment of probabilities to models and their parameters. It does so by producing their probability distributions from which all other statistical results can be calculated, such as point estimates and confidence intervals. The "philosophy" of the Bayesian approach is to describe our (incomplete) knowledge that we have about the world as faithfully as possible, and to quantify how the observed evidence (the data) changes that knowledge and fits into it.

In this chapter, the two approaches are described in more detail, with special focus given to their application to the Multifit problem. First, the data used for inference is described.

3.2.1 *Input data to the Multifit estimation process*

It is assumed that the data listed below are available at the start of the Multifit estimation process. The problem could be set up differently (depending on the environment, exact process of obtaining the data and so on) and this setup does not fundamentally change the way the algorithm functions, as it can be adapted to different models and parameters. But the setup does change the implementation details on which the subsequent sections depend. So, the problem will be confined to the environments having these data at the outset (these are typically produced by astronomical surveys' data processing pipelines):

- a set of images $I_1 \dots I_N$ of the observed object obtained at moments $t_1 \dots t_N$ and the corresponding set of "variance images" $V_1 \dots V_N$
- estimated PSFs $1 \dots N$, for each of the input images
- object's roughly-estimated position x_E and y_E (estimated from a coadd, with an implicit assumption of a static object)
- object's estimated magnitude M_E

Variance images and estimated PSFs are usually produced by data processing pipelines of large astronomical surveys (e.g. from Vera C. Rubin observatory). Variance images ($V_1 \dots V_N$) are maps that show the statistical uncertainty associated with each pixel in an astronomical image. The estimated PSFs can be expressed either as approximative 2D functions (e.g. of 2D Gaussians) or as normalized images (the sum of whose pixels is equal to 1). The latter is the case with Vera C. Rubin data processing pipeline.

3.2.2 *Model of a moving point source*

Apparent movement of stars ("apparent" meaning "as viewed from the Earth") consists of two main components: the *proper motion* of the star (i.e. its real movement in space) and the *parallax effect*, caused by the Earth's movement around the Sun, which makes objects that are close enough appear to be moving in the opposite direction. Asteroids, however, move much faster than stars, and their observation periods are often short, and so their parallax effect is often negligible. Since estimating asteroid movement will probably comprise the bulk of application of the work presented in this thesis, and to preserve simplicity, the model of a moving point source used here will omit the parallax component. However, nothing fundamentally restricts the Multifit method (both "batch" and "online") in this regard: it can estimate any kind of model parameters.

Expressing a star's or asteroid's movement is then very simple. If object's images were taken at moments $t_1 \dots t_N$, and if x_S and y_S are object's starting position at time $t = 0$ (which corresponds to image I_1), object's apparent motion v_x and v_y in x (RA) and y (Dec) directions can be expressed as:

$$x_i = x_S + v_x t_i$$
$$y_i = y_S + v_y t_i$$

However, since the starting position has already been roughly estimated by the data processing pipeline (see3.2.1), though with an error due to the fact that object's motion has not been taken into account, a good approach would be to allow for correction to this preliminary starting position estimate by expressing the starting position as comprising of the starting estimate (x_E, y_E) and a *position offset* (x_0, y_0; which is to be estimated):

$$x_i = x_E + x_0 + v_x t_i \tag{3.1}$$

$$y_i = y_E + y_0 + v_y t_i \tag{3.2}$$

The object's brightness can be expressed as "magnitude", which is a logarithmic measure with lower values for brighter objects. It is often used as a measure of brightness in astronomy.

To summarize, five parameters comprise the "moving point source model" and need to be estimated:

- M – object's magnitude
- x_0 – object's starting position offset in the *RA* direction (in arc-seconds)
- y_0 – object's starting position offset in the *Dec* direction (in arc-seconds)

- v_x – object's speed in the *RA* direction (in arc-seconds per day)
- v_y – object's speed in the *Dec* direction (in arc-seconds per day)

3.2.3 *The likelihood function*

Common to both frequentist and Bayesian statistics is that both make use of the likelihood function, with Bayesian statistics fundamentally depending on it.

The likelihood function gives the probability of the data given the model. It incorporates the knowledge of how the data get generated (e.g. what kind of images a point source of a certain brightness produces when observed through a telescope of a certain type) and is able to compare the actually observed and the modeled data. The likelihood function should have the highest value when the model (determined by its parameters) the most closely corresponds to the observed data. Likelihood function often has very small values, so in practice its logarithm is often used instead, giving the *"log-likelihood function"*, whose outputs are easier to handle by a computer.

↬ A SIMPLE EXAMPLE – A 2D LINE. An example of fitting a line will be used throughout the chapter for easy understanding of the concepts presented. A line in a 2D space is described by its slope (a) and intercept (b) parameters, which gives this model:

$$y = ax + b \tag{3.3}$$

For a fixed set of parameters a and b and an input vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ a noisy data set can be created by adding some random noise ϵ (vectors are denoted in **bold** letters):

$$\boldsymbol{y} = a\boldsymbol{x} + b + \boldsymbol{\epsilon} \tag{3.4}$$

If we assume a known Gaussian noise, the likelihood of observing value y, given a "true" value y_T is:

$$\mathcal{N}(y|y_T, \sigma^2) = (2\pi\sigma^2)^{-1/2} exp(-\frac{1}{2}\frac{(y - y_T)^2}{\sigma^2}) \tag{3.5}$$

With the assumption of independently sampled data points $\{y_n\}$, the full likelihood is then a product of their individual likelihoods. Since the "true" values are determined by line parameters ($y_T = ax + b$), the likelihood can also be written as:

$$\mathcal{L} = p(\{y_n\}|a, b, \{x_n\}, \{\sigma_n\}) = \prod_n^N p(y_n|a, b, x_n, \sigma_n) \tag{3.6}$$

The full set of model parameters (a and b in this case) is often denoted as θ and the likelihood function is often written as:

$$\mathcal{L} = f(\theta, \boldsymbol{y}) \tag{3.7}$$

The function f above contains in itself the knowledge of the model, i.e. how to produce the predictions based on model parameters θ, and how to compare them to the input data $\boldsymbol{y}$.

The log-likelihood can then be calculated as:

$$ln\mathcal{L} = \sum_n^N ln[(2\pi\sigma_n^2)^{1/2} exp(-\frac{1}{2}\frac{(y_n - (ax_n + b))^2}{\sigma_n^2})] \tag{3.8}$$

$$= -\frac{N}{2} ln(2\pi) - \frac{1}{2}\sum_n^N [\frac{(y_n - (ax_n + b))^2}{\sigma_n^2} + ln\sigma_n^2] \tag{3.9}$$

$$= const. - \frac{1}{2}\sum_n^N [\frac{(y_n - (ax_n + b))^2}{\sigma_n^2}] \tag{3.10}$$

The last equation stands because noise variances are known and can be regarded as being constant. So, to maximize the likelihood, and thus obtain the optimal parameters a and b, it is sufficient to minimize the well-known χ^2 statistic (using the more common notation $\hat{y_n}$ for the "predicted" value $ax_n + b$):

$$\chi^2 = \sum_n^N (\frac{y_n - \hat{y_n}}{\sigma_n})^2 \tag{3.11}$$

Or in the case of homoscedastic noise (where noise variance in each point is the same), the sufficient statistic to be minimized is the familiar sum of squares of differences between observed and predicted values:

$$S_s = \sum_n^N (y_n - \hat{y_n})^2 \tag{3.12}$$

↬ THE LIKELIHOOD FUNCTION FOR THE MOVING POINT SOURCE MODEL.In the case of a moving point source, the data are not points in a 2D space, but images, consisting of pixels. Each pixel is treated as a single data point and each pixel's value is determined by the number of photons that hit the surface of the photo-sensitive chip during the exposition time.

Probabilities of numbers of events that occur during a specific period of time (number of photons in this case) follow the Poisson distribution, which is determined by a single parameter: rate λ. Thus, a model-determined image $\hat{I}$ with Poisson noise can be written as:

$$\hat{I}_{x,y} \sim Poisson(\lambda = M(x, y)) \tag{3.13}$$

Poisson distribution can be safely approximated by a Gaussian when number of events is large (> 1000), which is usually true for pixels of astronomical images, and then it has the neat property that its mean μ and variance σ^2 are both equal to λ. So, the previous statement can be rewritten as:

$$\hat{I}_{x,y} \sim \mathcal{N}(\mu = M(x, y), \sigma = \sqrt{M(x, y)}) \tag{3.14}$$

This leads to the analogous log-likelihood function as in the previous section (3.11) and to the analogous χ^2 function (with that difference that the data points here are indexed with two indices (x, y), instead of with n):

$$\chi^2 = \sum_{x,y} \frac{(I_{x,y} - \hat{I}_{x,y})^2}{2\sigma_{x,y}^2} \tag{3.15}$$

As was already explained in3.2.1, the variances of each pixel are available as "variance images" $V_{x,y}$, , and so the χ^2 can be calculated with the original, observed image $I_{x,y}$, the model-generated image $\hat{I}_{x,y} = M(x, y)$ and the variance image $V_{x,y}$:

$$\chi^2 = \sum_{x,y} \frac{(I_{x,y} - M(x, y))^2}{2V_{x,y}} \tag{3.16}$$

↬ TESTING FOR GOODNESS OF FIT WITH χ^2_{DoF}.The χ^2 values are distributed as the χ^2 distribution with k degrees of freedom, where k is equal to the number of "free" data points: number of data points minus the number of model parameters, $N - N_{par}$. This distribution does not depend on model parameter values, but only on k, i.e. the number of data points N.

For larger values of k, the χ^2 distribution can be well approximated by a Gaussian distribution with the mean of k and standard deviation of $\sqrt{2k}$:

$$p(\chi^2|k) \sim \mathcal{N}(\mu = k, \sigma = \sqrt{2k}) \tag{3.17}$$

Dividing chi^2 by k gives the "χ^2 per degrees of freedom" value χ^2_{DoF} which is distributed as:

$$p(\chi^2_{DoF}) \sim \mathcal{N}(\mu = 1, \sigma = \sqrt{\frac{2}{N - N_{par}}}) \tag{3.18}$$

If data is coming from a distribution corresponding to the one assumed by the model (and the likelihood function), the χ^2_{DoF} value should be close to 1 plus or minus several $\sqrt{2/(N - N_{par})}$.

3.2.4 *Multifit using the frequentist approach*

For finding the "point estimate", i.e. the optimal model parameters, based on the designed likelihood function, the frequentist statistics prescribes the "maximum likelihood" approach, i.e. maximizing the likelihood function. The likelihood function can be maximized either analytically (if that is feasible), or an optimization procedure can be used to repeatedly change the model's parameters in such a way as to arrive to the maximum value of the likelihood function and thus "fit the model to the data".

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta, \boldsymbol{y}) \tag{3.19}$$

As was already stated, minimizing χ^2 has the same effect:

$$\hat{\theta} = \arg\min_{\theta \in \Theta} \chi^2(\theta, \boldsymbol{y}) \tag{3.20}$$

So, "solving Multifit" using the frequentist approach, with the model defined in the previous sections, means using an optimization procedure to find the optimal values of magnitude M, starting position offset (x_0, y_0) and object's speed (v_x, v_y) parameters that maximize the defined likelihood.

↬ OPTIMIZATION PROCEDURES.Sum of squares of residuals (differences between observed and predicted values) was shown in section3.2.3.1to be the sufficient statistic for a model with Gaussian errors. Minimizing this statistic can be accomplished with many different methods and techniques.

The most important procedures include gradient descent, Newton's optimization method [52], Gauss-Newton algorithm [53], Levenberg-Marquardt method [54], Trust Region Reflective or TRF [55], Dogbox [56], Powell [57], Nelder-Mead [58], BFGS [52], and others.

Implementations of these methods are available in Python packages `scipy` and `lmfit` and were tried out on the Multifit problem in this thesis.

Obtaining error estimates are different with different packages used. The implementations in the `lmfit` package return covariances directly and the error estimates can be found on the diagonal of the covariance matrix. Implementations from the `scipy` package (Levenberg-Margquardt, TRF and Dogbox) return the Jacobian matrix J (matrix of all first-order derivatives of the target function). The Hessian matrix H can be approximated from the Jacobian as $H = 2J^T J$. Inverting a Hessian then gives the covariance matrix with the estimated errors on its diagonal.

↬ JACKKNIFE METHOD OF ERROR ESTIMATION.Jackknife is a method of error (variance) estimation that calculates an estimate $\hat{x}_i$ of a parameter x N times always leaving out one of the data points from the data set. The method then averages out these N estimates to obtain the "Jackknife estimate" $\hat{x}_J$. Un unbiased estimate of the variance of the main estimation procedure can then be obtained with:

$$\hat{var}(\hat{x}) = \frac{N-1}{N} \sum_{i=1}^{N} (\hat{x}_i - \hat{x}_J) \tag{3.21}$$

However, if the original estimation procedure is itself expensive and there are lots of data points, Jackknife method is not practical for real usage.

3.2.5 *Bayesian statistics*

The main novelty in Bayesian statistics, compared to "classical statistics", is that probabilities are not only attached to data but also to models and their parameters (again, for a good pedagogical introduction see [51]). What is common to classical statistics and Bayes method is the usage of the likelihood function, but unlike in Bayesian statistics, in classical statistics the likelihood function cannot be interpreted as a probability density function of model parameters because that notion does not even exist there: model parameters are not random variables.

In Bayes statistics, the relationship of original knowledge, data and that knowledge which is learned from both is described by the Bayes Theorem, given with the following formula.

$$p(M, \theta | D, I) = \frac{p(D|M, \theta, I) p(M, \theta | I)}{p(D|I)} \tag{3.22}$$

Here M designates the model, θ is a set of parameters that determine the model and D are the data. The expression $p(D|M, \theta, I)$ is the likelihood of the data, or in other words,

the probability that we will see the observed data if their generation is described by model M with parameters θ. As was already said in section3.2.3, the likelihood comes from design of the experiment, or knowledge that we have about the problem at hand.

$p(M, \theta|I)$ is the "prior", or the current knowledge that we have about the possible model parameter values. Prior is something that does not exist in the classical statistics and is based on subjective judgment (which is one of the arguments against Bayesian statistics in the frequentist–Bayesian debate).

The expression in the denominator is called "evidence" and represents the raw probability of the data. Calculating evidence is computationally the most problematic part of Bayesian inference. However, it is almost never calculated directly but obtained by normalizing the numerator.

I is the "background information" and is explicitly shown in all probabilities in3.22in order to emphasize that the inference is not "suspended in vacuum", nor is it given us "from above", but is always performed in the context of knowledge and assumptions that we have about the problem (which can be incorrect, and which can render the whole procedure incorrect).

Finally, the result on the left-hand side, $p(M, \theta|D, I)$, is the "posterior" probability of model parameters and represents the new knowledge which is a combination of the prior, the model and the data.

3.2.6 *Posterior calculation*

Calculating posterior using the Bayes formula3.22requires calculating the evidence integral:

$$p(D) = \int p(D|\theta)p(\theta)d\theta \tag{3.23}$$

However, calculating that integral analytically is often impossible. That problem was often solved in history using *conjugate priors* which, in combination with the likelihood function (which also had to be of a simpler form), give a posterior of the same analytical form. If that is not possible, one would use approximative functions of the same form.

With the development of computer technology, the problem became solvable with approximative numerical methods. The first such method, applicable only in the case of a smaller number of parameters, a "grid" is constructed with different combinations of parameter values and the integral is calculated at points on the grid (an example of application of this method is "lensfit" mentioned in3.1.1). However, even a modest increase in the number of parameters leads to a combinatorial explosion and exponential increase in computing power required.

Another type of numerical approximations, which is used in this thesis, are Markov chain Monte Carlo algorithms which have seen wide adoption and development. They give posterior distributions of model parameters without calculating the integral3.23. We give their brief description in the next section.

3.2.7 *Markov chain Monte Carlo*

Monte Carlo methods are used today in almost all sciences for simulating complex systems and evaluating multidimensional integrals. In [59] a definition is proposed according to

which "Monte Carlo" means "using random numbers for estimating a value which itself is not random".

Monte Carlo integration, which is very important in Bayesian statistics, can be described in the following way ([60]). If $f(x)$ is a probability distribution defined on the interval from a to b and we randomly generate N numbers X uniformly distributed in the same interval, then the integral of the function $f(x)$ can be estimated using this formula:

$$\int_a^b f(x)dx \simeq \frac{1}{N}\sum_i f(X_i) \tag{3.24}$$

However, if the function $f(x)$ varies significantly, which is often the case with higher number of dimensions (i.e. x is a vector), Monte Carlo is very inefficient [51] because it spends too much time in the areas where the function contributes very little to the total value. If x would be sampled directly from distribution $f(x)$, algorithm would be more efficient (it would spend more time calculating areas where $f(x)$ value is higher). This is known as *importance sampling*, but designing such an algorithm when one needs to handle a large number of dimensions is difficult [61].

A much more efficient group of methods is Markov chain Monte Carlo (or MCMC for short), where random values are generated by the method of random walk where each new step depends only on the current state (which is the definition of a Markov chain) and whose distribution asymptotically approaches $f(x)$. With this group of methods, one needs to define a transition matrix $P\{p_{ij}\}$ over a set of states S, in relation to the probability distribution f, which satisfies these two conditions [62]:

- *Irreducibility* For any two elements i and j from S it is possible to go from i to j using transition matrix in a finite number of steps
- *Stationarity* For every $j \in S$ the following holds: $\sum_i f_i p_{ij} = f_j$

The sufficient condition for stationarity is the condition of "detailed balance", which is easier to check: for every $x, y \in S$, $f_i p_{ij} = f_j p_{ji}$.

When these conditions are satisfied, the distribution f is called *stationary* or *equilibrium* distribution.

↬ CONVERGENCE TO EQUILIBRIUM.It can be shown that Markov chains satisfying these conditions inevitably converge to the equilibrium distribution regardless of the distribution they were initialized with. Sampling from a chain that has not reached the equilibrium state introduces statistical errors. So, in practice, chain histories from before the equilibrium state was reached (the so-called "burn-in") are discarded. However, determining exactly when the chain reaches equilibrium is hard and can demand additional computing resources. The number of steps needed to reach stationarity, though, can usually be empirically determined by visually inspecting chain histories.

↬ METROPOLIS-HASTINGS ALGORITHM.There exist many versions of MCMC algorithms for generating Markov chains that satisfy conditions from3.2.7. The most famous one is the Metropolis-Hastings algorithm [63] where the next sample θ_{t+1} in Markov chain

for the current state θ_t is selected in two steps. First, a suggested value γ is sampled from a *proposal distribution* $q(\gamma|\theta_t)$ (which can be of any shape, such as multivariate Gaussian distribution), and the *Metropolis ratio* r is calculated (p is the posterior):

$$r = \frac{p(\gamma|D,I)}{p(\theta_t|D,I)} \frac{q(\theta_t|\gamma)}{q(\gamma|\theta_t)} \tag{3.25}$$

The second factor will be equal to 1 for symmetrical proposal distributions. As was already mentioned, calculation of evidence integral is problematic, but in ratio r it is canceled out. One only needs to calculate priors and the likelihood, and these functions are readily available. If the calculated ratio $r \geq 1$ (the proposed value γ is more probable than the current value θ_t), the proposal is immediately accepted and θ_{t+1} becomes γ. If $r < 1$, the proposed value is accepted with probability r. Otherwise, θ_{t+1} remains equal to θ_t.

It can be shown [64] that Metropolis-Hastings algorithm always (after a sufficient number of steps) ends up faithfully describing the target probability distribution.

↬ AFFINE INVARIANT ENSEMBLE MCMC METHOD.In [4] the authors propose an ensemble MCMC method invariant to affine transformations. For an algorithm to be "affine invariant" means that its efficiency is not dependent on the level of asymmetry of the target distribution (if it's significantly skewed in one or several of the dimensions). The classical MCMC algorithms (Metropolis-Hastings and Gibbs) cannot use different moves in different dimensions and so they become inefficient (although they still give equally good estimate of the target distribution after a large number of steps).

The authors call a MCMC algorithm "affine invariant" if the following holds:

$$R(a\theta + b) = aR(\theta) + b \tag{3.26}$$

where R is the sampling function $\theta(t+1) = R(\theta(t))$.

The suggested method is an ensemble method because it uses a set of L "walkers" that explore the domain space based on the current position of all other walkers. One step of the ensemble MCMC chain comprises the full circle across all walkers. The authors suggest the "stretch move", "walk move" and "replacement move" as options for choosing the next positions of a walker in an affine-invariant manner.

↬ KDE MOVE.During the work on the Multifit implementations described in later sections it became apparent that the default "stretch move" tended to leave some chains in local minima and skew the later density estimation results. "DE move" and "KDE move", based on [65], were much more stable (and worked equally well) and enabled the procedure to reach statistical performance described in3.8.

↬ MCMC SAMPLING EFFICACY AND AUTOCORRELATION.More efficient MCMC algorithms will describe the target distribution in a smaller number of steps. Every time a newly proposed chain value is rejected, time and computing power need to be spent at repeated sampling and prior and likelihood calculations, so the so-called "acceptance rate" (of proposed values) is one of measures of efficacy of MCMC algorithms. None of the extreme values of this measure are not satisfactory. If the acceptance rate is 0%, the

algorithm will not advance at all. If it is 100% the algorithm will advance regardless of the target distribution and such a result would be useless. In [61] authors recommend to tune the acceptance rate to be between 25% and 50%.

Chain *autocorrelation* is correlation of the chain with itself at two different moments with a lag of k steps. Autocorrelation usually drops when k increases. *Effective sample size* is another measure of MCMC sampling efficiency and tells us what would be the number of samples in the chain that would have no autocorrelation but would contain the same amount of information [64]. The higher the length of a chain from its effective sample size is, the less efficient the sampling has been.

Reliably estimating autocorrelation of a chain, however, is computationally demanding and requires availability of a large number of samples, which can render the procedure not feasible in practice.

3.2.8 *Estimating the posterior distribution*

Result of a MCMC inference procedure is a history of MCMC chain values. As was seen in section3.2.7.2, after a number of steps large enough, the chains end up spending, at different parameter values, an amount of time which is proportional to values' posterior probability. In other words, the more samples there are inside an interval, the probability of that interval is larger.

One often wants to know parameter's marginal probability density, i.e. posterior probability density function of a parameter considering all the possible values of all other parameters by integrating over them. The MCMC algorithm gives an easy way to find such marginal probability densities: from all the N-dimensional sampled values one only needs to retain the specific parameter's dimension and discard all the others.

To obtain a mathematical description of this posterior one needs to analyze the histogram of chain samples. There are two main groups of techniques for describing distributions based on data samples. In the first group are summary statistic-estimation techniques, which give only a partial estimate of the target distribution, often with a single value. Point estimation, maximum likelihood estimation, and estimation of confidence intervals belong to this group. In the second group are the methods that estimate the full posterior distribution, such as histogram-based and kernel density estimation (KDE) techniques.

↬ SUMMARY STATISTICS.Estimating summary statistics of probability densities, based on a set of samples, results in simple estimates in the form of a single value (point estimates) or in the form of intervals of parameter values. They do not give the full picture of a probability density and can therefore be misleading. MAP approximation is similar to the maximum likelihood estimate, but also includes information from the prior. Another estimate that can be used for describing posterior distribution is mean estimate: $\bar{\theta} = \int \theta p(\theta|D) d\theta$. In [61] MAP, mean and median of posterior probability distributions are compared and the conclusion is that the median is the most robust option of the three.

Posterior distributions can also be described by estimating *credible regions*, i.e. ranges within which a certain percentage of possible values resides. When estimating credible regions, one wants to find values a and b such that $\int_{-\infty}^{a} f(\theta) d\theta = \int_{b}^{\infty} f(\theta) d\theta = \alpha/2$. Then

the probability that the true value of parameter θ is between a and b is equal to $1 - \alpha$, and that interval is called *1 - α posterior interval* [51].

↬ DENSITY ESTIMATION USING HISTOGRAMS.Estimating density of a target distribution using the histogram itself is done by grouping the values into discrete bins, counting number of samples in each bin, and normalizing thusly obtained function with the number of samples. The histogram-based estimation function, for n samples, where $h(x)$ determines the width of a bin, and N_m denotes number of samples in the bin m, can therefore be described with the following formula [66]:

$$f_{hist}(x) = \frac{N_m(x)}{nh(x)} \tag{3.27}$$

↬ KERNEL DENSITY ESTIMATION.A better method of estimating densities (than using histograms) is kernel density estimation (KDE) which converges faster to the true distribution [67]. Besides bin sizes, even bin placement is problematic when using histograms and can influence the estimate. That problem could be circumvented if every sample would belong to its dedicated bin and if bins would be allowed to overlap. Replacing each sample with a Gauss function and summing up all of their values results in a KDE with a *Gaussian kernel*. However, then the question of width of the Gaussian kernel comes up. If the kernel is too narrow, the result is a function with too much variance (noise). A kernel too wide averages the peaks too much and some of the information is lost.

Instead of Gaussian, any other function K can be used as kernel if it satisfies the following requirements: that its strictly positive ($K(X) \geq 0$), that it's normalized to 1 ($\int K(x)dx = 1$), and that its mean is equal to 0 ($\int xK(x)dx = 0$). The estimated function is then obtained using this formula:

$$\hat{f}(x) = \frac{1}{n}\frac{1}{h^D}\sum_{i=1}^{n} K\left(\frac{d(x, X_i)}{h}\right) \tag{3.28}$$

where h is width of the kernel (or "bandwidth"), X_i are the measured values, d is distance function, and D is dimensionality of the parameter space [51]. Other options for choosing a kernel are the "top-hat" kernel:

$$K(u) = \begin{cases} \frac{1}{V_D(1)} & \text{za } u \leq 1, \\ 0 & \text{za } u > 1 \end{cases} \tag{3.29}$$

And Epanechnikov kernel:

$$K(u) = \frac{3}{4}(1 - u^2) \tag{3.30}$$

for $-1 \leq u \leq 1$.

Although kernel density estimation can describe the target distribution faithfully, in practice however, evaluating it on new samples can be prohibitively slow.

↬ APPROXIMATING POSTERIORS WITH A GAUSSIAN MIXTURE.Another method for approximating distributions is by using Gaussian mixtures where A mixture of Gaussian models can be fit to data (histograms) using the expectation-maximization (EM) algorithm.

The input to the process is the assumed number of components in the mixture and random parameters for each component. The EM algorithm then computes for each sample the probability that it is generated by each of the components in the mixture (the "expectation" step). Then the total likelihood of the data is maximized by changing the parameters (the "maximize" step). This is repeated in iterations until the likelihood cannot be further improved.

Another option for fitting a Gaussian mixture to data is variational inference, a similarly iterative procedure, which does not just maximize the likelihood but also includes the prior information, namely the prior on distribution of component weights, usually represented with the Dirichlet distribution.

Variational inference is slower than classical EM algorithm, but it is able to set the weights of some of the components close to zero, which is not the case with EM. That means that the number of components does not need to be specified in advance as the procedure can determine it on its own. However, its "weights concentration prior" hyperparameter, which needs to be specified in advance, can influence the results significantly. So, the flexibility of variational inference comes with additional cost.

Outliers can influence both procedures and skew the final solution. The same is true with incomplete data, where part of the Gaussian curve is missing, for example.

Implementations of both of these procedures are available in the Python's `scipy` package. Variational inference (`BayesianGaussianMixture` class) is used in this thesis. More information in section3.6.8.

3.2.9 *Multifit using the Bayesian approach*

Finally, putting all this together, to fit an object's magnitude M, starting position offset (x_0, y_0) and object's speed (v_x, v_y), based on all the object's acquired images $I_1 \dots I_N$ (i.e. the Multifit approach) using Bayesian statistic, one would:

1. Choose a prior distribution for each parameter based on existing knowledge about the world and the model. For example, it might be known that detecting objects with magnitude of less than 18 or more than 28 is not possible with the equipment used or in the area being observed. And it might also be the fact that observing objects of any magnitude in that range is equally likely. Then, one would specify a uniform (or "flat") prior for the magnitude parameter in the range between 18 and 28.

2. Design an image generation procedure, based on the assumed model (forward modeling)

3. Design a likelihood function that would compare the generated and observed images, using the χ^2 statistic (as was previously described in section3.2.3.2).

4. Run a MCMC procedure that would explore the posterior distribution (combining both priors and the likelihood) and produce a set of samples from the posterior (chain values)

5. Estimate the posterior distribution using the generated chain values with one of the methods from section3.2.8

6. Use the estimated posterior distribution to obtain the required statistical properties for all the parameters

For details on how this can be implemented in practice please skip to section3.6.

3.3ONLINE MULTIFIT

The "standard" Multifit approach, whether using frequentist or Bayesian statistics, always considers all available images of an object at the same time. This could also be called the *batch* Multifit approach.

The novelty of the *online* Multifit is to consider images one by one. For the first image x_1, the procedure uses the original parameter priors ($\pi(\theta)$) and produces the first posterior:

$$\pi_1 = p(\theta|x_1) \propto p(x_1|\theta)\pi(\theta) \tag{3.31}$$

This posterior then becomes the prior for the next image x_2:

$$\pi_2 = p(\theta|x_2) \propto p(x_2|\theta)\pi_1(\theta) \tag{3.32}$$

and so on. For the N-th image we have:

$$p(\theta|x_N) \propto \pi(\theta) \prod_{i=1}^{N} p(x_i|\theta) \tag{3.33}$$

This procedure, also known as "sequential" (and also "recursive") "Bayesian updating", results in the same posterior as the batch procedure, if the posteriors in each step can be perfectly calculated.

3.3.1 *Fitting a 2D line with online Bayesian inference*

How online Bayesian inference works in practice will now be illustrated using the simple model of a 2D line presented back in section3.2.3.1. The rightmost graph in Figure3.1shows the example dataset. The true slope a and intercept b values of the example line are 0.5 and 1.5. The four points on the line were randomly chosen and Gaussian noise with $\sigma = 0.2$ was added to their y dimension. Batch fitting of a line through those points (using Numpy's `polyfit`) gives an estimate of $a = 0.54$ and $b = 1.48$.

Model and the log-likelihood function can be defined in Python like this:

```
def M(x, a, b):
  return a * x + b

def logL(x, y, a, b, sigma):
  return ((y - M(x, a, b)) / sigma)**2
```

Online approach dictates that each data point is processed sequentially one by one. If a range of acceptable a and b values is assumed (for example $[0, 2]$ for both parameters) and if each range is divided into a number of bins (for example 1000) and those bins are represented as pixels, the log-likelihood value can be calculated for each combination of the parameter values (i.e. bins, which makes 1 million points for this example). That produces

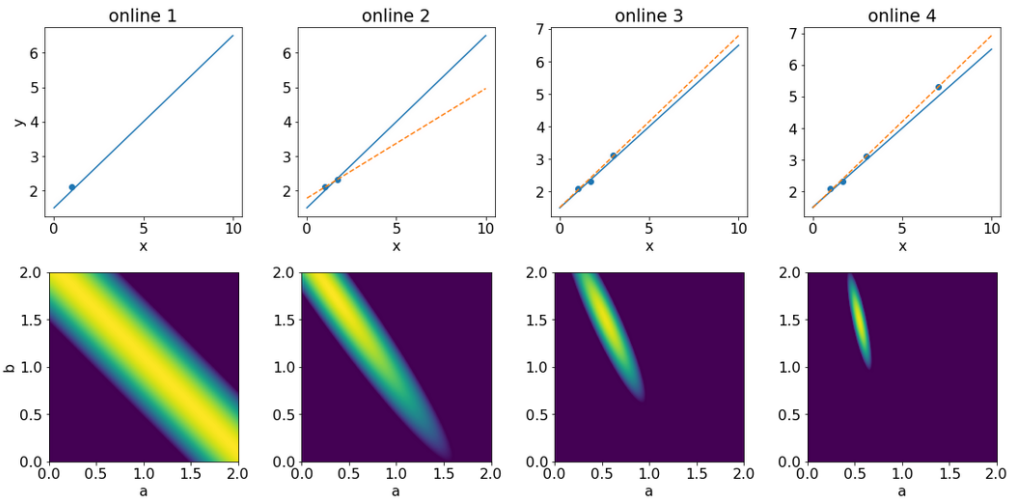

Figure 3.1: Fitting a line using a random dataset with online Bayesian inference. Each image in the lower row is the log-posterior after processing one additional point, plotted on a grid corresponding to different a and b values. Upper row shows the ground truth (blue line) and the best estimate (orange line) at each step. Precision increases with each new image.

a 2D matrix of log-likelihood values which can be plotted as an image. As was already explained in section3.2.3.1, a fitting choice for the log-likelihood function is the negative sum of squared differences between the predicted and data y values. The larger the difference, the smaller the likelihood of that combination of parameter values.

According to the Bayes theorem (equation3.22), log-prior values need to be added to the log-likelihood. In this example a flat prior will be used, or an image with pixels equal to zero (which means that all parameter values are equally likely at the onset), so the log-posterior after the first point is equal to the calculated log-likelihood image. This log-posterior is then used as a log-prior for the next data point, i.e. it is added to the next log-likelihood to produce the next log-posterior, and so on.

Using this method, posterior values can be directly calculated on a grid and there is no need for the full-fledged MCMC simulation. However, there is obviously a limit to the method as adding more parameters exponentially increases the grid size and, hence, computational cost. Besides, the accuracy of the method is limited by the granularity of the grid.

Figure3.1shows the four consecutive log-posteriors after processing each of the four data points (the values are clipped in the range $[-1, 0]$). The first posterior looks like a line. All combinations of a and b values on that line correspond to lines in $x - y$ space that all go through the first data point. After the second data point, the posterior is already centered at the correct a and b values, but with a relatively large uncertainty. The uncertainty decreases with each subsequent point.

After the last point, the a and b values corresponding to coordinates pointing to the largest log-likelihood value in the grid are 0.54 and 1.48, the same values as those estimated with the batch approach.

3.3.2 *Applying online Bayesian inference to Multifit*

The same principle is applied to the Multifit method in this thesis. The difference is that the data are images and not points in a 2D space and the model has more parameters, so calculating log-likelihoods and posteriors on a grid is not possible. Instead, posteriors can be estimated from MCMC histograms after each image and approximated with a mixture of Gaussians. This approximation will undoubtedly introduce some errors and the main challenge is to reduce those to the minimum. On the other hand, approximating the posteriors with a set of Gaussians compresses the information from sample histograms into means and covariances of the mixture components.

3.3.3 *Benefits of online Multifit and computational efficiency*

When the data used for Bayesian inference are images instead of points in 2D space, the advantages of the approach become more apparent: an online algorithm needs to process only a single image at each step, while the batch algorithm always needs to process all the available images.

Consider an astronomical catalog consisting of hundreds of detections (images) of each object in the catalog (potentially billions). Each night brings new images to the catalog and the measurements need to be updated with the new information. If the Multifit algorithm is used for estimating objects' measurements, the batch version would need to process all the available images for each detected object (some surveys will observe the same objects hundreds of times). The online version processes only the new images and updates the posteriors accordingly. When the next image arrives, only the mathematical representation of the posterior needs to be loaded to represent all the knowledge accumulated so far.

Let's assume the same number of MCMC chains W and the same number of iterations I is used for both *batch* and *online* algorithms. Let's also assume that all of N images have the same dimensions. Then the following observations can be made.

CPU EFFICIENCY.For each iteration I and each walker W both procedures will compare the available images with the simulated ones: N images for the *batch* version and 1 image for the *online* version. For each iteration both procedures also need to generate new proposals for new walker positions, but that step does not depend on the number of images and so it can be left out from the analysis. This gives $W \times I \times N$ image operations for the *batch* version and $W \times I$ image operations for the *online* version.

However, when the next image $N + 1$ is added and the new posterior needs to be calculated, the *batch* version will now need to perform $W \times I \times (N + 1)$ image operations while the *online* version stays at $W \times I$. So, in total, if N images are added one by one, and the new posterior is calculated every time, the *batch* version would perform $W \times I \times \frac{N(N+1)}{2}$ image operations (which is $O(N^2)$ complexity) while the *online* version performs only $W \times I \times N$ ($O(N)$ complexity).

In reality, astronomical catalogs are not updated after each object gets a single new observation, so the number of *batch* operations would need to be divided by some average number of observations that are processed in bulk, but the complexity still remains $O(N^2)$ for the *batch* version.

↬ MEMORY EFFICIENCY.The main observation stays the same when comparing memory consumption: the *batch* version needs to hold in memory all N images and the *online* version needs only the new image. That gives the memory complexity of $O(N)$ for the *batch* version and $O(1)$ for the *online* version.

The *online* version does not even have to store the images once they have been processed. However, it needs to store the estimated posterior data which consists of means and covariance matrices of Gaussian approximations, which is smaller in size than a single image.

3.4COMMON ELEMENTS OF THE MULTIFIT IMPLEMENTATIONS PRESENTED

Before describing various Multifit implementations developed in this thesis, this section describes two elements that are common to all of them: the simulated data and the likelihood function.

3.4.1 *Simulated data*

The data that were used for implementation and testing of the various Multifit implementations were based on the "RC2 subset" (available at `https://github.com/lsst/rc2_subset`) of the HSC RC2 dataset, which is "a collection of three tracts (in the GAMA, VVDS, and COSMOS fields) from the first public data release of the HSC SSP survey, used for regular testing of the LSST Data Release Production pipelines" (`https://github.com/lsst-dm/gen3-hsc-rc2`). The subset consists of the central 6 detectors for 8 randomly chosen exposures (exposing the telescope's focal plane to the light coming from a certain location in the sky) in the 5 broad band filters. So, the images that were used came from a real telescope.

However, the HSC RC2 dataset does not contain moving point sources that could be used for algorithm evaluation, so a set of fake moving sources had to be inserted into the existing images. The code used for generating fake sources was taken from the `insertFakes` LSST task [1]. The same, although slightly modified code was also used for implementing "forward modeling" in the likelihood function, as will be explained below.

The RC2 subset is covering the region between coordinates (149.7829, 1.9873) and (150.6144, 2.3215), but not all parts of the region have been observed the same number of times. The Figure3.2shows the coverage of the region, i.e. the number of overlapping exposures covering each pixel (with Dec coordinate placed on the x axis to better fit the page). With the pixel scale of the telescope roughly equal to 0.1686 arc-seconds per pixel, the full region spreads over an image $7138x17758$ pixels in size.

A small subregion of $502x574$ pixels in size with a good coverage was chosen for placing the simulated moving point sources into.

The exposures making up the chosen region cover a time range of 421 days (since 56741.4 until 57163.3 in MJD – Modified Julian Day, i.e. since 25 March 2014 until 21 May 2015).

[1] The source code can be found here: `https://github.com/lsst/pipe_tasks/blob/main/python/lsst/pipe/tasks/insertFakes.py` and some documentation here: `https://pipelines.lsst.io/v/daily/py-api/lsst.pipe.tasks.insertFakes.InsertFakesTask.html#lsst.pipe.tasks.insertFakes.InsertFakesTask`

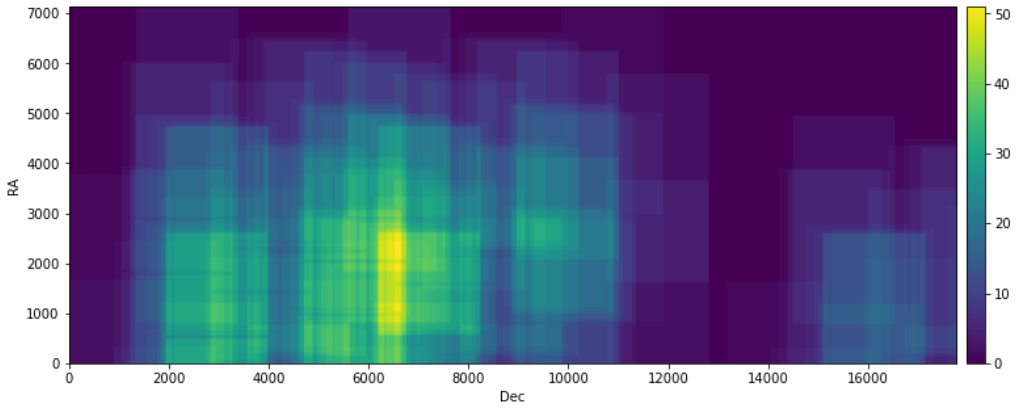

Figure 3.2: Number of observations for each pixel in the region covered by the RC2 subset

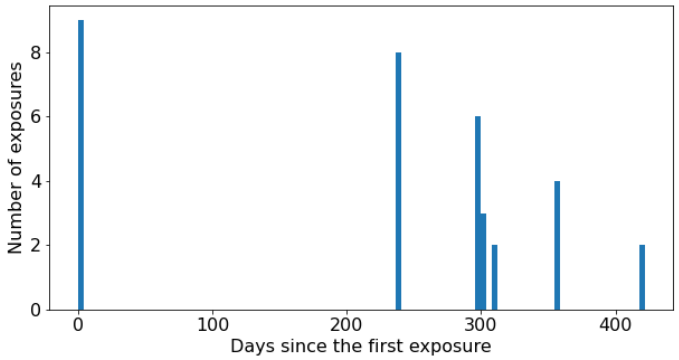

Figure 3.3: Distribution of times (in days) when exposures were made relative to the first exposure in the dataset.

Figure3.3shows time of each exposure in the dataset in days since the first exposure. The distribution of exposures is not ideal: instead of being spread out evenly over a period of time, exposures are grouped on only several specific days.

↬ PREPARING HSC DATA.Before inserting fake moving sources, the HSC data is prepared with the "single frame processing" task of the LSST pipelines (described here: `https://pipelines.lsst.io/v/weekly/getting-started/singleframe.html`). This step "removes instrumental signatures with dark, bias and flat field calibration images" and also "uses the reference catalog to establish a preliminary WCS and photometric zeropoint solution".

↬ SIMULATING MOVING POINT SOURCES.The chosen subregion was divided into 304 cutouts (16x19 – 16 in RA direction and 19 in Dec direction) of 30x30 pixels in size. At the center of each cutout a simulated object was placed with a random speed (except those in the leftmost column which all have the speed of zero in both directions). The speeds were chosen so that the objects travel at most 5 pixels during the observed period with most of them traveling about 1 pixel.

The Figure3.4shows speeds of the simulated objects. The lower two graphs show

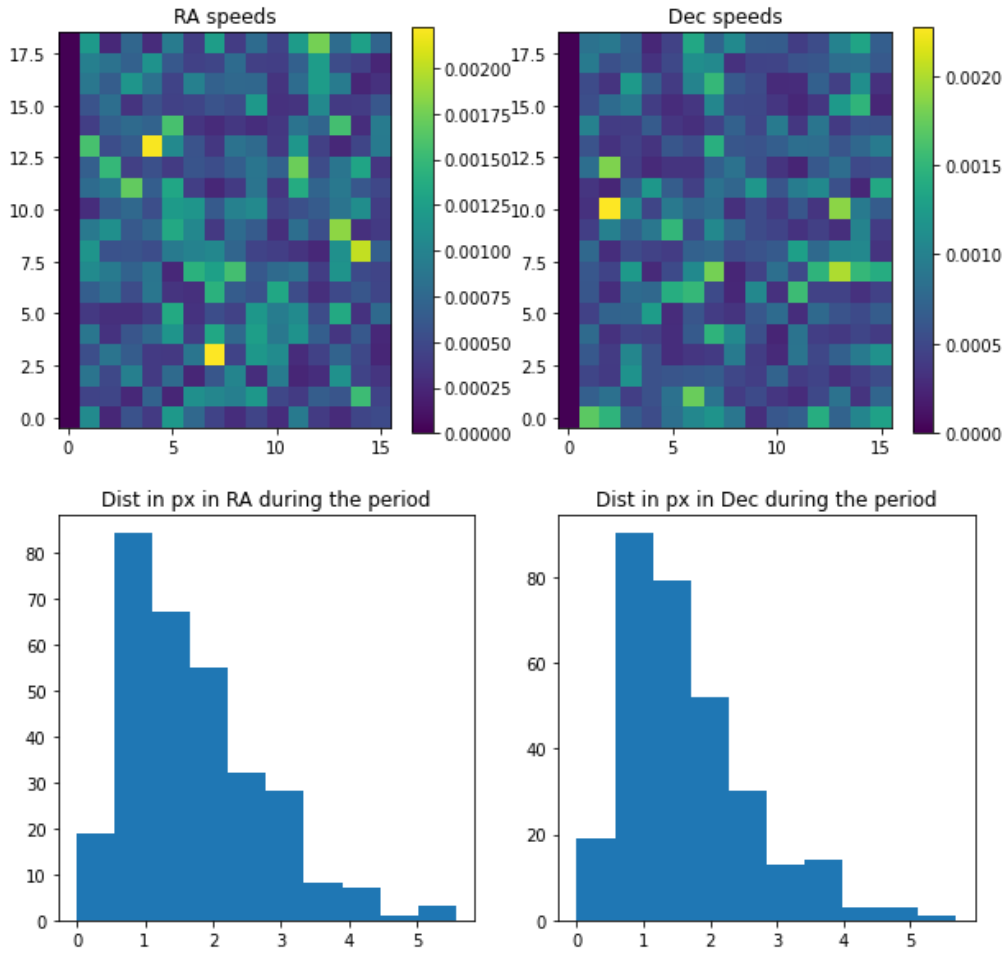

Figure 3.4: Speeds of simulated objects. The lower two graphs show histograms of speed distributions in RA and Dec directions, respectively. The upper two graphs show speeds in the two directions as pixel intensities, where each pixel correspond to an object in a cutout as they are placed inside the chosen subregion.

histograms of speed distributions in RA and Dec directions, respectively. The upper two graphs show speeds in the two directions as pixel intensities, where each pixel correspond to an object in a cutout as they are placed inside the chosen subregion.

Magnitudes of the simulated objects increase with their Dec position going from 18 to 27, i.e. the objects are becoming dimmer.

The `insertFakes` LSST task used for generating fake sources uses `galsim` Python package behind the scenes. The task code was modified so that it also adds fake sources to variance images (otherwise, the simulation would not be realistic). The code is using the PSF function estimated during the initial processing of the data and places objects at their proper locations into each original image depending on the object speed and the current image's timestamp.

The method used for inserting fake sources is the following:

```
def insert_fakes(objs, dataIds, butler, in_collection, out_collection,
                 add_to_variance=True):
  for dref in dataIds:
```

```
calexp = butler.get('calexp', dataId=dref, collections=in_collection)

calexp_clone = calexp.clone()
insert_fakes_in_calexp(calexp_clone, objs,
                       add_to_variance=add_to_variance)

butler.registry.registerCollection(out_collection, CollectionType.RUN)
butler.put(calexp_clone, 'calexp', dref, run=out_collection)
```

The method inserts all `MovingSource` objects (defined in code listingA.1) from `objs` into exposures from the `in_collection` and identified with `dataIds` and stores the results into `out_collection`. LSST `Butler`[2] object is used for reading and writing data (images and metadata) to and from a repository. A full description of the LSST API and how to use the `Butler` is out of scope of the discussion here as it is quite involved. The main method for simulating fake data, `insert_fakes_in_calexp`, is the same method later used in the likelihood and is given in code listingA.2.

Figure3.5shows one of the original images with some of the simulated objects added.

PROCESSING THE SIMULATED OBJECTS.After the fake moving objects are added to all images covering the region they're placed in, the images are processed with the LSST software stack so that the objects are detected and measured on the coadded images, thus simulating the typical scenario how the Multifit code would be used. The steps performed by the LSST processing pipeline are these [3]:

- Forward Global Calibration Method (FGCM) – photometric calibration using a reference catalog from Pan-STARRS
- `jointcal` – Refined astrometric calibration algorithm
- `makeWarp` – warps the exposures created by the singleFrame pipeline onto the pixel grids of patches described in the skymap
- `assembleCoadd` – assembles the warped images into coadds for each patch
- Coadd measurement – detects sources in coadded images; merges detections into a single catalog; deblends and measures sources in the individual coadds using the unified catalog; merges multi-band catalogs to identify the best positional measurement for each source
- Forced photometry – re-measures the coadds in each band using fixed positions

Multifit algorithm can now use data produced by these steps to find the most likely motions of the detected objects.

[2] More information here: `https://pipelines.lsst.io/v/weekly/py-api/lsst.daf.butler.Butler.html`

[3] For more information see `https://pipelines.lsst.io/v/v24_0_0/index.html`

FINDING THE DETECTIONS. After processing the images with the LSST processing pipeline, the resulting object catalog can be queried for detections and various measurements. Figure 3.6 shows with red X marks the objects that were detected and measured by the pipeline.

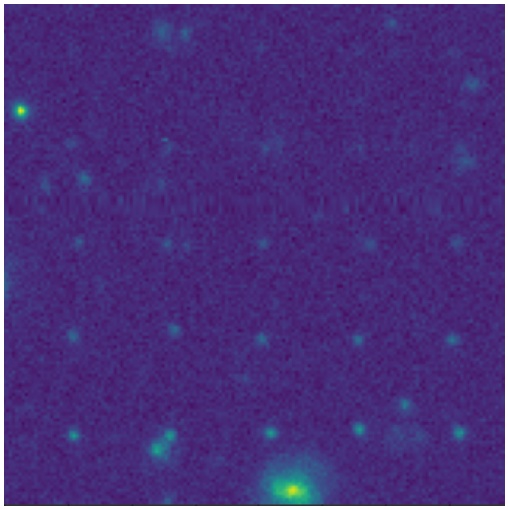

Figure 3.5: Part of the original image with simulated objects added.

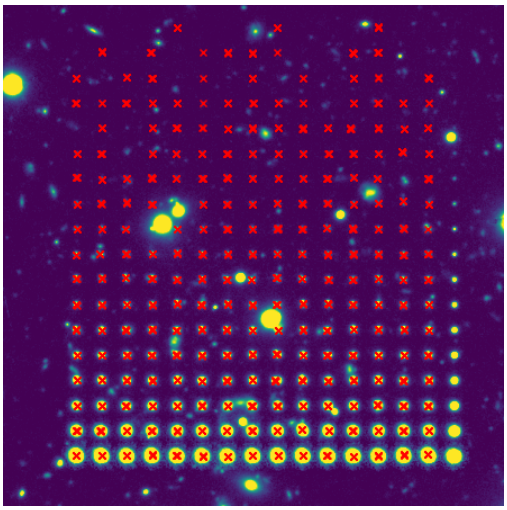

Figure 3.6: The simulated objects (red X marks) detected by the LSST processing pipeline.

REMOVING OBJECTS WITH BRIGHT SOURCES NEXT TO THEM. When an object is present next to a bright source, this can interfere with measurement algorithms. This is usually handled by the telescope's data processing pipeline itself by masking those sources. However, this does not always work perfectly. So, all such objects, that had bright sources next to them, were removed from the data set.

3.4.2 *The likelihood function*

As was already stated, the same likelihood function is used for both frequentist and Bayesian implementations. Python implementation of the likelihood function is given in listing A.3. That code also uses the model generation code from listing A.2 and helper code from A.1.

The likelihood function `lnlike` takes in a set of input images (along with other metadata such as image timestamps, PSF functions, variance images, etc.) of an object and a set of model parameters (magnitude, position, speeds). It uses model parameters to produce a corresponding set of images of the object representing the "model" (the `generate_model` function is used for this). It then subtracts generated images from the original ones, divides the square of this difference image by the variance image (pixel by pixel) and finally returns all pixels thusly obtained as "residuals". The optimization code using the likelihood function will then square and add up all the residuals to calculate the final χ^2 value.

MODEL GENERATION CODE. Images are generated according to provided parameters using the method `generate_model`, given in code listing A.2. The method receives an

instance of the `MovingSource` class (`obj`), RA and Dec coordinates for the center of the cutout (the resulting subimage), a cloned "calibrated exposure" object ("calexp" for short, an instance of the LSST class `lsst.afw.image.exposure.ExposureF`) and cutout size. It needs to add a fake source to a cloned exposure, instead of the original one, because otherwise the fake source would also be automatically added to the collection and saved on disk. The method then just calls `insert_fakes_in_calexp`, but without adding the fake to the variance image and without noise (as would be done when simulating testing data). The rest is the same as with the data simulation code, except that the method returns a cutout centered on the specified `ra` and `dec` parameters and of the size `image_size`.

↬ VALIDATING THE LIKELIHOOD FUNCTION.One way to validate the likelihood function is to examine its output for each of the inner steps it performs. Figure3.7shows outputs of these steps for three cutouts of an example simulated object from three different moments. Going from left to right, the images shown are:

1.The input image of a simulated object

2. The generated image within the likelihood function based on the parameters provided

3.The difference image between the two

4.The difference image divided by the square root of the variance image

There should be no "artifacts" in the difference images when using the correct model parameters. This was shown to be the case for the objects in the simulated dataset.

Figure3.8shows an example of images produced by the likelihood function when given model parameters do not match the actual object in the images. Typical "dipoles" can be seen in the difference images when positions of real and modeled objects do not match exactly.

↬ VALIDATING LIKELIHOOD WITH χ^2_{DoF}.Another way of validating the likelihood function is to calculate its "χ^2 per degrees of freedom" (χ^2_{DoF}) value by dividing the χ^2 value by the number of valid pixels (across all the images). This value should be close to 1, as was explained in section3.2.3.3. Indeed, this is the case for the simulated objects in the dataset when there are no other bright objects in the vicinity.

↬ VALIDATING LIKELIHOOD ON A GRID.Yet another way of making sure that the likelihood is modeled correctly is to visually examine likelihood values calculated on a grid for each pair of parameters (corner plot). To calculate the likelihood on a grid means to place an equally-spaced N-dimensional mesh into a region of the parameter space and calculate the value of the likelihood for each combination of the parameters defined by the mesh. This is time consuming and the cost of this calculation exponentially rises with the number of dimensions and the grid size. The marginalized parameter pair-wise plots can then be obtained by summing up the calculated values along all other dimensions not being visualized. Visualizing likelihood values like this can then reveal shapes due to correlation of different parameters.

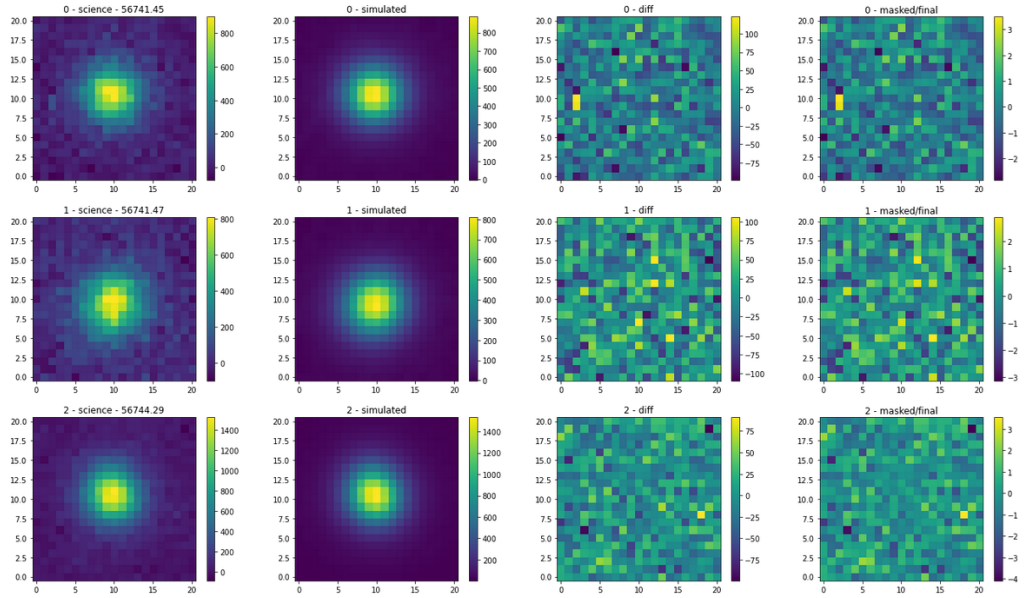

Figure 3.7: Images internally used and produced by the likelihood function for the first three cutouts of a simulated object, using the model parameters with which the object was originally simulated with. From left to right: the input images, the model-generated images, the difference images, the difference images divided by the square root of the variance images.

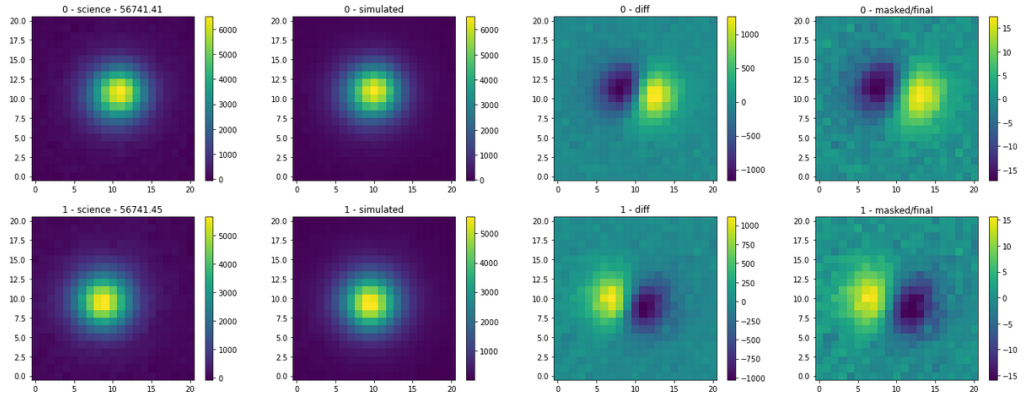

Figure 3.8: An example of images the likelihood function produces in the case of a non-optimal estimate.

3.5 Frequentist batch implementation

The "frequentist" batch Multifit implementation is based on [3], which was described in more detail in section3.1.1.2. The implementation uses methods of traditional (frequentist) statistics, which were explained in section3.2.4. This implementation is a batch implementation, meaning that it uses all available images at once, as was also explained previously.

The implementing code was packaged as an LSST task [4] and will probably be used within LSST pipelines for processing images from Vera C. Rubin Observatory. The code was used

[4] The code is available on Github here: `https://github.com/dirac-institute/batch-multifit-proto`

for comparing how well different algorithms work on the Multifit problem, comparing their accuracy and processing efficiency. The main parts of the code are given in listing A.4. The two main functions are `do_lmfit` and `do_multifit`. `do_multifit` function implements least squares, Levenberg-Marquardt, "Dogbox" and Trust Region Reflective (TRF) algorithms using `Scipy`'s `least_squares` and `leastsq` methods, while `do_lmfit` implements Cobyla, BFGS, L-BFGS-B, Conjugate-Gradient, Truncated Newton, "trust-region for constrained optimization", Sequential Linear Squares Programming, Simplicial Homology Global Optimization, Nelder-Mead, differential evolution, and Powell algorithms using the `LMFIT` Python package. `do_multifit` will call `do_lmfit` depending on the algorithm.

The `LMFIT` package (used by `do_lmfit` function) requires the user to formally declare model parameters and their boundaries. The two main functions also differ in the way they obtain covariance matrices: the `LMFIT`'s `minimize` method calculates it on its own and returns it directly, while in `do_multifit` the Jacobian matrix needs to be obtained first, then a Gauss-Newton approximation of the Hessian matrix is calculated by multiplying the Jacobian by its transpose, and then finally the covariance matrix is calculated by inverting the Hessian.

The different algorithms listed above were compared and the best performance, accuracy-wise, was obtained using the Powell algorithm. The Powell algorithm was the slowest, but accuracy was deemed to be more important for this thesis. The details are given in section 3.8.1.

Implementations of all the algorithms listed above result in a `LinAlgError` at varying percentages of runs, which means that no solution could be reached. This could probably be explained (this is pure speculation) by floating point inconsistencies when the algorithm gets to a point where changes in parameters suddenly make unrealistic jumps in the likelihood, or do not make any effect at all. Although the cause of this problem is still uncovered, the situation is mitigated by running the algorithm again a certain number of times with starting parameters varied randomly with standard deviation of 0.1%.

3.6 Bayesian batch implementation

This section describes a MCMC-based Multifit implementation processing all available images at the same time, i.e. the batch Multifit version. The result of this procedure is not just the maximum likelihood estimate of the model (moving point source) parameters, as is the case with the frequentist approach, but also an estimate of the full posterior probability distribution (see section3.2.8) of all model parameters.

In this section, implementation details are given, while various tests and results obtained using this implementation are described in section3.8.2.

3.6.1 *Emcee package and EnsembleSampler*

The main Python package used for implementing MCMC inference was `Emcee` ([68])[5]. It is an implementation of the "affine invariant ensemble MCMC sampling method" described

[5] More info at `https://emcee.readthedocs.io/`

back in section3.2.7.3. The package offers the `EnsembleSampler` class as the main driver of the inference process. It requires the following to be specified:

- **nwalkers** - Number of walkers (chains) in the ensemble
- **ndim** - Number of parameters to be estimated
- **log_prob_fn** - Log-likelihood function

Optionally, `EnsembleSampler` can also accept additional arguments to be forwarded to the log-likelihood function (the **args** argument), a list of "moves" to be used for choosing next walker positions, and a few arguments enabling parallelization of the computations.

The list of available moves can be found in the documentation [6] and several of them can be combined with specific weights (percentages specifying how often different moves will be used). Different moves are supposed to explore the parameter space more or less efficiently, using more or less iterations for exploring the full posterior distribution, but should arrive at the same result. For the dataset used here, `KDEMove` and `DEMove` were shown to work equally well.

3.6.2 *The log-likelihood function*

The log-likelihood function used by the `Emcee` package works a bit differently than the function that was used in section3.5. First of all, the log-likelihood function here is supposed to return a single log-likelihood value for each combination of input parameters values, i.e. the function should calculate the final χ^2 value on its own. If the function returns `-numpy.inf` (because of hard limits on parameter values, for example), this will stop the sampler from further exploring the parameter space in that direction.

Furthermore, parameter log-priors can (and should) be added to the final χ^2 value. The only prior used in the batch Multifit implementation is a wide Gaussian prior on vx and vy parameters, centered on 0 with a standard deviation of 1 pixel in 50 days.

So, a simple wrapper function is used for calling the original function described in3.4.2, checking the parameter hard limits and calculating the prior and the total χ^2 return value. In other words, the sampling procedure is essentially using the same code for generating model images and comparing them to the inputs as was used in section3.5. The likelihood function for the Bayesian batch Multifit is implemented as the `lnprob` method in listing A.5.

3.6.3 *Running MCMC*

After creating an `EnsembleSampler` instance, its `run_mcmc` method can then be used for running the actual inference. It only requires a starting *state* and a number of iterations to run. The state needs to specify the starting values for each walker, i.e. it needs to be an `array` with dimensions [number of parameters, number of walkers]. The authors of the `Emcee` package recommend using a "tiny random ball" for starting values, which will expand across the parameter space during the sampling procedure.

[6] Here: `https://emcee.readthedocs.io/en/stable/user/moves/`

After running `run_mcmc` method, the `EnsembleSampler` object contains the history of walker positions in the `chain` property. Another useful property is `acceptance_fraction`, containing fractions of proposed steps that were accepted, for each walker chain. Acceptance fraction was discussed back in section3.2.7.5). Its value should be somewhere between 0.25 and 0.5.

The best practice when running MCMC is to allow the sampler to "stabilize", i.e. to "converge to the equilibrium distribution" (see section3.2.7.1). However, finding out when that happens is a hard problem, so what is often done instead, which is also the approach taken here, is to run the process for a certain number of iterations (the so-called "burn-in") which are then discarded. Here, these are called "warm-up iterations".

3.6.4 *A 2D example*

In order to better explain the procedure, we will now revisit the example from section3.2.3.1 and try to fit a line with the `Emcee` package. With the model function `M` and log-likelihood function `logL` defined back in section3.3.1, a function for fitting a 2D line with `Emcee` can be defined as in listing3.1.

Listing 3.1: A function for MCMC fitting of a 2D line

```
import emcee
def doemcee(data, nwalkers=150, niter=500):
  ndim = 2
  x, y = data

  def lnlike(params):
    a, b = params
    return -np.sum(logL(x, y, a, b, sigma))

  sampler = emcee.EnsembleSampler(nwalkers, ndim, lnlike,
                                  moves=emcee.moves.DEMove())
  start_params = (np.random.rand(nwalkers * ndim)).\
                    reshape((nwalkers, ndim))
  state = sampler.run_mcmc(start_params, niter)
  return sampler.chain.reshape((-1, ndim))
```

The function returns walker chains that can be used for further processing of the results.

3.6.5 *Examining walker chains*

Once a MCMC procedure is done, the chain histograms can be analyzed to gain insight into the MCMC process. Firstly, a "chain plot" showing the timeline of walker (chain) values can be examined for visual inspection of the MCMC process.

Secondly, a "corner plot", whose manually-produced version was discussed back in section3.4.2.4, shows 2D "heat maps" for each pair of the estimated model parameters. Corner plots can be produced with the `corner` Python library. It automatically draws histograms for each of the individual parameters and can also show the true parameter values, if they are provided.

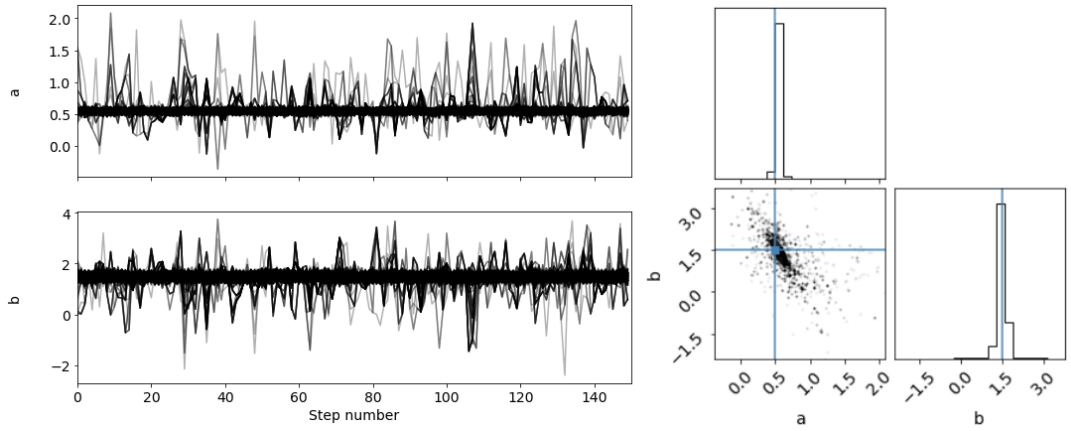

Figure 3.9: Left: An example of walker chain histories when estimating 2D-line model parameters (from a MCMC run using 150 walkers and 500 iterations). Right: Corner plot obtained from chain histories shown on the left with vertical and horizontal lines showing the true parameter values.

Example chains for the 2D line-fitting problem, and the matching corner plot, obtained from a MCMC run that was using 150 walkers and 500 iterations, are given in figure3.9.

3.6.6 *Examining Q-Q plots*

"Quantile-Quantile plots", or "Q-Q plots", are used for comparing two different distributions: quantiles of the two histograms being compared are plotted on the two axes and compared visually. The two distributions are more similar, the more the resulting graph – connecting the matching quantiles – resembles a straight line.

In the case discussed here, since the goal is to approximate the posterior with a Gaussian, it would be useful to compare histograms of each estimated parameter (i.e. a and b parameters for the 2D example, or M, x_0, y_0, v_x and v_y for the moving point source model) to histograms of a Gaussian distribution. If the Q-Q plot does not show a good correspondence between the two distributions, one can expect larger error obtained when approximating the distribution with a Gaussian.

In Python, 101 equally-spaced percentiles (0th to 100th) of a normalized Gaussian distribution (with the mean of 0 and standard deviation of 1) can be generated as follows:

```
from scipy.stats import norm
normq = norm.ppf(np.arange(0, 1.01, 0.01))
```

Equivalent percentiles to the ones above, normalized in the same way, can be obtained from the histogram chains for parameter i like this:

```
def normalized_percentiles(chains, i):
    std = np.std(chains[:, i])
    q = np.percentile(chains[:, i], np.arange(0, 101, 1))
    mean = np.mean(chains[:, i])
    # normalize the distribution:
    return (q - mean) / std
```

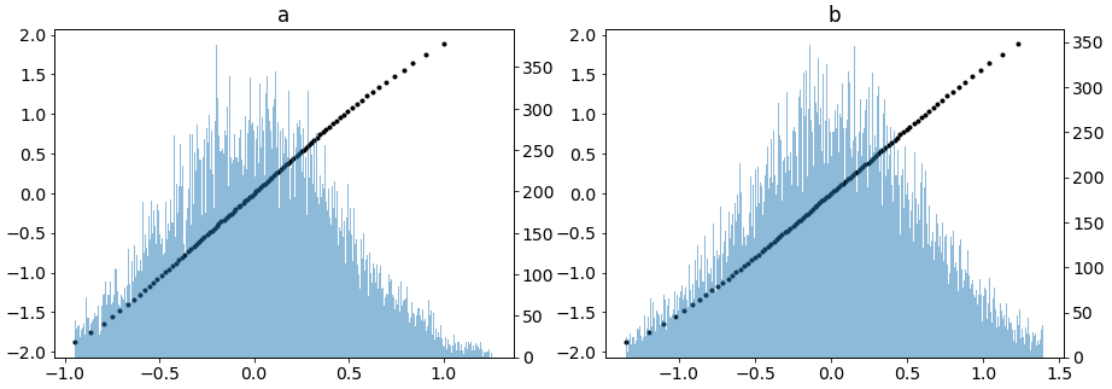

Figure 3.10: Q-Q plots for parameters *a* and *b* of the 2D line-fitting model corresponding to chains from figure3.9shown with dark points with the original histogram in the background.

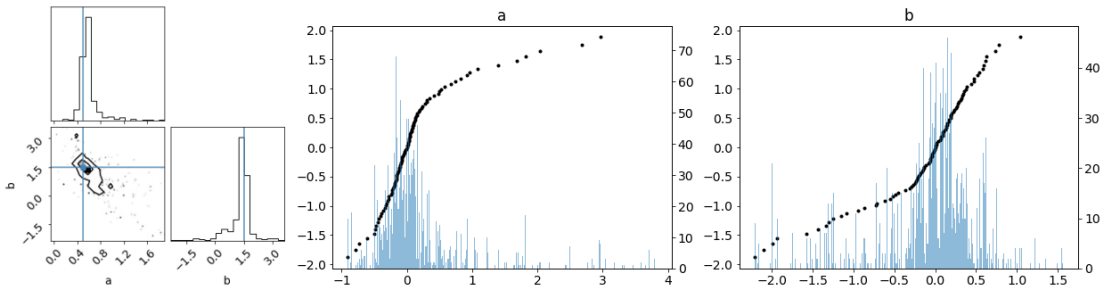

Figure 3.11: Corner plot and Q-Q plots for parameters *a* and *b* of the 2D line-fitting model obtained from a MCMC procedure ran with only 50 walkers and 50 iterations.

Figure3.10shows Q-Q plots comparing a normalized Gaussian distribution with percentiles of histograms for parameters *a* and *b* from the same MCMC run from figure3.9. One can notice a very good match between the two distributions (the lines are almost straight).

3.6.7 *The effect of poor sampling*

If we were to run a MCMC procedure with a lower number of walkers and iterations, the sampler would not adequately sample the posterior distribution and the resulting histograms would not be that reliable. Figure3.11shows an example of a corner plot and Q-Q plots for the 2D line-fitting model, from a MCMC procedure ran with only 50 walkers and 50 iterations. A poor correspondence to the Gaussian distribution can be noticed so one can expect larger errors when approximating these histograms with a Gaussian.

3.6.8 *Obtaining posterior distributions*

Obtaining a posterior distribution from a histogram of chain values is the critical issue for the success of the online procedure. If these estimates are inaccurate, the errors propagate and accumulate with each new data point (e.g. image) and the final result can veer away from the real solution.

For that reason quite some effort during the work on the online Multifit problem for this thesis has been put into accurate estimation of posterior distributions. Experiments were done with different methods and implementations:

- SKLearn's `KernelDensity` algorithm
- SKLearn's `BayesianGaussianMixture` and `GaussianMixture` algorithms
- Gaussian mixture implementations from Python's `pyro`, `SVAE` and `pygmmis` packages

All of these take a set of samples from a multidimensional space of possible values and produce an object representing an estimate of their distribution. These objects expose a `score_samples` method which returns log-likelihood values for new sets of proposed parameter values. Approximating a histogram with a probability distribution determined by a finite set of parameters, such as the Gaussian distribution, allows one to "compress" information contained in chain histograms to a much smaller set of numbers.

Obtaining a `BayesianGaussianMixture`, for example, from historical chain samples can be done in the following way:

```
from sklearn import mixture
gm = mixture.BayesianGaussianMixture(n_components=N,
                                     covariance_type='full')\
                  .fit(chains)
```

This gives us a set of Gaussian distributions, determined by their centers (the `means_` property) and their covariance matrices (the `covariances_`). The `weights_` property determines the influence of each Gaussian distribution in the overall mixture.

However, approximating the target distribution with a mixture of Gaussians would represent multiple competing solutions (a Gaussian for each solution). But these rarely exist. Usually a single solution is dominant and the procedure most often finds a single Gaussian "cluster". Therefore, in order to simplify the implementation, reduce the amount of data that needs to be stored, and lower the probability of the online procedure "wandering off" in wrong directions, **a single Gaussian estimate is always searched for**, still using the `BayesianGaussianMixture` implementation.

3.6.9 *Obtaining MAP estimates*

Maximum a-posteriori (MAP) estimates are "point estimates" of summary statistics of the posterior distribution, described back in section3.2.8.1. One is, of course, primarily interested in MAP estimates of the parameter values.

One way of obtaining these estimates is in the case of a single Gaussian distribution approximated from the histogram values: the most likely parameter values according to this distribution are determined by the Gaussian's N-dimensional mean point.

Another way of obtaining the most likely parameter values is by calculating medians of each parameter's histogram values. Histogram medians actually represent an unbiased and the most robust solution (see section3.2.8.1). This estimate is also more accurate compared to the one determined from the approximated Gaussian distribution.

For example, in the 2D line-fitting example presented above (figures3.9and3.10), a 2D Gaussian fit gives an estimate of $a = 0.548$ and $b = 1.477$ (with the true values being $a = 0.5$ and $b = 1.5$). Histogram medians give $a = 0.545$ and $b = 1.484$, which is about 17% smaller error.

The difference becomes even more noticeable when the Gaussian approximation poorly fits the underlying histogram. For the MCMC run from figure3.11, a 2D Gaussian fit gives an estimate of $a = 0.608$ and $b = 1.297$. Histogram medians give $a = 0.564$ and $b = 1.416$, which is about 50% smaller error.

Taking a cue from these facts, the Multifit method developed in this thesis employs the following procedure: it takes the covariance matrices (the shapes) from the estimated Gaussian mixture, but **corrects the mean of the most significant component using the histogram medians or Powell run results** (see the below section3.6.12).

3.6.10 *Expanding to the full Multifit model*

Everything described in the previous sections regarding the 2D-line model fitting can be applied to the 5D moving point source model. For the full moving point source model, the parameters to be estimated are magnitude M, offset from the detected position (x_0 and y_0 expressed in arc-seconds) and speed in both directions (v_x and v_y expressed in arc-seconds per day). An example of a corner plot with all these parameters is given in figure3.12.

One can notice the correlation between x_0 and v_x parameters and between y_0 and v_y parameters. This correlation is understandable: changing the starting point makes changing the speed in that direction also necessary (and all other speed values less likely). The magnitude plots show interesting correlations in the shape of a letter "T" (rotated counter-clockwise), especially with respect to v_x and v_y parameters. The vertical part of this shape is easily explainable: the same magnitude (brightness) of the object, near its true magnitude, corresponds to several combinations of v_x and x_0 parameters, or v_y and y_0 parameters, because they are correlated. The "stem" of the "T shape" (the horizontal part) is showing that a solution describing a fainter (larger magnitude) but static object is also somewhat likely.

3.6.11 *Choosing the number of iterations*

As was already said in section3.6.7, running MCMC with lower number of walkers and/or iterations results in poor sampling, which means that the samples obtained do not faithfully describe the posterior distribution. By consequence, the sample medians also do not correspond to the most likely parameter values. Left graph in figure3.13shows the χ^2 per degrees of freedom value depending on the number of iterations (the tests were made for two objects of magnitude 18). The χ^2 values clearly decrease, and hence the accuracy of the estimates increases, when more iterations are used.

3.6.12 *Correcting the estimates using Powell*

Accuracy of the estimates can be improved if the Powell algorithm was ran using the obtained medians as a starting point. The right graph in figure3.13shows the same runs

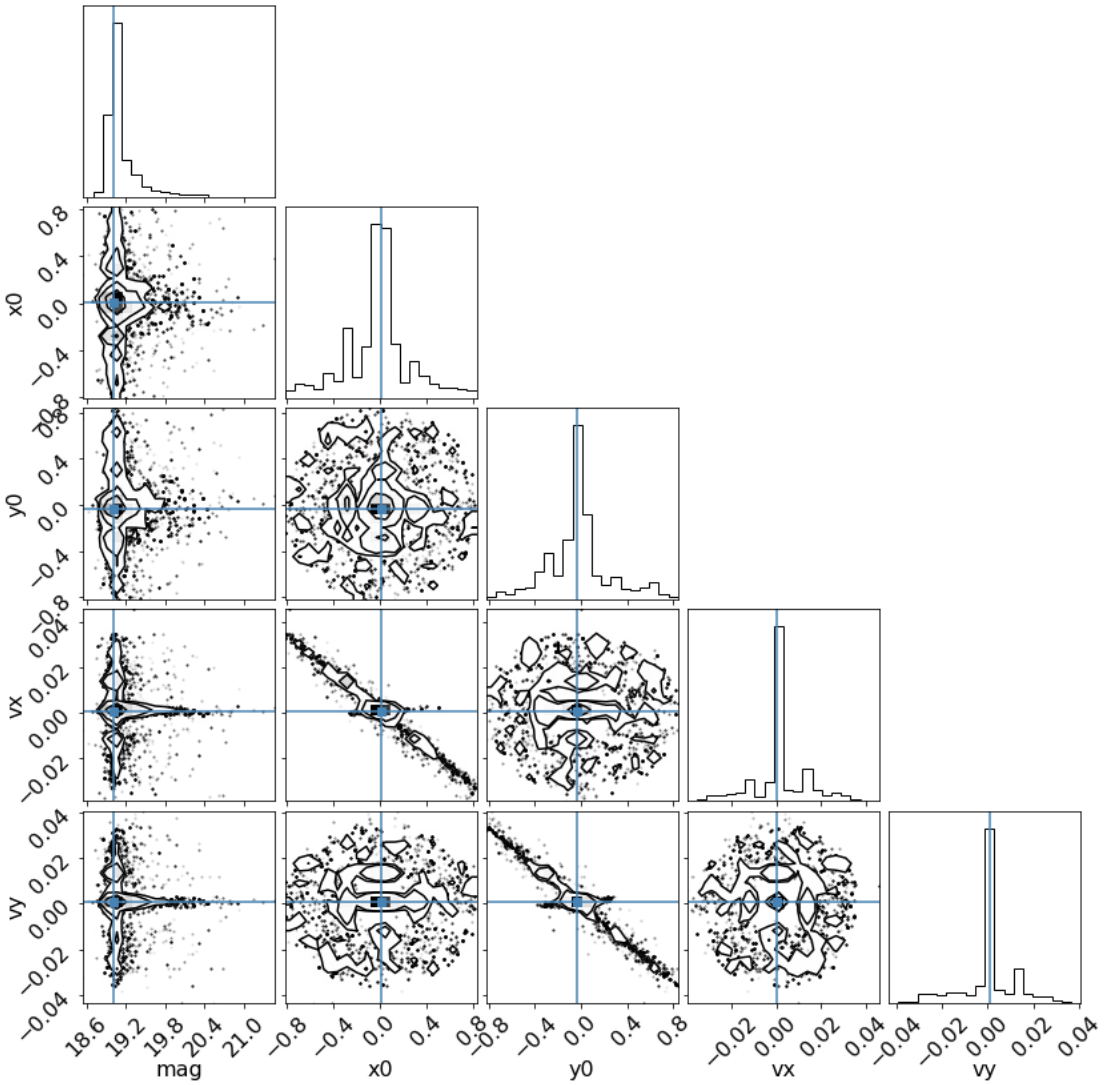

Figure 3.12: Corner plot for an example batch run for the moving point source model.

from the previous section, but corrected with runs of the Powell algorithm. One can see that the correlation between χ^2 and number of iterations is now gone: the same point-estimate accuracy can now be obtained with any number of iterations.

However, accurately describing the shape of the posterior distribution still depends on the number of iterations. As a compromise between algorithm running time and posterior-describing accuracy, **400 iterations was chosen** for all MCMC runs.

So, to summarize: the procedure employed by the Multifit method developed in this thesis is the following. Shapes of the Gaussian distributions forming the mixture describing the posterior are approximated from MCMC run samples (ran with 400 iterations), but the center of the Gaussian with the largest weight is corrected using a Powell run. If the Powell procedure does not produce a result, the sample median is used instead.

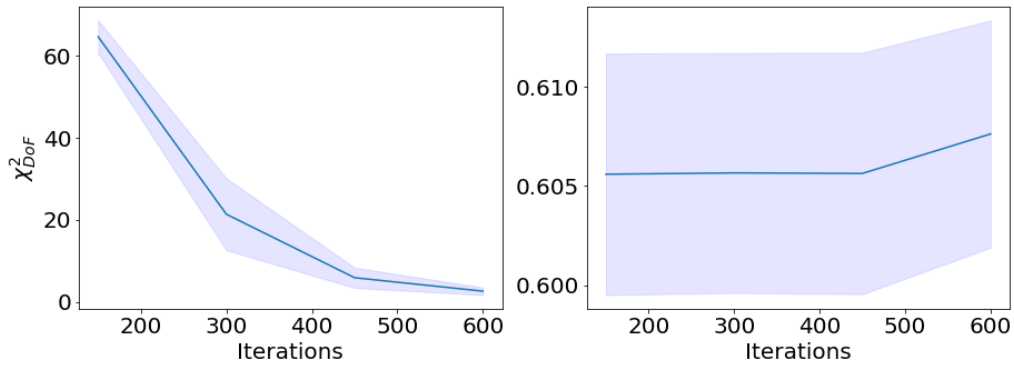

Figure 3.13: Left graph: χ^2 per degrees of freedom depending on the number of MCMC iterations, for two objects of magnitude 18. Right graph: The results improved with additional runs of the Powell algorithm.

3.6.13 *Plotting marginalized posteriors*

Another useful way of examining and validating the procedure is by plotting marginalized Gaussian approximations of the posterior, along with histograms that were used for obtaining the N-dimensional Gaussian. The approximated Gaussian mixture is evaluated on a grid and those values are then marginalized to a single dimension and plotted along with the original histogram values. Figure3.14shows an example for the moving point source model. It also shows the true and estimated values (the medians) of the parameters using dashed and solid vertical lines, respectively.

What can be noticed on these plots is that marginalized approximated posterior shows the worst fit for the magnitude parameter. The reason is that the chains do not explore magnitude values equally in both directions around the true value. That is because lower magnitude values represent brighter objects, which are less likely than fainter ones, and so the distribution of the samples is asymmetrical. Those kinds of distributions are harder to approximate with a Gaussian and so this introduces additional error into the overall process.

3.7BAYESIAN ONLINE IMPLEMENTATION

The online procedure, as was already stated, takes a single image at a time, along with any priors, and produces a posterior probability distribution. That posterior becomes a prior for the next image and so on. However, starting with only a single image when determining motion parameters does not make much sense (all speed values are equally likely) so the procedure described here starts by running in a "batch mode" for the first 3 images and then continues from there in an "online mode".

At each step of this process (i.e. after each image) one can calculate metrics and create diagnostic plots that were described in the previous sections: plots of chain histories, corner plots, Q-Q plots and plots of marginalized posteriors.

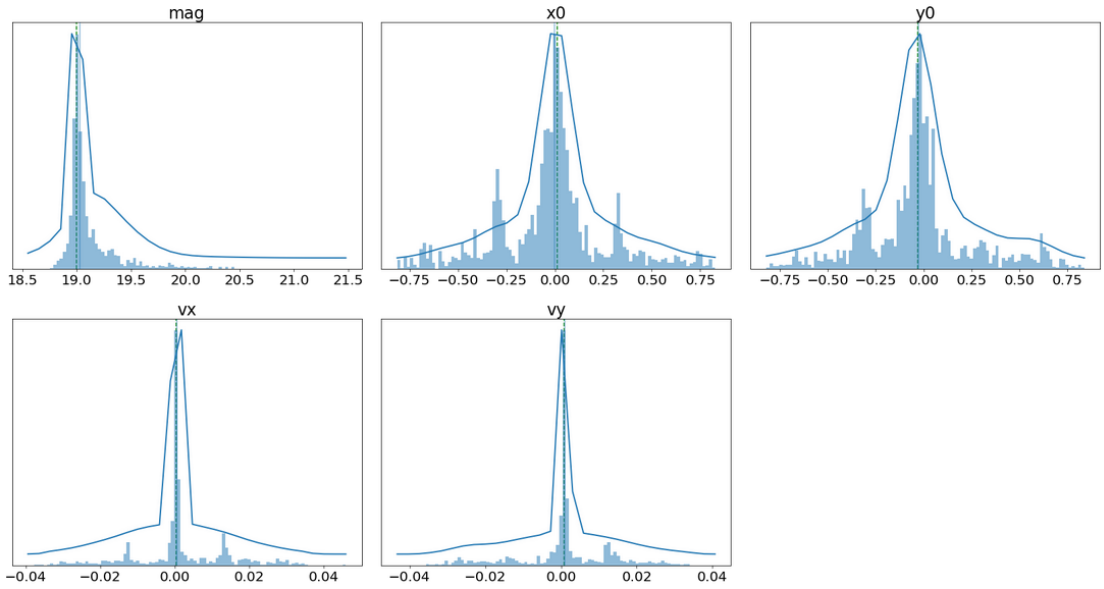

Figure 3.14: Marginalized posteriors and histograms for the moving point source model for a MCMC run on an object with the magnitude of 19. Vertical dashed lines show the true parameter values while the vertical solid lines mark histogram medians.

3.7.1 *A 2D example*

Illustrating the procedure is easier with a smaller number of dimensions so let us return to the 2D example from sections3.2.3.1and3.6.4. The `doemcee` function from code listing3.1 now needs to be expanded so that it takes a prior into account, if it is provided, and add it to the calculated χ^2 value. The resulting function that implements the change, given in code listing3.2, also calculates and returns the resulting posterior estimate (the `kderes` object). This object is actually an instance of a helper class `ScipyWrapper` (given in Addendum in code listingA.6) which is used for correcting the mean values of the most significant component in the Gaussian mixture, as was explained previously.

Listing 3.2: A modified function for MCMC fitting of a 2D line from listing3.1so that it can now be used in online mode.

```
import emcee
import numpy as np
def doemcee(data, nwalkers=150, niter=500, kde=None):
  ndim = 2
  x, y = data

  def lnlike(params):
    prior = 0
    if kde is not None:
      prior = kde.score([params])
    a, b = params
    return -np.sum(logL(x, y, a, b, sigma)) + prior

  sampler = emcee.EnsembleSampler(nwalkers, ndim, lnlike,
```

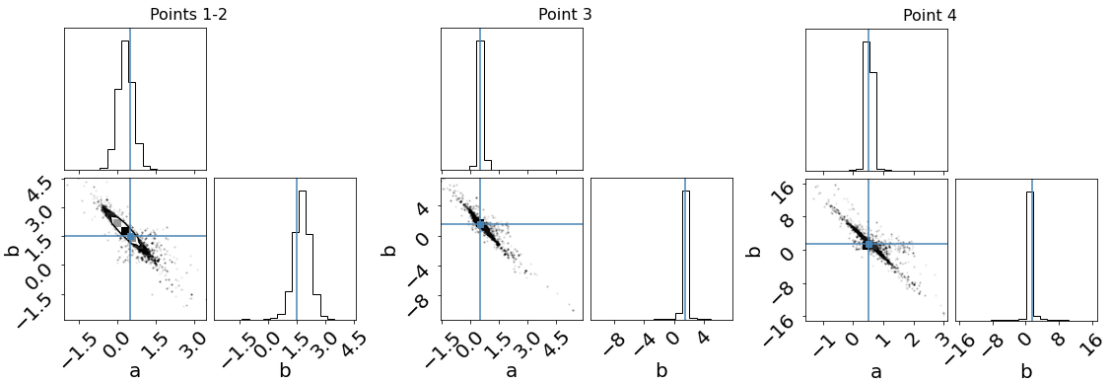

Figure 3.15: Fitting a 2D line using data from section3.3.1.

```
                                    moves=emcee.moves.DEMove())
  start_params = (np.random.rand(nwalkers * ndim) * 2).\
                      reshape((nwalkers, ndim))
  state = sampler.run_mcmc(start_params, niter)
  samples = sampler.chain.reshape((-1, ndim))
  kderes = mixture.BayesianGaussianMixture(n_components=5,
                                           covariance_type='full').\
                    fit(samples)
  medians = np.median(samples, axis=0)
  kderes = ScipyWrapper(kderes, medians)
  return samples, kderes
```

Running the full online procedure on all data points is simple as in the following code listing. Because estimating a line using a single point does not make much sense, the procedure starts with two points and proceeds from there.

```
samples, kde = doemcee((x[:2], y[:2]))
allsamples = [samples]
allkdes = [kde]
for xp, yp in zip(x[2:], y[2:]):
  samples, kde = doemcee((xp, yp), kde=kde)
  allsamples.append(samples)
  allkdes.append(kde)
```

Corner plots and other diagnostic plots can now be examined after each of the three steps (the first two data points together and then two others one by one). Figure3.15shows corner plots after each step. Figure3.16shows how a and b parameter estimates change after each step. The dashed and dotted horizontal lines show the true parameter values and the estimates obtained using the batch procedure, respectively. What can be noticed is that the estimates of the online procedure are approaching the estimate from the batch procedure closer and closer after each step.

3.7.2 *The full moving point source model*

The full code listing for the Bayesian MCMC implementation is given in code listingA.5. The same code can be used for both batch Multifit and individual steps of online Multifit,

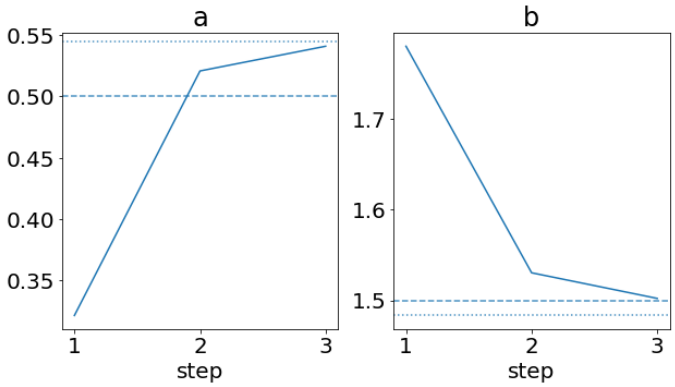

Figure 3.16: Fitting a 2D line using data from section3.3.1. The two graphs show how the estimated a and b parameter values change after each step of the online procedure. The dashed horizontal lines show the true parameter values and the dotted horizontal lines show the estimates from the batch procedure.

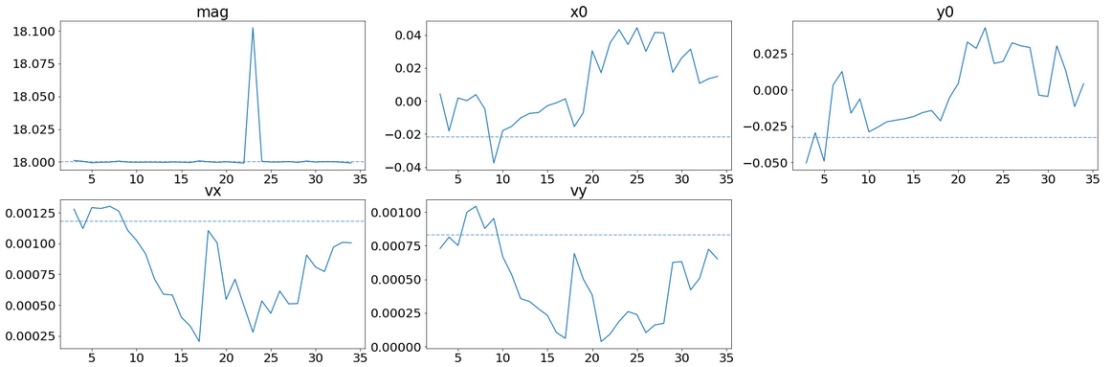

Figure 3.17: Estimates of the five parameters of the moving point source model depending on the step (image) number, for a run of the online procedure on an object of magnitude 18.

similarly to what was done with the 2D model: if the prior object (`kde`) is supplied, the prior will be calculated and added to the final result. Also, the `imagedata` object would contain all images (and the related information) for the batch procedure, but only a single image for a step of the online procedure. The listing also contains additional code for saving results, printing out diagnostic messages and so forth.

Similarly to what was done for the 2D mode, estimated parameter values can be plotted depending on the step (image). Figure3.17shows an example for a run on a simulated object of magnitude 18. Figure3.18shows χ^2_{DoF} values and "average trajectory distances" of the estimates at each step, for the same run. "Average trajectory distance" is calculated as an average Euclidian distance between estimated and true positions of the object at the simulated time points.

The figures show that the estimates at first move away from the true solution but go back after a certain number of steps. The reason for this is mostly due to the distribution of observation dates of the exposures in the simulated dataset. This distribution was shown in figure3.3back in section3.4.1. There are 10 observations on the first day, then a pause of 230 days when the next 8 observation are made. This causes the online algorithm to first start

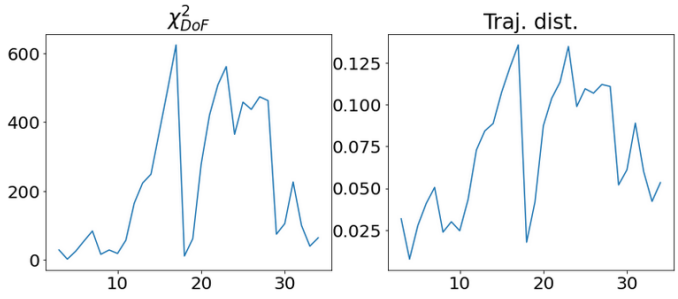

Figure 3.18: The χ^2_{DoF} and average trajectory distance of estimates from the true positions depending on the step (image) number, for the same run from figure3.17.

based on 10 observations made within a short period of time, too short for reliable motion estimation. Then the next 8 data points (also made within a too short of a time span) cause the solution to suddenly veer in a different direction. As can be seen from the figures, as the new points appear, the accuracy of the estimates slowly goes back close to the levels comparable to the batch procedure.

Figure3.23in the next section shows the same information averaged over MCMC runs on different objects, for each magnitude separately.

3.8Experimental results

In this section experimental results from tests of all three methods are given: frequentist batch, Bayesian batch and Bayesian online implementation. All three methods are evaluated using several metrics:

- **Sigma distance** – (Euclidian) distance from the real solution in the units of the estimated standard deviations.
- **Trajectory average distance** – obtained by calculating the average difference between object's real and estimated positions (distance in arc-seconds) at time points when each image was taken
- χ^2 **per degrees of freedom** – χ^2_{DoF} obtained from the likelihood function using the estimated parameter values, as was explained in section3.4.2.3
- **Raw parameter estimates** – offsets of the estimated from the real (simulated) parameter values
- **Duration** - Duration of the three methods are compared

3.8.1 *Frequentist batch Multifit (Lang et al., 2009)*

The frequentist batch Multifit implementation is an implementation of the idea from Lang et al., 2009 ([3]), as was already mentioned. Before running actual tests of the frequentist batch procedure, different optimization algorithms were compared. Figure3.19and table3.1 show their comparisons with regards to the accuracy of trajectory estimates, χ^2_{DoF} of the

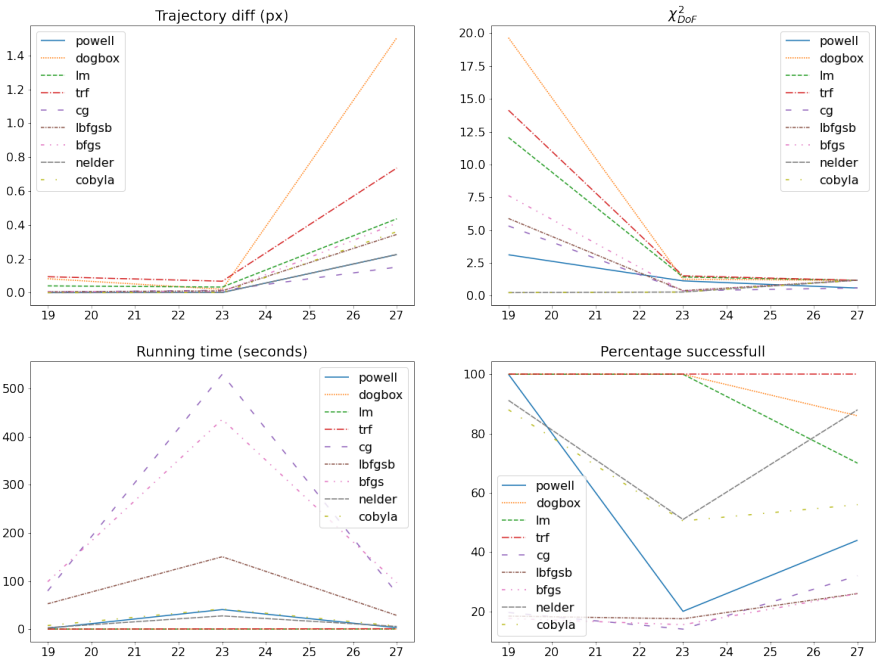

Figure 3.19: Comparison of different algorithms with regards to trajectory estimate accuracy, χ^2_{DoF} and duration, depending on magnitude. While being slower than the others, Powell algorithm shows the best accuracy overall.

solutions, duration and percentage of successful runs (failed runs do not complete because of a `LinAlgError`, mentioned in section3.5). All tests were done on 5 objects for each magnitude and were repeated 50 times (50 repetitions, times 5 objects, times number of magnitudes, times number of algorithms).

Powell algorithm was shown to be among the best algorithms accuracy-wise. Nelder and Cobyla have similar accuracy, however they are somewhat slower so the Powell algorithm was used for all tests and implementations.

What can also be noticed is that, for higher magnitudes, the difference in χ^2_{DoF} tends to become less significant. The reason is that the objects are then much fainter and, for the highest magnitudes, even not visible by the naked eye and so, the difference in residuals between the good and bad estimates is exponentially smaller than for lower magnitudes.

Also, Powell has high percentage of failed runs for higher magnitudes. TRF is the most robust in that regard, but it is much less accurate.

ACCURACY DEPENDING ON MAGNITUDE.Figure3.20and tables3.2and3.3show accuracy of the frequentist batch procedure (using the Powell algorithm), measured by the metrics previously described (sigma distances, trajectory average distances, χ^2_{DoF} of the solutions, and raw parameter estimates, all depending on magnitude. All the tests were done using only the first 15 images of each object.

Algorithm	Mag.	Traj.dist.	χ^2_{DoF}	Duration	% success
Powell	19	0.0005	3.12	1.83	100
	23	0.001	1.13	40.79	20
	27	0.225	0.58	3.14	44
Dogbox	19	0.082	19.66	0.17	100
	23	0.020	1.26	0.26	100
	27	1.504	1.17	0.41	86
LM	19	0.040	12.06	0.25	100
	23	0.033	1.43	0.31	100
	27	0.436	1.17	0.52	70
TRF	19	0.094	14.14	0.22	100
	23	0.068	1.51	0.66	100
	27	0.736	1.17	0.97	100
CG	19	0.0042	5.32	79.53	20
	23	0.014	0.37	529.64	14
	27	0.151	0.59	75.03	32
LBFGSB	19	0.005	5.88	52.82	18
	23	0.011	0.39	150.55	18
	27	0.344	1.18	28.7	26
BFGS	19	0.006	7.63	98.66	18
	23	0.006	0.35	435.83	16
	27	0.406	1.18	96.59	26
Nelder	19	0.0002	0.24	3.2	92
	23	0.001	0.28	27.67	51
	27	0.226	1.18	5.82	88
Cobyla	19	0.0002	0.23	7.67	88
	23	0.001	0.28	42.10	51
	27	0.360	1.18	6.56	56

Table 3.1

As can be seen from the plots and the tables, and as could be expected, accuracy of the estimates tends to decrease with magnitude. The average trajectory distance of the estimates goes up more significantly only above magnitude 23, and even for magnitude 27 the mean is only 0.7 pixels. “Sigma distances” are Euclidian distances (in 5D space) from the true parameter values in the units of standard deviations. But that measure depends on standard deviations estimated by the optimization procedure (Powell algorithm) and makes sense only when comparing different runs of the same type of algorithm (different algorithms and procedures tend to have wildly different uncertainty estimates).

As was already noted, χ^2_{DoF} tends to vary less for higher magnitudes and can be seen here to start to stagnate, although the accuracy of the estimates is starting to get significantly worse. In other words, the signal is getting lost in the noise and the algorithm is starting to have more trouble finding the solution.

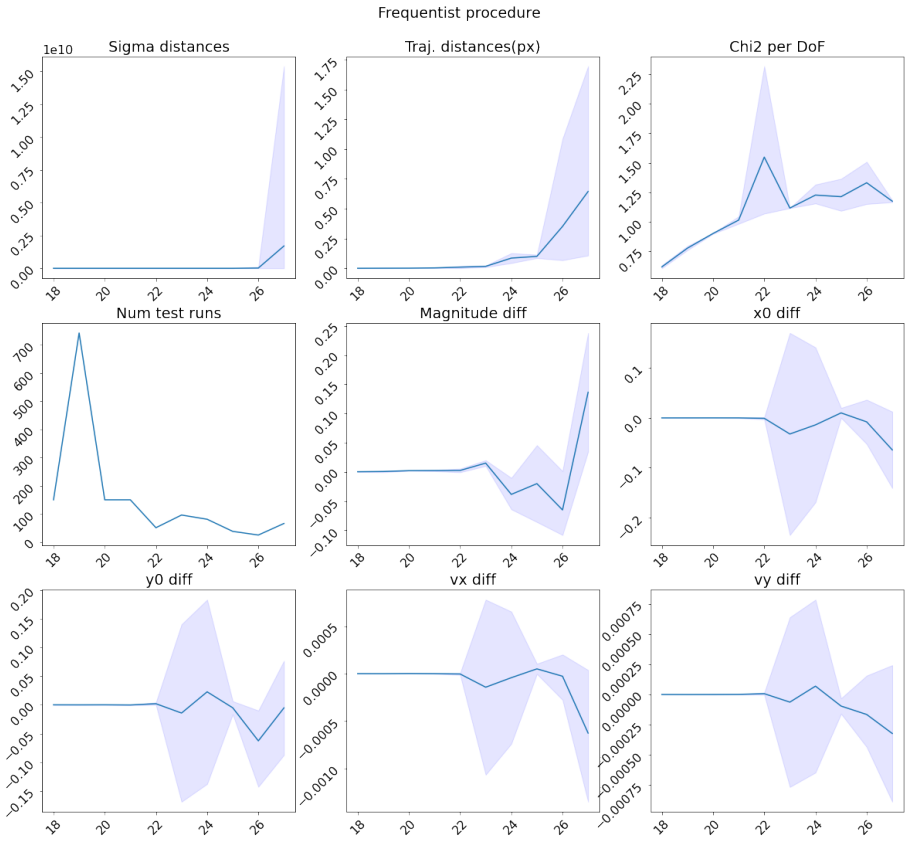

Figure 3.20: Frequentist batch procedure results comparing various metrics depending on magnitude. The shaded areas mark the range of a standard deviation around the mean.

3.8.2 *Bayesian batch Multifit*

The Bayesian batch Multifit, i.e. running MCMC on all available images at the same time (batch mode), was run for 8-10 objects of each magnitude using the first 15 images of each object. The results, measured by the metrics previously described, are given in figure3.21 and tables3.4and3.5.

Average trajectory distances are somewhat higher than for the frequentist procedure, but not much. The same can be said for χ^2_{DoF} and sigma distances as before, with the note that χ^2_{DoF} is also, not surprisingly, higher than for the frequentist procedure.

BAYESIAN BATCH DURATION.Processing 15 images in batch mode with 150 walkers and 400 iterations takes about 172 minutes on average. This is obviously prohibitively high. However, batch implementation is not meant to be used directly and is done only to serve as a comparison between frequentist and online implementations: batch implementation is not expected to be more accurate than the frequentist one and online implementation is similarly not expected to be more accurate than the batch implementation.

	mag	18	19	20	21	22
Tr.dist.(px)	Avg	3.6×10^{-4}	8.7×10^{-4}	1.3×10^{-3}	3.5×10^{-3}	9.98×10^{-3}
	Stdev	8.9×10^{-5}	2.2×10^{-3}	2.7×10^{-4}	1.2×10^{-3}	9.9×10^{-3}
χ^2_{DoF}	Avg	0.67	0.78	0.90	1.02	1.55
	Stdev	0.014	0.018	0.003	0.031	0.768
Sig. dist.	Avg	1.8	2.7	2.9	2.0	6.2
	Stdev	0.6	2.9	0.4	0.6	7.0
Mag. diff	Avg	0.0003	0.0007	0.0021	0.0023	0.0025
	Stdev	0.0003	0.0013	0.0003	0.0011	0.0031
x_0 diff	Avg	8.1×10^{-6}	3.8×10^{-5}	7.0×10^{-5}	3.4×10^{-5}	-8.2×10^{-4}
	Stdev	2.0×10^{-5}	3.5×10^{-4}	8.0×10^{-5}	3.8×10^{-4}	2.2×10^{-3}
y_0 diff	Avg	3.0×10^{-5}	-2.9×10^{-5}	6.5×10^{-5}	-2.2×10^{-4}	1.9×10^{-3}
	Stdev	4.1×10^{-5}	6.2×10^{-5}	1.1×10^{-4}	3.6×10^{-4}	2.3×10^{-3}
v_x diff	Avg	-1.1×10^{-8}	-3.4×10^{-7}	1.2×10^{-6}	-1.2×10^{-7}	-3.7×10^{-6}
	Stdev	2.4×10^{-7}	4.2×10^{-7}	3.8×10^{-7}	1.9×10^{-6}	9.6×10^{-6}
v_y diff	Avg	1.8×10^{-8}	-3.5×10^{-7}	3.6×10^{-8}	8.3×10^{-7}	6.7×10^{-6}
	Stdev	2.5×10^{-7}	4.9×10^{-7}	1.2×10^{-6}	1.0×10^{-6}	9.2×10^{-6}

Table 3.2: Frequentist batch procedure: accuracy results corresponding to figure3.20for magnitudes 18-22.

	mag	23	24	25	26	27
Tr.dist.(px)	Avg	1.65×10^{-2}	8.56×10^{-2}	1.00×10^{-1}	3.51×10^{-1}	6.44×10^{-1}
	Stdev	6.8×10^{-3}	4.2×10^{-2}	1.4×10^{-2}	7.4×10^{-1}	1.1
χ^2_{DoF}	Avg	1.12	1.23	1.21	1.33	1.17
	Stdev	0.001	0.089	0.151	0.180	0.008
Sig. dist.	Avg	1.5×10^{4}	3.3×10^{3}	1.9×10^{1}	1.9×10^{7}	1.7×10^{9}
	Stdev	1.1×10^{5}	1.7×10^{4}	1.8×10^{1}	9.3×10^{7}	1.4×10^{10}
Mag. diff	Avg	0.015	-0.038	-0.020	-0.065	0.136
	Stdev	0.005	0.028	0.066	0.066	0.102
x_0 diff	Avg	-3.2×10^{-2}	-1.4×10^{-2}	1.0×10^{-2}	-7.9×10^{-3}	-6.4×10^{-2}
	Stdev	2.0×10^{-1}	1.6×10^{-1}	9.9×10^{-3}	4.4×10^{-2}	7.7×10^{-2}
y_0 diff	Avg	-1.4×10^{-2}	2.3×10^{-2}	-5.2×10^{-3}	-6.3×10^{-2}	-5.3×10^{-3}
	Stdev	1.5×10^{-1}	1.6×10^{-1}	1.1×10^{-2}	8.0×10^{-2}	8.2×10^{-2}
v_x diff	Avg	-1.4×10^{-4}	-4.3×10^{-5}	5.0×10^{-5}	-2.7×10^{-5}	-6.3×10^{-4}
	Stdev	9.2×10^{-4}	7.0×10^{-4}	5.4×10^{-5}	2.5×10^{-4}	7.2×10^{-4}
v_y diff	Avg	-6.3×10^{-5}	6.9×10^{-5}	-9.6×10^{-5}	-1.7×10^{-4}	-3.2×10^{-4}
	Stdev	7.0×10^{-4}	7.2×10^{-4}	6.4×10^{-5}	3.2×10^{-4}	5.7×10^{-4}

Table 3.3: Frequentist batch procedure: accuracy results corresponding to figure3.20for magnitudes 23-27.

3.8.3 *Bayesian online Multifit*

Online Bayesian procedure was tested in two ways: testing the accuracy depending on magnitude (as was done for the previous two procedures), and also testing the accuracy depending on the step (image) number.

Accuracy results of the Bayesian online procedure, measured by the metrics previously

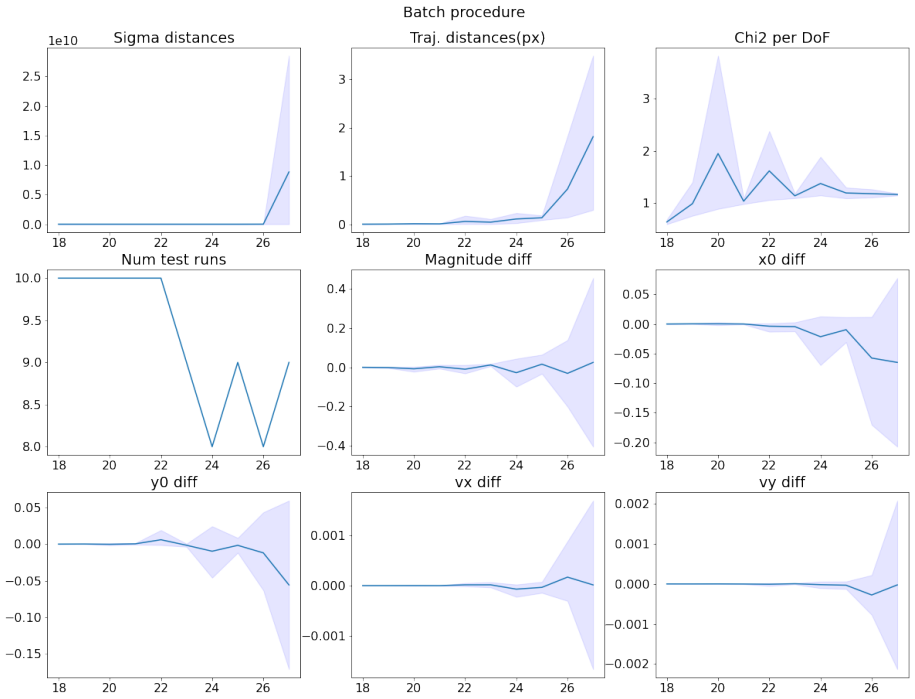

Figure 3.21: Bayesian batch procedure results comparing various metrics depending on magnitude. The shaded areas mark the range of a standard deviation around the mean.

	mag	18	19	20	21	22
Tr.dist.(px)	Avg	6.1×10^{-4}	2.7×10^{-3}	9.2×10^{-3}	6.7×10^{-3}	5.7×10^{-2}
	Stdev	5.6×10^{-4}	3.8×10^{-3}	1.0×10^{-2}	7.9×10^{-3}	1.2×10^{-1}
χ^2_{DoF}	Avg	0.64	0.99	1.95	1.04	1.62
	Stdev	0.05	0.41	1.87	0.05	0.76
Sig. dist.	Avg	3.21	5.78	8.05	4.36	17.74
	Stdev	2.10	6.21	8.21	5.38	36.90
Mag. diff	Avg	-0.0002	-0.0013	-0.0077	0.0029	-0.0098
	Stdev	0.0008	0.0025	0.0146	0.0093	0.0214
x_0 diff	Avg	-4.44×10^{-5}	2.82×10^{-4}	2.49×10^{-4}	1.28×10^{-8}	-3.80×10^{-3}
	Stdev	8.85×10^{-5}	6.21×10^{-4}	2.26×10^{-3}	7.97×10^{-4}	9.29×10^{-3}
y_0 diff	Avg	1.40×10^{-5}	1.51×10^{-4}	-2.46×10^{-4}	3.74×10^{-4}	5.98×10^{-3}
	Stdev	6.94×10^{-5}	3.09×10^{-4}	1.26×10^{-3}	1.02×10^{-3}	1.31×10^{-2}
v_x diff	Avg	-1.16×10^{-7}	9.20×10^{-9}	4.08×10^{-7}	-5.37×10^{-7}	1.50×10^{-5}
	Stdev	1.88×10^{-7}	7.90×10^{-7}	3.56×10^{-6}	1.83×10^{-6}	3.60×10^{-5}
v_y diff	Avg	-1.88×10^{-8}	-2.49×10^{-7}	8.13×10^{-7}	-1.71×10^{-6}	-7.43×10^{-6}
	Stdev	5.40×10^{-7}	1.09×10^{-6}	3.50×10^{-6}	5.08×10^{-6}	3.96×10^{-5}

Table 3.4: Bayesian batch procedure: accuracy results corresponding to figure3.21for magnitudes 18-22.

	mag	23	24	25	26	27
Tr.dist.(px)	Avg	4.26×10^{-2}	1.08×10^{-1}	1.33×10^{-1}	7.28×10^{-1}	1.81
	Stdev	6.47×10^{-2}	1.23×10^{-1}	4.75×10^{-2}	1.10	1.68
χ^2_{DoF}	Avg	1.14	1.38	1.20	1.18	1.17
	Stdev	0.04	0.51	0.10	0.08	0.02
Sig. dist.	Avg	2.24	1.11×10^{1}	7.97	7.55×10^{6}	8.84×10^{9}
	Stdev	1.40	1.40×10^{1}	1.42×10^{1}	2.00×10^{7}	1.97×10^{10}
Mag. diff	Avg	0.01	-0.03	0.02	-0.03	0.02
	Stdev	0.01	0.07	0.05	0.17	0.43
x_0 diff	Avg	-4.49×10^{-3}	-2.15×10^{-2}	-9.69×10^{-3}	-5.76×10^{-2}	-6.50×10^{-2}
	Stdev	8.03×10^{-3}	4.80×10^{-2}	2.11×10^{-2}	1.13×10^{-1}	1.42×10^{-1}
y_0 diff	Avg	-1.59×10^{-3}	-9.74×10^{-3}	-1.68×10^{-2}	-1.20×10^{-2}	-5.58×10^{-2}
	Stdev	2.09×10^{-3}	3.62×10^{-2}	1.02×10^{-2}	5.55×10^{-2}	1.15×10^{-1}
v_x diff	Avg	1.54×10^{-5}	-7.07×10^{-5}	-3.50×10^{-5}	1.69×10^{-4}	1.36×10^{-5}
	Stdev	5.12×10^{-5}	1.53×10^{-4}	1.09×10^{-4}	7.14×10^{-4}	1.68×10^{-3}
v_y diff	Avg	4.95×10^{-6}	-1.81×10^{-5}	-3.19×10^{-5}	-2.76×10^{-4}	-2.47×10^{-5}
	Stdev	1.39×10^{-5}	8.83×10^{-5}	9.26×10^{-5}	4.96×10^{-4}	2.10×10^{-3}

Table 3.5: Bayesian batch procedure: accuracy results corresponding to figure3.21for magnitudes 23-27.

	mag	18	19	20	21	22
Tr.dist.(px)	Avg	0.38	0.28	0.46	0.01	0.35
	Stdev	0.55	0.50	0.37	0.02	0.39
χ^2_{DoF}	Avg	744.52	164.05	53.82	1.06	5.22
	Stdev	1.48×10^{3}	3.51×10^{2}	5.80×10^{1}	1.08×10^{-1}	4.59
Sig. dist.	Avg	6.32	6.19	5.32	2.69	6.66
	Stdev	5.02	6.1	1.64	2.17	9.04
Mag. diff	Avg	-1.41×10^{-4}	-1.42×10^{-3}	-6.21×10^{-3}	-7.15×10^{-4}	-1.56×10^{-2}
	Stdev	0.001	0.003	0.016	0.004	0.041
x_0 diff	Avg	0.003	0.006	0.009	-0.001	0.004
	Stdev	0.006	0.010	0.009	0.001	0.010
y_0 diff	Avg	0.010	0.005	0.009	-0.0001	0.011
	Stdev	0.014	0.011	0.008	0.001	0.013
v_x diff	Avg	-1.73×10^{-4}	-2.61×10^{-4}	-4.51×10^{-4}	4.64×10^{-6}	-2.66×10^{-4}
	Stdev	3.62×10^{-4}	4.60×10^{-4}	4.19×10^{-4}	1.54×10^{-5}	3.60×10^{-4}
v_y diff	Avg	-4.79×10^{-4}	-2.37×10^{-4}	-4.05×10^{-4}	7.39×10^{-6}	-3.20×10^{-4}
	Stdev	7.09×10^{-4}	5.15×10^{-4}	3.40×10^{-4}	2.25×10^{-5}	4.64×10^{-4}

Table 3.6: Bayesian online procedure: accuracy results corresponding to figure3.22for magnitudes 18-22.

described, depending on magnitude, are given in figure3.22and tables3.6and3.7. These results show the final accuracy after the 15th image.

PERFORMANCE AFTER EACH STEP.Figure3.23shows the average and the spread (standard deviations) of "trajectory distances" after each image, for each magnitude separately. The procedure was run on 10 objects for each magnitude, using the first 15 images

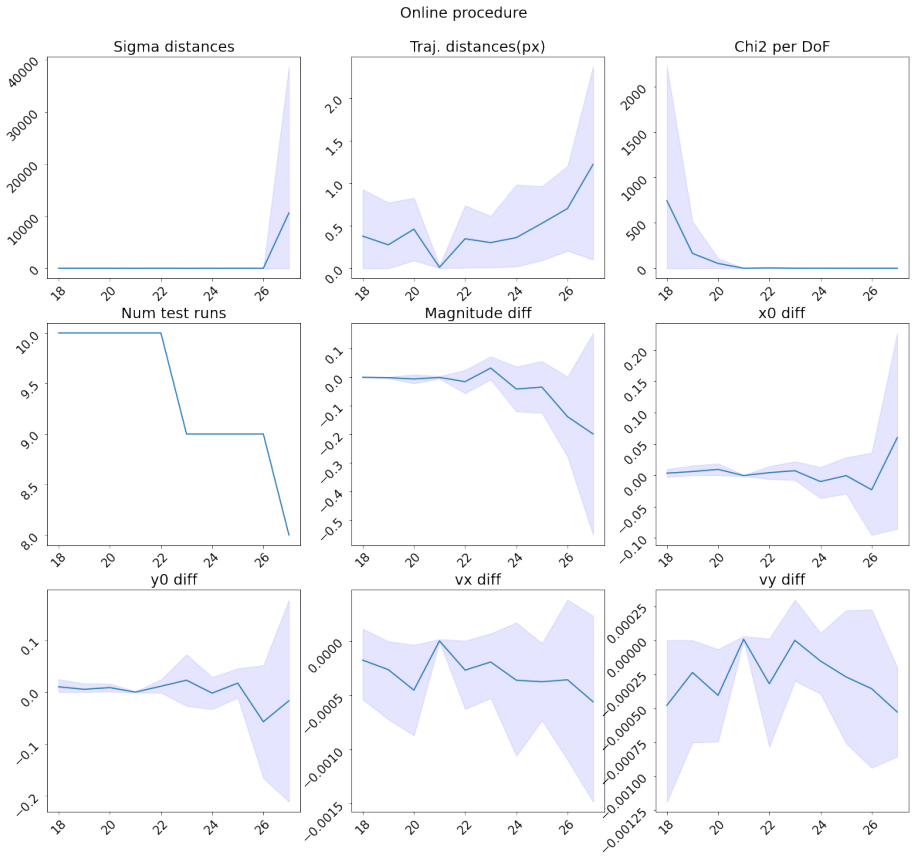

Figure 3.22: Bayesian online procedure results comparing various metrics depending on magnitude. The shaded areas mark the range of a standard deviation around the mean.

only. The graphs show steady improvement in results with each new image, more so for higher magnitudes. The reason is that the estimates for objects with higher magnitude (fainter objects) start with worse results and have more room for improvement. For brighter objects the procedure makes a pretty good estimate at the start and can improves more gradually.

BAYESIAN ONLINE DURATION.Regarding the duration, processing a single image takes about 7 minutes on average when running 400 iterations with 150 walkers. With 150 iterations the running time is decreased to 3 minutes, but that also makes the quality of posterior estimates also poorer, so it is a tradeoff.

3.8.4 *Comparison of all Multifit implementations*

As a summary of performance of all three methods, average trajectory distance for all three methods depending on magnitude is shown in figure3.24and table3.8.

	mag	23	24	25	26	27
Tr.dist.(px)	Avg	0.30	0.36	0.53	0.70	1.22
	Stdev	0.31	0.62	0.43	0.50	1.15
χ^2_{DoF}	Avg	1.48	1.84	1.28	1.22	1.17
	Stdev	5.45×10^{-1}	1.22	1.30×10^{-1}	1.15×10^{-1}	2.02×10^{-2}
Sig. dist.	Avg	2.15	4.84	5.72	2.81	1.06×10^{4}
	Stdev	1.53	3.47	7.15	1.96	2.81×10^{4}
Mag. diff	Avg	3.20×10^{-2}	-4.13×10^{-2}	-3.46×10^{-1}	-1.38×10^{-1}	-1.99×10^{-1}
	Stdev	0.04	0.08	0.09	0.14	0.35
x_0 diff	Avg	0.0071	-0.0102	-0.0006	-0.0233	0.0600
	Stdev	0.0148	0.0264	0.0290	0.0722	0.1671
y_0 diff	Avg	0.023	-0.002	0.017	-0.057	-0.017
	Stdev	0.050	0.031	0.028	0.109	0.195
v_x diff	Avg	-1.91×10^{-4}	-3.59×10^{-4}	-3.74×10^{-4}	-3.55×10^{-4}	-5.59×10^{-4}
	Stdev	3.35×10^{-4}	7.01×10^{-4}	3.56×10^{-4}	7.40×10^{-4}	9.24×10^{-4}
v_y diff	Avg	-3.88×10^{-7}	-1.52×10^{-4}	-2.70×10^{-4}	-3.56×10^{-4}	-5.28×10^{-4}
	Stdev	2.99×10^{-4}	2.41×10^{-4}	4.89×10^{-4}	5.82×10^{-4}	3.27×10^{-4}

Table 3.7: Bayesian online procedure: accuracy results corresponding to figure3.22for magnitudes 23-27.

Mag.		Lang et al, 2009	Bayes. batch	Bayes.online
18	Avg	0.0004	0.0006	0.3777
	Stdev	0.0001	0.0006	0.5486
19	Avg	0.0009	0.0027	0.2755
	Stdev	0.0022	0.0038	0.4959
20	Avg	0.0013	0.0092	0.4592
	Stdev	0.0003	0.0105	0.3672
21	Avg	0.0035	0.0067	0.0116
	Stdev	0.0012	0.0079	0.0214
22	Avg	0.0100	0.0572	0.3468
	Stdev	0.0090	0.1173	0.3906
23	Avg	0.0165	0.0492	0.3165
	Stdev	0.0068	0.0645	0.2993
24	Avg	0.0856	0.1179	0.4003
	Stdev	0.0417	0.1192	0.6477
25	Avg	0.1002	0.1333	0.5279
	Stdev	0.0144	0.0475	0.4343
26	Avg	0.3511	0.7278	0.7720
	Stdev	0.7350	1.0989	0.4841
27	Avg	0.6443	1.8109	1.2208
	Stdev	1.0500	1.6784	1.1513

Table 3.8: Average trajectory distance for all three methods depending on magnitude and corresponding to the figure3.24.

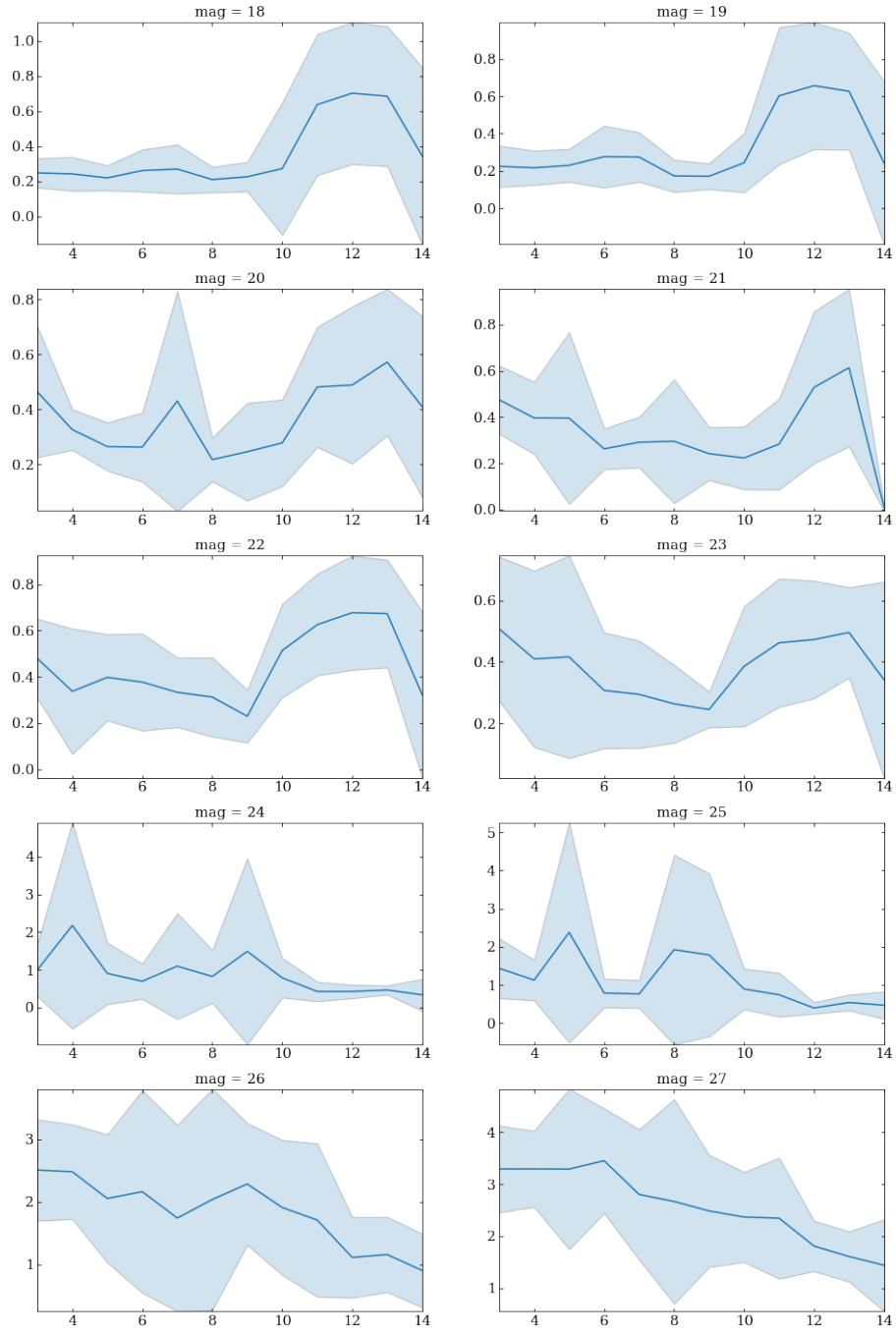

Figure 3.23: Average trajectory distance measure (for 10 tests/objects) plotted for each magnitude separately, depending on step (image) number.

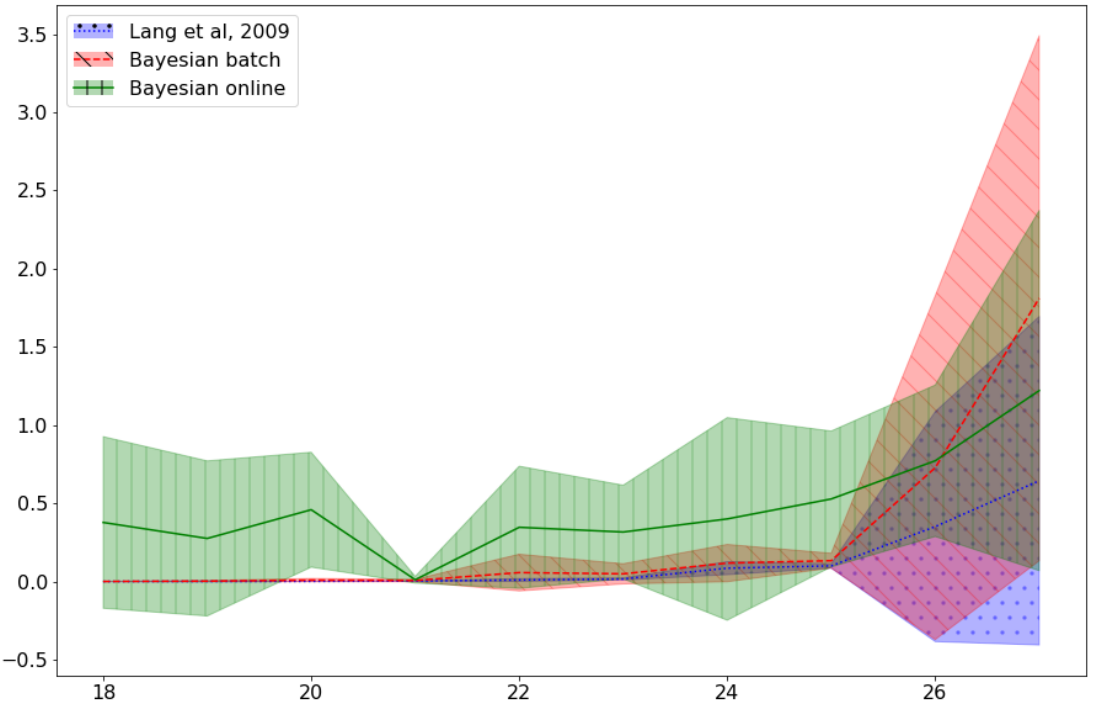

Figure 3.24: Accuracy (trajectory distance in px) of all three Multifit implementations compared depending on magnitude: frequentist batch (based on Lang et al, 2009) shown with dotted blue line and dotted background, Bayesian batch shown with dashed red line and leaning-striped background, Bayesian online shown with solid green line and vertically-striped background.

4

Conclusions and future work

Astronomy is the common theme connecting the two main topics of this dissertation: cross-matching of astronomical catalogs and parameter estimation of a moving point source model. Technological advances are causing explosion of data in all spheres of human activity and so is the case with astronomy. New generations of astronomical telescopes, most notably the upcoming Vera C. Rubin Observatory, are enabling "wide" and "deep" surveys, meaning: they observe large regions of the sky while at the same time capturing light from the most distant objects. This translates to increased pressure on data storage and processing requirements.

4.1 Positional cross-matching of astronomical catalogs

One of the most fundamental astronomical operations is positional cross-matching. Since (most of the) stars and galaxies do not have names, nor IDs, they have to be identified by their positions. When astronomers want to compare data obtained by different telescopes, in the modern times they issue queries to the respective "catalogs": large databases containing various measurements of objects in the sky observed by a particular telescope. The traditional paradigm astronomers employ is to select a subset of the data using a tool associated with a catalog, download the data to a workstation, and analyze the data locally.

However, lots of today's astronomical research, such as explaining the nature of Dark Energy, automatic classification of observed objects or searching for outliers, requires processing full catalogs and combining data from many of them. The issue then are sizes of these catalogs with billions of measurements of stars and galaxies in them. Fast and on-the-fly cross-matching, therefore, is a strong requirement for any such astronomical data analysis tool.

This work has shown that by organizing data using a distributed, zone-based approach, one can accomplish faster positional cross-matching of astronomical data than what existing solutions offer. Astronomical Extensions for Spark (AXS) is an astronomical tool developed as part of this thesis, based on Apache Spark but adding astronomy-specific functionalities and implementing the "Distributed Zones Algorithm". The algorithm consists of a distributed data organization scheme based on zones and an implementation of an "epsilon join" using a moving window. This scheme enables the system to process data in parallel, always retaining in memory just enough data minimally required for the cross-matching operation.

The extensive tests, both on a single machine and "in the cloud" have shown that this

system is capable of significantly better cross-matching performance than the existing ones.

4.2 Estimating parameters of a moving point-source model with sequential Bayesian updating

Measuring properties of faint objects from astronomical images is traditionally done by coadding images. Coaddition is a method of increasing signal-to-noise ratio by averaging (simply put) multiple images of the same object. The resulting "coadds" are brighter and crisper than the original images. However, coaddition cannot be applied to moving objects because the result are their blurred trails.

Multifit is a method for estimating parameters of models of astronomical objects based on their images using "forward modeling", i.e. simulating images of an object based on assumed model and its parameters, comparing the simulated images with the real ones and changing the parameters until the difference is minimized. In this way no coaddition is necessary and information from all images is preserved.

In this work, Multifit method is first implemented using methods of both Bayesian and "traditional" ("frequentist") statistics. Traditional statistics means maximizing the likelihood function using an optimization algorithm, as was done in [3]. Different optimization algorithms were compared in this thesis based on running time, accuracy and reliability. Unlike in [3], here it was found that Powell algorithm had the best accuracy when estimating the trajectory of a moving source, although it was the slowest. Accuracy was chosen over duration as being more important for this thesis and Powell algorithm was used for all other tests and implementations where a frequentist approach was needed (i.e. obtaining only MAP estimates and not the full posterior distribution). Jackknife method of estimating variance of the estimates has proved itself to be too computationally demanding to be useful for this use case. It was also shown that the difference between algorithms in χ^2_{DoF} for higher magnitudes tends to become less significant. The reason is that the objects are then much fainter and, for the highest magnitudes, even not visible by the naked eye. So the difference in residuals between the good and bad estimates becomes exponentially smaller for higher magnitudes. The resulting frequentist batch implementation code is packaged as an "LSST task" to be used as part of the data processing pipeline of the Vera C. Rubin Observatory.

The Multifit is also implemented with techniques of Bayesian statistics, i.e. MCMC, using Python's `Emcee` package. In addition to the maximum a-posteriori (MAP) estimates of parameter values, this procedure also gives out an estimate of full posterior probability distribution in the form of MCMC sample histograms. The posterior estimates can be obtained from histograms using different techniques. The method employed in this work is to approximate posterior with a mixture of Gaussians using variational inference (`BayesianGaussianMixture` class from Python's `scipy` package). However, for obtaining MAP estimates, the histogram medians were shown to be much more accurate than the estimated Gaussian mixture components. MAP estimates are further improved by running an optimization procedure (Powell algorithm), starting from the median values. It was demonstrated that accuracy of this final MAP estimate does not depend on the number of iterations of the MCMC procedure. The trajectory-estimation accuracy of this batch procedure was comparable to the frequentist batch implementation, except for the higher

magnitudes (26 and 27).

Finally, a new procedure called "Online Multifit" is developed using Bayesian sequential updating where the posterior is updated after each new data point (image). The positions of the posteriors (Gaussians) are corrected after each step with results from an additional run of an optimization procedure (Powell algorithm).

All three procedures have been tested on fake objects simulated on top of images from a real telescope using LSST simulation code. The tests have shown that the frequentist approach gives a very good trajectory-predicting accuracy, with the maximum of only 0.64 pixels for the magnitude of 27. The Bayesian batch approach has slightly worse accuracy than the frequentist approach, with the maximum of 1.8 pixels for the magnitude of 27 and with the accuracy of 0.7 pixels reached for magnitude of 26. The Bayesian online approach has again worse accuracy than the Bayesian batch procedure. That was, however, expected with approximations that are taken after each step. The accuracy is about 0.25-0.5 pixels for all magnitudes except for the 26 and 27 with accuracies there of 0.7 and 1.2 pixels, respectively.

The duration of the three methods are as follows. The Powell algorithm takes about 4 seconds to process 15 images in a batch mode. The Bayesian procedures are MCMC procedures trying to explore the full posterior and naturally take longer to accomplish that. Processing 15 images in batch mode with 150 walkers and 400 iterations takes about 172 minutes on average, which is obviously prohibitively high. However, batch implementation was only meant to be used as a guide post for online implementation's accuracy target and not to be used directly. Online implementation takes about 7 minutes on average when running with 400 iterations. This goes down to about 3 minutes with 150 iterations, but then the quality of the posterior estimate also goes down, so it is a tradeoff.

4.3 Further research directions

There are many different directions in which further research could be conducted.

For cross-matching of astronomical catalogs, and Astronomy Extensions for Spark, research has already begun on its next version called HIPSCat ([69]), which will be using astronomy-standard formats, and which will probably be used by LSST and NASA.

For Online Multifit, firstly, particle filters are an alternative approach whose application could be investigated. It is a vast field of many methods and approaches but the main idea is the same: maintaining a set of points that are describing the posterior distribution at all times. Compressing that information while maintaining an accurate estimate could be an interesting direction of research.

Secondly, behavior of the procedures in the presence of bright sources in the images could be investigated. Online procedure has the potential to overcome those kinds of situations, perhaps with a smart choice of priors and maintaining a set of alternative solutions.

Finally, performance of the Multifit procedures could be perhaps improved with parallel (multi-thread or multi-process) processing and by using Graphics Processing Units (GPUs).

A

Code listings

Listing A.1: Helper code for the likelihood function

```
import numpy as np
from lsst.geom import SpherePoint, degrees

class MovingSource():
  ''' A helper object describing a moving point source.
   ra0, dec0 - position at mjd0 in degrees
   sra, sdec - speeds in arcsec/day
   mag - magnitude '''
  def __init__(self, ra0, dec0, mjd0, sra, sdec, mag):
    self.ra0 = ra0
    self.dec0 = dec0
    self.mjd0 = mjd0
    self.sra = sra
    self.sdec = sdec
    self.mag = mag
    self.sra_degs = sra / 3600
    self.sdec_degs = sdec / 3600

  def position_at(self, mjd):
    ra = self.ra0 + (mjd - self.mjd0) * self.sra_degs
    dec = self.dec0 + (mjd - self.mjd0) * self.sdec_degs
    return (ra, dec)

  def sphere_point_at(self, mjd):
    pos = self.position_at(mjd)
    return SpherePoint(pos[0], pos[1], degrees)

  def __repr__(self):
    return f"Moving source: \n\tMagnitude: {self.mag}, \n\t"+\
      f"ra0: {self.ra0} deg, \n\tdec0: {self.dec0} deg, \n\t"+\
      f"MJD0: {self.mjd0}, \n\tSpeed in ra: {self.sra} "+\
      f"arsec/day, \n\tSpeed in dec: {self.sdec} arcsec/day"

def get_object(params, imagedata):
  if params is None:
```

```
    return None
  mag, x0, y0, vx, vy = params
  t0 = imagedata['t0']
  return MovingSource(imagedata['ra']+x0/3600, imagedata['dec']+y0/3600,
                      t0, vx, vy, mag)

def get_clone_empty_image(calexp):
  key = f"EMPTY{calexp.getWidth()}x{calexp.getHeight()}"
  if key in OBJ:
    OBJ[key].fill(0.)
    return OBJ[key]
  img = np.zeros(calexp.image.array.shape)
  OBJ[key] = img
  return img

def get_imgcutout(wcs, img, center_coord, SIZE=21):
  MARGIN = SIZE // 2
  extra_pixel = SIZE % 2
  pix = wcs.skyToPixel(SpherePoint(center_coord[0], center_coord[1],
                       degrees))
  h, w = img.shape
  if pix.x < 0 or pix.y < 0 or pix.x >= w or pix.y >= h:
    return None
  xl, yl = (round(pix.x - SIZE/2 + 0.5), round(pix.y - SIZE/2 + 0.5))
  xh, yh = (round(xl + SIZE), round(yl+SIZE))

  c = (img[max(0, yl) : min(h, yh),  max(0, xl) : min(w, xh)]).\
    astype(np.float64)
  if c.shape == (SIZE, SIZE):
      return c
  offs = [0, 0]
  if xh >= w:
      offs[0] = xh - w
  if yh >= h:
      offs[1] = yh - h
  newimg = np.zeros((SIZE, SIZE))
  newimg[offs[1]:, offs[0]:] = c
  return newimg

# caching the computation
APFLUXES = {}
def get_ap_flux(psf, cfr):
  import warnings
  warnings.filterwarnings("ignore", category=FutureWarning)
  key = f"{psf.computeImage().array.sum()}#{cfr}"
  if key in APFLUXES:
    return APFLUXES[key]
  apflux = psf.computeApertureFlux(cfr, psf.getAveragePosition())
  APFLUXES[key] = apflux
```

```
  return apflux

def calexp_contains_point(calexp, coord):
  p = calexp.getWcs().skyToPixel(SpherePoint(coord[0],
        coord[1], degrees))
    return p.x > 0 and p.y > 0 and p.x < calexp.\
      getBBox().getWidth() and p.y < calexp.getBBox().getHeight()
```

Listing A.2: Model generation code

```
import galsim
from lsst import geom

# The method copied from lsst.pipe.tasks.insertFakes.py and modified so that:
#     - we can add fakes also to variance images
#     - we can switch off noise.
def add_fake_sources(exposure, objects, empty_img=None, add_to_variance=True,
                     add_noise=True, calibFluxRadius=12.0, logger=None):
  """Add fake sources to the given exposure

  Parameters
  ----------
  exposure : `lsst.afw.image.exposure.exposure.ExposureF`
    The exposure into which the fake sources should be added
  objects : `typing.Iterator` [
              `tuple` [`lsst.geom.SpherePoint`,
                       `galsim.GSObject`]]
    An iterator of tuples that contains (or generates) locations and object
    surface brightness profiles to inject.
  add_to_variance : whether to also add noise to the variance image
  add_noise : if False will turn off Poisson noise
  calibFluxRadius : `float`, optional
    Aperture radius (in pixels) used to define the calibration for this
    exposure+catalog.  This is used to produce the correct instrumental
    fluxes within the radius.  The value should match that of the field
    defined in slot_CalibFlux_instFlux.
  logger : `lsst.log.log.log.Log` or `logging.Logger`, optional Logger.
  """
  exposure.mask.addMaskPlane("FAKE")
  bitmask = exposure.mask.getPlaneBitMask("FAKE")
  if logger:
    logger.info(f"Adding mask plane with bitmask {bitmask}")

  wcs = exposure.getWcs()
  psf = exposure.getPsf()

  bbox = exposure.getBBox()
  fullBounds = galsim.BoundsI(bbox.minX, bbox.maxX, bbox.minY, bbox.maxY)
  if empty_img is None:
    img = exposure.image.array
```

```
  else:
    img = empty_img
  _add_fake_sources(img, fullBounds, wcs, psf, objects, exposure=exposure,
                    bitmask=bitmask, add_noise=add_noise,
                    calibFluxRadius=calibFluxRadius, logger=logger)

def _add_fake_sources(imgArr, fullBounds, wcs, psf, objects, exposure=None,
                      bitmask=None, add_noise=True,
                      calibFluxRadius=12.0, logger=None):
  gsImg = galsim.Image(imgArr, bounds=fullBounds)

  pixScale = wcs.getPixelScale().asArcseconds()

  for spt, gsObj in objects:
    pt = wcs.skyToPixel(spt)
    posd = galsim.PositionD(pt.x, pt.y)
    posi = galsim.PositionI(pt.x // 1, pt.y // 1)
    if logger:
        print(f"Adding fake source at {pt}")

    mat = wcs.linearizePixelToSky(spt, geom.arcseconds).getMatrix()
    gsWCS = galsim.JacobianWCS(mat[0, 0], mat[0, 1], mat[1, 0], mat[1, 1])

    # This check is here because sometimes the WCS
    # is multivalued and objects that should not be
    # were being included.
    gsPixScale = np.sqrt(gsWCS.pixelArea())
    if gsPixScale < pixScale / 2 or gsPixScale > pixScale * 2:
      print("WCS check failed. Skipping the calexp...")
      continue

    try:
      psfArr = psf.computeKernelImage(pt).array
    except InvalidParameterError:
      # Try mapping to nearest point contained in bbox.
      bbox = exposure.getBBox()
      contained_pt = geom.Point2D(
        np.clip(pt.x, bbox.minX, bbox.maxX),
        np.clip(pt.y, bbox.minY, bbox.maxY)
      )
      if pt == contained_pt:  # no difference, so skip immediately
        print(f"Cannot compute Psf for object at {pt}; skipping")
        continue
      # otherwise, try again with new point
      try:
        psfArr = psf.computeKernelImage(contained_pt).array
      except InvalidParameterError:
        print(f"Cannot compute Psf for object at {pt}; skipping")
        continue
```

```
    # OPTIMIZED:
    apCorr = get_ap_flux(psf, calibFluxRadius)
    psfArr /= apCorr
    gsPSF = galsim.InterpolatedImage(galsim.Image(psfArr), wcs=gsWCS)

    conv = galsim.Convolve(gsObj, gsPSF)
    stampSize = conv.getGoodImageSize(gsWCS.minLinearScale())
    subBounds = galsim.BoundsI(posi).withBorder(stampSize // 2)
    subBounds &= fullBounds

    if subBounds.area() > 0:
      subImg = gsImg[subBounds]
      offset = posd - subBounds.true_center
      # Note, for calexp injection, pixel is already part of the PSF and
      # for coadd injection, it's incorrect to include the output pixel.
      # So for both cases, we draw using method='no_pixel'.

      if not add_noise:
        conv.drawImage(
          subImg,
          add_to_image=True,
          offset=offset,
          wcs=gsWCS,
          method='fft',
          )
      else:
        conv.drawImage(
          subImg,
          add_to_image=True,
          offset=offset,
          wcs=gsWCS,
          method='fft'
        )

      subBox = geom.Box2I(
        geom.Point2I(subBounds.xmin, subBounds.ymin),
        geom.Point2I(subBounds.xmax, subBounds.ymax)
      )
      if exposure and add_noise:
        exposure[subBox].mask.array |= bitmask
    else:
      raise ValueError("Subbounds area 0")

def insert_fakes_in_calexp(calexp, objs, empty_img=None,
                           add_to_variance=True, add_noise=True):
  # object speed in arcseconds per day
  spoints = []
  stars1 = []
```

```
  mjd = calexp.getMetadata()['MJD']
  wcs = calexp.getWcs()
  for obj in objs:
    if calexp_contains_point(calexp, obj.position_at(mjd)):
      spoint = obj.sphere_point_at(mjd)
      spoints.append(spoint)
      f = calexp.getPhotoCalib().magnitudeToInstFlux(obj.mag,
            wcs.skyToPixel(spoint))
      stars1.append(galsim.DeltaFunction().withFlux(f))

  arr_copy = None
  if add_to_variance:
    arr_copy = calexp.image.array.copy()

  add_fake_sources(calexp, [(spoints[i], stars1[i]) \
                       for i in range(len(spoints))],
                     empty_img=empty_img, add_to_variance=add_to_variance,
                     add_noise=add_noise)

  if add_to_variance:
    calexp.getVariance().array += (calexp.image.array - arr_copy)

def generate_model(obj, ra, dec, calexp_clone, image_size=100):
  ''' ra, dec should be in degrees '''
  empty_img = get_clone_empty_image(calexp_clone)
  insert_fakes_in_calexp(calexp_clone, [obj], empty_img=empty_img,
                         add_to_variance=False, add_noise=False)
  return get_imgcutout(calexp_clone.getWcs(), empty_img, (ra, dec),
                       SIZE=image_size)
```

Listing A.3: The likelihood function

```
def lnlike_fun(params, imagedata, calexp_clones):
  """Returns chi-square differences between pixels of simulated images
  (according to the provided parameters) and input images."""

  cutouts = imagedata['cutouts']
  IMGSIZE = cutouts[0].shape[0]
  psfs = imagedata['psfs']
  wcss = imagedata['wcss']
  variances = imagedata['variances']
  ras = imagedata['ras']
  decs = imagedata['decs']
  prepmasks = imagedata['prepmasks']
  ra = imagedata['ra']
  dec = imagedata['dec']

  # MODEL:
  # mag - magnitude
  # x0, y0 - object offset from the detected position at t=0 in arcsec
```

```
  # t0 - calculated from input time points; should be calculated
  #      so that it corresponds to the object's detected position
  #      (from the coadd) as much as possible
  # vx0, vy0 - object speed in x and y directions in arcsec per day
  mag, x0, y0, vx0, vy0 = params
  t0 = imagedata['t0']
  obj = MovingSource(ra+x0/3600, dec+y0/3600, t0, vx0, vy0, mag)

  total = None

  for _i in range(len(cutouts)):
    simimg = generate_model(obj, ras[_i], decs[_i], calexp_clones[_i],
                            image_size=IMGSIZE)

    diffimg = cutouts[_i] - simimg
    msk = prepmasks[_i]
    varimg = variances[_i]
    imgsum = diffimg
    imgsum[msk] = (diffimg[msk] / np.sqrt(varimg[msk]))
    imgsum[~msk] = np.inf
    imgsum = imgsum[np.isfinite(imgsum)].flatten()

    if total is not None:
      total = np.concatenate([total, imgsum])
    else:
      total = imgsum

  return total
```

Listing A.4: Frequentist batch Multifit implementation

```
class MultifitResult(object):
  def __init__(self, mov_src, res_norm, res_denorm, errors, errors_denorm,
               C, result, real_solution, real_obj, object_mjds):
    self.moving_source = mov_src
    self.x_normalized = res_norm
    self.x = res_denorm
    self.errors_normalized = errors
    self.errors = errors_denorm
    self.C = C
    self.result_object = result
    self.real_solution = real_solution
    self.real_source = real_obj
    self.object_mjds = object_mjds

  def get_trajectory_diff(self):
    if self.real_source is None:
      return [np.inf] * len(self.object_mjds)
    return trajectory_diff(self.moving_source, self.real_source,
                           self.object_mjds)
```

```
  def get_mag_diff(self):
    if self.real_source is None:
      return np.inf
    return self.moving_source.mag - self.real_source.mag

  def __getstate__(self):
    # omitting result_object because those can be HUGE
    return [self.moving_source, self.x_normalized, self.x,
            self.errors_normalized, self.errors, self.C,
            self.real_solution, self.real_source, self.object_mjds]

  def __setstate__(self, depickl):
    (mov_src, res_norm, res_denorm, errors, errors_denorm, C, real_params,
       realsrc, mjds) = depickl
    self.moving_source = mov_src
    self.x_normalized = res_norm
    self.x = res_denorm
    self.errors_normalized = errors
    self.errors = errors_denorm
    self.C = C
    self.real_solution = real_params
    self.real_source = realsrc
    self.object_mjds = mjds

  def __repr__(self):
    return f"ESTIMATED OBJECT: {self.moving_source}, " + \
      f"REAL OBJECT: {self.real_source}, " + \
      f"\nX: {self.x}, " + \
      f"\nX normalized: {self.x_normalized}, " + \
      f"\nStandard deviations: {self.errors}, " + \
      f"\nStandard deviations normalized: {self.errors_normalized},"+ \
      f"\nCovariance matrix: {self.C}, " + \
      f"\nReal parameters: {self.real_solution}, " + \
      f"\nAverage trajectory diff: {self.get_trajectory_diff()}, " + \
      f"\nMagnitude diff: {self.get_mag_diff()}, " + \
      "\nObject from the field 'result_object' ommited..."

def do_lmfit(imagedata, butler, real_solution=None, start_params=None,
             clones=None, in_collection=None,
             method='leastsq', bounds=(-np.inf, np.inf)):

  x0 = [20, 0, 0, 0, 0] if start_params is None else start_params

  maxs = [27.5, 2*MAX_PIXEL_SCALE, 2*MAX_PIXEL_SCALE,
          MAX_SPEED, MAX_SPEED]
  mins = [15, -2*MAX_PIXEL_SCALE, -2*MAX_PIXEL_SCALE,
          -MAX_SPEED, -MAX_SPEED]
```

```
  fit_params = Parameters()
  fit_params.add('mag', value=x0[0], max=maxs[0], min=mins[0])
  fit_params.add('x0', value=x0[1], max=maxs[1], min=mins[1])
  fit_params.add('y0', value=x0[2], max=maxs[2], min=mins[2])
  fit_params.add('vx', value=x0[3], max=maxs[3], min=mins[3])
  fit_params.add('vy', value=x0[4], max=maxs[4], min=mins[4])

  if clones is None:
    clones = get_calexp_clones(butler, imagedata['dataIds'], in_collection)

  def _lnlike_fun_wrap(params, imagedata, calexp_clones):
    ps = (params['mag'], params['x0'], params['y0'],
          params['vx'], params['vy'])
    return lnlike_fun(ps, imagedata, calexp_clones)

  result = minimize(_lnlike_fun_wrap, fit_params,
                    args=(imagedata, clones),
                    method=method, calc_covar=True, nan_policy='omit')

  if not hasattr(result, 'covar') or result.covar is None:
    raise LinAlgError("Covariance not found")

  C = result.covar
  diag = np.diag(result.covar)
  if (diag < 0).any():
    raise LinAlgError("Negative variances")
  errors = np.sqrt(diag)

  resmag, resx0, resy0, resvx, resvy = (result.params['mag'].value,
                                        result.params['x0'].value,
                                        result.params['y0'].value,
                                        result.params['vx'].value,
                                        result.params['vy'].value)

  res = np.array([resmag, resx0, resy0, resvx, resvy])

  ms = get_object(res, imagedata)

  return MultifitResult(ms, res, res, errors, errors,
                        C, result, real_solution,
                        get_object(real_solution, imagedata),
                        imagedata['times'])

def do_multifit(imagedata, butler, real_solution=None, start_params=None,
                clones=None, in_collection=None, method='powell',
                bounds=(-np.inf, np.inf), verbose=0):
  if method in ["cg", "lbfgsb", "bfgs", "tnc", "trust-constr", "slsqp",
                "shgo", "nelder", "differential_evolution", "cobyla",
                "powell"]:
```

```
  return do_lmfit(imagedata, butler, real_solution, start_params,
                  clones, in_collection, method, bounds)

start_mag = 20
errors = None
C = None

x0 = [start_mag, 0, 0, 0, 0] if start_params is None else start_params

if clones is None:
  clones = get_calexp_clones(butler, imagedata['dataIds'], in_collection)

if method == 'leastsq':
  result = leastsq(lnlike_fun, x0, args=(imagedata, clones),
                   full_output=1)

  res = result[0]

  fjac = result[2]['fjac']
  ipvt = result[2]['ipvt']

  n = len(x0)
  perm = np.mat(np.take(np.eye(n), ipvt-1, axis=1))

  r = np.mat(np.triu(fjac[:, :n]))

  jac = np.dot(r, perm)
elif method in ['trf', 'dogbox', 'lm']:
  result = least_squares(lnlike_fun, x0, args=(imagedata, clones,
                         False, None, fast_model),
                         method=method, verbose=verbose, max_nfev=100)
  res = result.x

  jac = result.jac
else:
  raise NotImplementedError("Method not supported")

# With Jacobian J (result.jac): J^T J is a Gauss-Newton
# approximation of the Hessian of the cost function.
# Hessian is an inverse of covariance matrix C
if jac is None:
  raise LinAlgError("Jacobian not found")

hess = 2 * jac.T @ jac
C = np.linalg.inv(hess)  # this can raise an Exception

errors = np.sqrt(np.diagonal(C))

resmag, resx0, resy0, resvx, resvy = res
```

```
  ms = get_object(res, imagedata)

  return MultifitResult(ms, res, res, errors, errors,
                        C, result, real_solution,
                        get_object(real_solution, imagedata),
                        imagedata['times'])
```

Listing A.5: Bayesian (MCMC) batch and online Multifit implementation

```
import datetime
import emcee
import numpy as np
import pickle
from multifit_lib import get_calexp_clones, lnlike_fun
from multifit_lib import generate_random_params

def run_emcee(imagedata, OBJID, CUTID, butler, in_collection, IMGID=None,
              kde=None, nwalkers=150, move=None, warmup_iters=100,
              main_iters=500, save_every=100, ndim=5, filename=None,
              verbose=True):
  clones = get_calexp_clones(butler, imagedata['dataIds'], in_collection)

  emcee_move = emcee.moves.DEMove()
  if move is not None:
    if move == "de":
      emcee_move = emcee.moves.DEMove()
    elif move == "kde":
      emcee_move = emcee.moves.KDEMove()
    elif move == "desnooker":
      emcee_move = emcee.moves.DESnookerMove()

  pixel_scale = 4.6831063024143785e-05
  MAX_DIST = 5 * pixel_scale * 3600  # 5 pixels in arcsec
  MAX_SPD = pixel_scale * 3600 / 10  # 1 pixel in 10 days
  SPD_STDEV = MAX_SPD / 5
  MIN_MAG = 15
  MAX_MAG = 30

  def lnprob(params, imagedata, clones):
    mag, x0, y0, vx, vy = params
    dist_from_detection = np.sqrt(x0 ** 2 + y0 ** 2)
    if dist_from_detection > MAX_DIST:
      return -np.inf
    if mag <= MIN_MAG or mag >= MAX_MAG:
      return -np.inf
    priorval = 0
    if kde is not None:
      priorval = kde.score([params])
    # speed prior
```

```
    priorval += np.sum(scipy.stats.norm.logpdf([vx, vy], 0, SPD_STDEV))
    if not np.isfinite(priorval):
      return -np.inf
    try:
      chi2 = lnlike_fun(params, imagedata, clones)
      chi2 = np.sum(chi2 ** 2)
      return -chi2 + priorval
    except ValueError:
      return -np.inf

start_params = generate_random_params(nwalkers, priors=priors)

sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob,
                                args=[imagedata, clones],
                                moves=[(emcee_move, 1)])
state = start_params
if warmup_iters > 0:
    if verbose:
      print(f"Running {warmup_iters} warmup iterations...")
    state = sampler.run_mcmc(start_params, warmup_iters,
                             progress='notebook' if verbose else False)
    if verbose:
      print(f"Mean acceptance fraction: "+\
            f"{np.mean(sampler.acceptance_fraction):.3f}")
    sampler.reset()

if filename is None:
    datestr = datetime.date.today().strftime('%Y%m%d')
    imgidstr = "" if IMGID is None else f"_{IMGID}"
    filename = f"flatchain_{datestr}_{OBJID}_{CUTID}{imgidstr}"+\
               f"_{main_iters}_{warmup_iters}.pickle"

if verbose:
    print(f"Running {main_iters} main iterations. "+\
          f"Saving flatchain to {filename} every {save_every} iterations.")
for i in range(int(main_iters/save_every)+1):
    # if state is None, then resume where it left off the last time it ran
    st = state if i == 0 else None
    iters = save_every
    if i == int(main_iters/save_every):
      iters = main_iters % save_every
    if iters > 0:
      state = sampler.run_mcmc(st, iters, progress='notebook')
      with open(filename, 'wb') as f:
        if verbose:
            print("Saving")
        samples = sampler.chain
        pickle.dump(samples, f)
```

```
      if verbose:
        print(f"Mean acceptance fraction: "+\
              f"{np.mean(sampler.acceptance_fraction):.3f}")
      try:
        print(f"Autocorrelation time: {sampler.get_autocorr_time()}")
      except emcee.autocorr.AutocorrError:
        pass
  return samples, filename, sampler
```

Listing A.6: A helper object used for modifying means of a Gaussian mixture

```
import scipy
class ScipyWrapper():
  def __init__(self, kde, newmean):
    self.means = kde.means_
    self.covs = kde.covariances_
    self.weights = kde.weights_
    maxind = np.argmax(self.weights)
    self.means[maxind] = newmean

  def score_samples(self, samples):
    return self.score(samples)

  def score(self, samples):
    ret = 0
    for i, w in enumerate(self.weights):
      ret += scipy.stats.multivariate_normal.\
        pdf(samples, mean=self.means[i], cov=self.covs[i],
            allow_singular=True) * w
    return np.log(ret)

  def __getstate__(self):
    return [self.means, self.covs, self.weights]

  def __setstate__(self, depickl):
    means, covs, weights = depickl
    self.means = means
    self.covs = covs
    self.weights = weights
```

BIBLIOGRAPHY

[1] Jim Gray, María A. Nieto-Santisteban, and Alexander S. Szalay. The zones algorithm for finding points-near-a-point or cross-matching spatial datasets. *CoRR*, abs/cs/0701171, 2007.

[2] J. A. Tyson, C. Roat, J. Bosch, and D. Wittman. Lsst and the dark sector: Image processing challenges. 2008.

[3] Dustin Lang, David W. Hogg, Sebastian Jester, and Hans-Walter Rix. Measuring the undetectable: Proper motions and parallaxes of very faint sources. *The Astronomical Journal*, 137(5):4400, 2009.

[4] Jonathan Goodman and Jonathan Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

[5] Daniel Foreman-Mackey, Will Farr, Manodeep Sinha, Anne Archibald, David Hogg, Jeremy Sanders, Joe Zuntz, Peter Williams, Andrew Nelson, Miguel de Val-Borro, Tobias Erhardt, Ilya Pashchenko, and Oriol Pla. emcee v3: A python ensemble sampling toolkit for affine-invariant MCMC. *Journal of Open Source Software*, 4(43):1864, nov 2019.

[6] Keaton J Bell. The search for planet and planetesimal transits of white dwarfs with the zwicky transient facility. *Proceedings of the International Astronomical Union*, 15(S357):37–40, 2019.

[7] Katelyn Breivik, Andrew J Connolly, KE Ford, Mario Jurić, Rachel Mandelbaum, Adam A Miller, Dara Norman, Knut Olsen, William O'Mullane, Adrian Price-Whelan, et al. From data to software to science with the rubin observatory lsst. *arXiv preprint arXiv:2208.02781*, 2022.

[8] MF Skrutskie, RM Cutri, R Stiening, MD Weinberg, S Schneider, JM Carpenter, Capps Beichman, R Capps, T Chester, J Elias, et al. The two micron all sky survey (2mass). *The Astronomical Journal*, 131(2):1163, 2006.

[9] Donald G York, J Adelman, John E Anderson Jr, Scott F Anderson, James Annis, Neta A Bahcall, JA Bakken, Robert Barkhouser, Steven Bastian, Eileen Berman, et al. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579, 2000.

[10] Gaia Collaboration and et al. The Gaia mission. *aap*, 595:A1, November 2016.

[11] Gaia Collaboration and et al. Gaia Data Release 3: Summary of the content and survey properties. *arXiv e-prints*, page arXiv:2208.00211, July 2022.

[12] LSST Science Collaboration, P. A. Abell, J. Allison, S. F. Anderson, J. R. Andrew, J. R. P. Angel, L. Armus, D. Arnett, S. J. Asztalos, T. S. Axelrod, and et al. LSST Science Book, Version 2.0. *ArXiv e-prints*, December 2009.

[13] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[14] Gregory M. Green, Edward F. Schlafly, Douglas P. Finkbeiner, Hans-Walter Rix, Nicolas Martin, William Burgett, Peter W. Draper, Heather Flewelling, Klaus Hodapp, Nicholas Kaiser, Rolf Peter Kudritzki, Eugene Magnier, Nigel Metcalfe, Paul Price, John Tonry, and Richard Wainscoat. A Three-dimensional Map of Milky Way Dust. *APJ*, 810:25, September 2015.

[15] T. Budavári and A. Basu. Probabilistic cross-identification in crowded fields as an assignment problem. *AJ*, 152(4):86, 2016.

[16] Joseph J. Mohr, Robert Armstrong, Emmanuel Bertin, Greg Daues, Shantanu Desai, Michelle Gower, Robert Gruendl, William Hanlon, Nikolay Kuropatkin, Huan Lin, John Marriner, Donald Petravic, Ignacio Sevilla, Molly Swanson, Todd Tomashek, Douglas Tucker, and Brian Yanny. The dark energy survey data processing and calibration system, 2012.

[17] W. Romanishin. An introduction to astronomical photometry using ccds. `http://www.physics.csbsju.edu/370/photometry/manuals/OU.edu_CCD_photometry_wrccd06.pdf`, 2006.

[18] Barak Zackay and Eran O. Ofek. How to coadd images? i. optimal source detection and photometry using ensembles of images. 2015.

[19] Ivezić, A. J. Connolly, and M. Jurić. Everything we'd like to do with lsst data, but we don't know (yet) how. 2016.

[20] Aaron Roodman Christopher P. Davis, Jamie Rodriguez. Wavefront-based psf estimation, 2016.

[21] James Annis, Marcelle Soares-Santos, Michael A. Strauss, Andrew C. Becker, Scott Dodelson, Xiaohui Fan, James E. Gunn, Jiangang Hao, Željko Ivezić, Sebastian Jester, Linhua Jiang, David E. Johnston, Jeffrey M. Kubo, Hubert Lampeitl, Huan Lin, Robert H. Lupton, Gajus Miknaitis, Hee-Jong Seo, Melanie Simet, and Brian Yanny. The sloan digital sky survey coadd: 275 deg2 of deep sloan digital sky survey imaging on stripe 82. *The Astrophysical Journal*, 794(2):120, 2014.

[22] Barak Zackay and Eran O. Ofek. How to coadd images? ii. a coaddition image that is optimal for any purpose in the background dominated noise limit. 2015.

[23] Petar Zečević, Colin T Slater, Mario Jurić, Andrew J Connolly, Sven Lončarić, Eric C Bellm, V Zach Golkhou, and Krzysztof Suberlak. Axs: A framework for fast astronomical data processing based on apache spark. *The Astronomical Journal*, 158(1):37, 2019.

[24] M. Jurić. Large Survey Database: A Distributed Framework for Storage and Analysis of Large Datasets. In *American Astronomical Society Meeting Abstracts #217*, volume 43 of *Bulletin of the American Astronomical Society*, page 433.19, January 2011.

[25] M. Jurić. LSD: Large Survey Database framework. Astrophysics Source Code Library, September 2012.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[28] K. M. Górski and et al. Analysis issues for large CMB data sets. In A. J. Banday, R. K. Sheth, and L. N. da Costa, editors, *Evolution of Large Scale Structure : From Recombination to Garching*, page 37, January 1999.

[29] ISO. *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. ISO, December 2011.

[30] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130-136. Citeseer, 2015.

[31] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[32] M. T. Soumagnac and E. O. Ofek. catsHTM: A Tool for Fast Accessing and Cross-matching Large Astronomical Catalogs. *Publications of the Astronomical Society of the Pacific*, 130(7):075002, July 2018.

[33] M. Brahem, K. Zeitouni, and L. Yeh. Astroide: A unified astronomical big data processing engine over spark. *IEEE Transactions on Big Data*, pages 1–1, 2018.

[34] P. M. Marrese, S. Marinoni, M. Fabrizio, and G. Giuffrida. Gaia Data Release 1. Cross-match with external catalogues. Algorithm and results. *aap*, 607:A105, November 2017.

[35] María A Nieto-Santisteban, Aniruddha R Thakar, and Alexander S Szalay. Cross-matching very large datasets. In *National Science and Technology Council (NSTC) NASA Conference*, 2007.

[36] X. Jia, Q. Luo, and D. Fan. Cross-matching large astronomical catalogs on heterogeneous clusters. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 617–624, Dec 2015.

[37] László Dobos, Tamas Budavari, Nolan Li, Alexander S. Szalay, and István Csabai. Skyquery: An implementation of a parallel probabilistic join engine for cross-identification of multiple astronomical databases. *CoRR*, abs/1206.5021, 2012.

[38] Peter Z. Kunszt, Alexander S. Szalay, and Aniruddha R. Thakar. The hierarchical triangular mesh. In Anthony J. Banday, Saleem Zaroubi, and Matthias Bartelmann, editors, *Mining the Sky*, pages 631–637, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[39] Jim Gray, Alexander S. Szalay, Aniruddha R. Thakar, Gyorgy Fekete, William O'Mullane, María A. Nieto-Santisteban, Gerd Heber, and Arnold H. Rots. There goes the neighborhood: Relational algebra for spatial data search. *CoRR*, cs.DB/0408031, 2004.

[40] J. Gray, A. S. Szalay, A. R. Thakar, P. Z. Kunszt, C. Stoughton, D. Slutz, and J. vandenBerg. Data Mining the SDSS SkyServer Database. *eprint arXiv:cs/0202014*, February 2002.

[41] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 892–903, March 2010.

[42] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[43] Nasim Ahmed, Andre LC Barczak, Teo Susnjak, and Mohammed A Rashid. A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibench. *Journal of Big Data*, 7(1):1–18, 2020.

[44]Parquet Project. Apache parquet documentation. 2018.

[45] Julien Peloton, Christian Arnault, and Stéphane Plaszczynski. Analyzing astronomical data with apache spark. 04 2018.

[46] Steven Stetzler, Mario Jurić, Kyle Boone, Andrew Connolly, Colin T Slater, and Petar Zečević. The astronomy commons platform: A deployable cloud-based analysis platform for astronomy. *The Astronomical Journal*, 164(2):68, 2022.

[47] L. Miller; T. D. Kitching; C. Heymans; A. F. Heavens; L. Van Waerbeke. Bayesian galaxy shape measurement for weak lensing surveys – i. methodology and a fast-fitting algorithm. *Monthly Notices of the Royal Astronomical Society*, 382, 2007.

[48] James Bosch. Modeling techniques for measuring galaxy properties in multi-epoch surveys, 2011.

[49] Catherine Heymans, Ludovic Van Waerbeke, Lance Miller, Thomas Erben, Hendrik Hildebrandt, Henk Hoekstra, Thomas D. Kitching, Yannick Mellier, Patrick Simon, Christopher Bonnett, Jean Coupon, Liping Fu, Joachim Harnois-Déraps, Michael J. Hudson, Martin Kilbinger, Koenraadad Kuijken, Barnaby Rowe, Tim Schrabback, Elisabetta Semboloni, Edo van Uitert, Sanaz Vafaei, and Malin Velander. Cfhtlens: the canada–france–hawaii telescope lensing survey. *Monthly Notices of the Royal Astronomical Society*, 427, 11 2012.

[50] M. Jarvis, E. Sheldon, J. Zuntz, T. Kacprzak, S. L. Bridle, A. Amara, R. Armstrong, M. R. Becker, G. M. Bernstein, C. Bonnett, C. Chang, R. Das, J. P. Dietrich, A. Drlica-Wagner, T. F. Eifler, C. Gangkofner, D. Gruen, M. Hirsch, E. M. Huff, B. Jain, S. Kent, D. Kirk, N. MacCrann, P. Melchior, A. A. Plazas, A. Refregier, B. Rowe, E. S. Rykoff, S. Samuroff, C. Sánchez, E. Suchyta, M. A. Troxel, V. Vikram, T. Abbott, F. B. Abdalla, S. Allam, J. Annis, A. Benoit-Lévy, E. Bertin, D. Brooks, E. Buckley-Geer, D. L. Burke, D. Capozzi, A. Carnero Rosell, M. Carrasco Kind, J. Carretero, F. J. Castander, J. Clampitt, M. Crocce, C. E. Cunha, C. B. D'Andrea, L. N. da Costa, D. L. DePoy, S. Desai, H. T. Diehl, P. Doel, A. Fausti Neto, B. Flaugher, P. Fosalba, J. Frieman, E. Gaztanaga, D. W. Gerdes, R. A. Gruendl, G. Gutierrez, K. Honscheid, D. J. James, K. Kuehn, N. Kuropatkin, O. Lahav, T. S. Li, M. Lima, M. March, P. Martini, R. Miquel, J. J. Mohr, E. Neilsen, B. Nord, R. Ogando, K. Reil, A. K. Romer, A. Roodman, M. Sako, E. Sanchez, V. Scarpine, M. Schubnell, I. Sevilla-Noarbe, R. C. Smith, M. Soares-Santos, F. Sobreira, M. E. C. Swanson, G. Tarle, J. Thaler, D. Thomas, A. R. Walker, and R. H. Wechsler. The des science verification weak lensing shear catalogues. *Monthly Notices of the Royal Astronomical Society*, 460(2):2245–2281, 2016.

[51] Željko Ivezić, Andrew J Connolly, Jacob T VanderPlas, and Alexander Gray. *Statistics, data mining, and machine learning in astronomy: a practical Python guide for the analysis of survey data*, volume 1. Princeton University Press, 2014.

[52] Ll G Chambers. Practical methods of optimization (2nd edn), by r. fletcher. pp. 436.£ 34.95. 2000. isbn 0 471 49463 1 (wiley). *The Mathematical Gazette*, 85(504):562–563, 2001.

[53] J Nocedal and SJ Wright. Numerical optimization 2nd edition springer. *New York*, 2006.

[54] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical Analysis: Proceedings of the Biennial Conference Held at Dundee, June 28–July 1, 1977*, pages 105–116. Springer, 2006.

[55] Mary Ann Branch, Thomas F Coleman, and Yuying Li. A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems. *SIAM Journal on Scientific Computing*, 21(1):1–23, 1999.

[56] C Voglis and IE Lagaris. A rectangular trust region dogleg approach for unconstrained and bound constrained nonlinear optimization. In *WSEAS International Conference on Applied Mathematics*, volume 7, pages 9780429081385–138, 2004.

[57] Michael JD Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

[58] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[59] Malvin H Kalos and Paula A Whitlock. *Monte carlo methods*. John Wiley & Sons, 2009.

[60] Jasper Vivian Wall and Charles R Jenkins. *Practical statistics for astronomers*. Cambridge University Press, 2012.

[61] Phil Gregory. *Bayesian Logical Data Analysis for the Physical Sciences: A Comparative Approach with Mathematica® Support*. Cambridge University Press, 2005.

[62] A Sokal. Monte carlo methods in statistical mechanics: foundations and new algorithms. In *Functional integration*, pages 131–192. Springer, 1997.

[63] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[64] John Kruschke. *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan*. Academic Press, 2014.

[65] B. Farr and W. M. Farr. kombine: a kernel-density-based, embarrassingly parallel ensemble sampler. 2015. in prep.

[66] Eric D Feigelson and G Jogesh Babu. *Modern statistical methods for astronomy: with R applications*. Cambridge University Press, 2012.

[67]Larry Wasserman. All of statistics: a concise course in statistical inference. 2004.

[68] Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman. emcee: The MCMC Hammer. *Publications of the Astronomical Society of the Pacific*, 125(925):306, March 2013.

[69] Samuel Wyatt, Mario Juric, Steven Stetzler, Colin Slater, Andrew Connolly, Melissa DeLucchi, Max West, Rachel Mandelbaum, Jeremy Kubica, George Amvrosiadis, et al. Lincc—hipscat and lsd2: Joint distributed analysis of lsst-scale datasets. In *American Astronomical Society Meeting Abstracts*, volume 55, pages 105–06, 2023.

CURRICULUM VITAE

Petar Zečević was born in Koprivnica, Yugoslavia in 1977. He finished Mathematical Gymnasium "Lucian Vranjanin" in 1995 and obtained his Diploma of Master of Engineering degree in electrical engineering from the Faculty of Electrical Engineering and Computing at University of Zagreb in 2002 with the topic "Security of computer systems in Java environment". He worked at the company WBS-Tech d.o.o. as a Java developer since 2000 until 2002 and then since 2002 in the company SV Group d.o.o. as a Java Architect, Team Leader, and Director of Technologies. His book "Spark in Action" (Manning) was published in 2016. He enrolled in a PhD program at Faculty of Electrical Engineering and Computing at University of Zagreb in 2017. Since 2022, he is employed at "Poslovna inteligencija" company as a Senior Principal Consultant. Petar is a regular speaker at conferences where he talks about Big Data processing and the related technologies.

The full list of publications is given below.

FULL LIST OF PUBLICATIONS

JOURNAL PUBLICATIONS:

1. P. Zečević, C. T. Slater, M. Jurić, A. J. Connolly, S. Lončarić, E. C. Bellm, V. Z. Golkhou and K. Suberlak. AXS: A Framework for Fast Astronomical Data Processing Based on Apache Spark. *The Astronomical Journal*, 158:37 (14pp), 2019 July, IF: 5.491.

2. S. Stetzler, M. Jurić, K. Boone, A. Connolly, C. T. Slater, and P. Zečević. The Astronomy Commons Platform: A Deployable Cloud-based Analysis Platform for Astronomy. *The Astronomical Journal*, 164:68 (18pp), 2022 August, IF: 5.491.

CONFERENCE PUBLICATIONS:

1. P. Zečević, C. T. Slater, M. Jurić, and S. Lončarić. AXS: Making end-user petascale analyses possible, scalable, and usable. *Astronomical Data Analysis Software and Systems XXVIII*. University of Maryland, College Park, MD, USA, 11-15 November 2018.

BOOKS:

1.P. Zečević, M. Bonaći. Spark in Action. Manning, 2016.

ŽIVOTOPIS

Petar Zečević rođen je u Koprivnici, Jugoslavija, 1977. godine. Matematičku gimnaziju "Lucijan Vranjanin" završio je 1995. godine, a diplomirao na Fakultetu elektrotehnike i računarstva, smjer Računarstvo, 2002. godine s temom "Sigurnost računalnih sustava u Java okruženju". Od 2000. do 2002. godine zaposlen je u tvrtki WBS-Tech d.o.o. kao Java razvonji inženjer, a od 2002. godine u tvrtki SV Group d.o.o. kao Java arhitekt, voditelj tima te direktor tehnologija. 2016. godine izlazi njegova knjiga "Spark in Action" (Manning). 2017. godine upisuje doktorat na Fakultetu elektrotehnike i računarstva, a od 2022. godine zaposlen je u tvrtki Poslovna inteligencija. Petar je redoviti govornik na konferencijama vezanim za obradu velikih količina podataka i povezane tehnologije.

COLOPHON

This document was typeset and inspired by the typographical look-and-feel `classicthesis` developed by André Miede, which was based on Robert Bringhurst's book on typography *The Elements of Typographic Style*, and by the `FERElemental` developed by Ivan Marković whose design was based on `FERBook` developed by Jadranko Matuško.