Rekonstruktion des Magnetfeldes der Milchstraße
Reconstructing the Galactic Magnetic Field

Theo Steininger

München 2018

Rekonstruktion des Magnetfeldes der Milchstraße
Reconstructing the Galactic Magnetic Field

Theo Steininger

Dissertation
an der Fakultät für Physik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Theo Steininger
aus Landau a.d. Isar

München, den 26.02.2018

Erstgutachter: PD Dr. Torsten A. Enßlin
Zweitgutachter: Prof. Dr. Jochen Weller
Tag der mündlichen Prüfung: 09.05.2018

Für Hannah, Magda und Gabriel

Contents

List of Figures

List of Tables

Zusammenfassung

Diese Dissertation befasst sich mit der Rekonstruktion des Magnetfeldes der Milchstraße (GMF für *Galaktisches Magnetfeld*). Eine genaue Beschreibung des Magnetfeldes ist für mehrere Fragestellungen der Astrophysik relevant. Erstens spielt es eine wichtige Rolle dabei, wie sich die Struktur der Milchstraße entwickelt, da die Ströme von interstellarem Gas und kosmischer Strahlung durch das GMF abgelenkt werden. Zweitens stört es die Messung und Analyse von Strahlung extra-galaktischer Quellen. Drittens lenkt es ultra-hoch-energetische kosmische Strahung (UHECR) derartig stark ab, dass die Zuordnung von gemessenen UHECR zu potentiellen Quellen nicht ohne Korrekturrechnung möglich ist. Viertens kann mit dem GMF ein kosmischer Dynamo-Prozess inklusive dessen innerer Strukturen studiert werden. Im Gegensatz zum GMF ist bei Sternen und Planeten nur das äußere Magnetfeld zugänglich und messbar.

So großen Einfluss das GMF auf eine Vielzahl von Effekten hat, genauso schwer ist es auch zu ermitteln. Der Grund dafür ist, dass das Magnetfeld nicht direkt, sondern nur durch seinen Einfluss auf verschiedene physikalische Observablen messbar ist. Messungen dieser Observablen liefern für eine konkrete Sichtlinie ihren gesamt-akkumulierten Wert. Aufgrund der festen Position des Sonnensystems in der Milchstraße ist es daher eine Herausforderung der gemessenen Wirkung des Magnetfelds einer räumlichen Tiefe zuzuordnen.

Als Informationsquelle dienen vor allem Messungen der Intensität und Polarisation von Radio- und Mikrowellen, sowohl für den gesamten Himmel, als auch für einzelne Sterne, deren Position im Raum bekannt ist. Durch die Betrachtung der zugrunde liegenden physikalischen Prozesse wie Synchrotronemission und Faraday Rotation kann auf das GMF rückgeschlossen werden. Voraussetzung dafür sind jedoch dreidimensionale Dichte-Karten anderer Konstituenten der Milchstraße, beispielsweise der thermischen Elektronen oder des interstellaren Staubes. Für die Erstellung dieser Hilfskarten sind physikalische Prozesse wie Dispersion und Staubabsorption von entscheidender Bedeutung.

Um das GMF anhand der vorhandenen Messdaten zu rekonstruieren, gibt es im Wesentlichen zwei Herangehensweisen. Zum einen benutzt man den phänomenologischen Ansatz *parametrischer Magnetfeld-Modelle*. Dabei wird die Struktur des Magnetfeldes durch analytische Formeln mit einer begrenzten Anzahl von Parametern festgelegt. Diese Modelle beinhalten die generelle Morphologie des Magnetfeldes, wie etwa Galaxie-Arme und Feld-Umkehrungen, aber auch lokale Charakteristika wie Nebel in der Nachbarschaft des Sonnensystems. Gegeben einem Satz Messdaten versucht man nun, jene Modellparameter zu finden, die eine möglichst gute Übereinstimmung mit den Observablen ergeben. Zu diesem Zweck wurde im Rahmen dieser Doktorarbeit Imagine, die *Interstellar MAGnetic field INference Engine*, entwickelt. Aufgrund

der verhältnismäßig geringen Anzahl an Parametern ist eine Parameteranpassung auch mit robusten all-sky maps möglich, auch wenn diese keine Tiefen-Information enthalten. Allerdings gibt es bei der Herangehensweise über parametrische Modelle das Problem der Beliebigkeit: es gibt eine Vielzahl an Modellen verschiedenster Komplexität, die sich darüber hinaus häufig gegenseitig widersprechen. In der Vergangenheit wurden dann meist auch noch die Unsicherheit der Parameter-Rekonstruktionen unterschätzt. Im Gegensatz dazu ermöglicht eine rigorose Bayes'sche Analyse, beispielsweise mit dem in dieser Doktorarbeit entwickelten IMAGINE, eine verlässliche Bestimmung der Modellparameter.
Neben parametrischen Modellen kann das GMF auch über einen nicht-parametrischen Ansatz rekonstruiert werden. Dabei hat jedes Raumvoxel zwei unabhängige Freiheitsgrade für das Magnetfeld. Diese Art der Rekonstruktion stellt deutlich höhere Ansprüche an die Datenmenge und -qualität, die Algorithmik, und die Rechenkapazität. Aufgrund der hohen Anzahl an Freiheitsgraden werden Messdaten benötigt, die direkte (Parallax-Messungen) oder indirekte (über das Hertzsprung Russel Diagramm) Tiefeninformation beinhalten. Zudem sind starke Prior für jene Raumbereiche notwendig, die von den Daten nur schwach abgedeckt werden. Einfache Bayes'sche Methoden reichen hierfür nicht mehr aus. Vielmehr ist nun Informationsfeldtheorie (IFT) nötig, um die verschiedenen Informationsquellen korrekt zu kombinieren, und verlässliche Unsicherheiten zu erhalten. Für diese Aufgabe ist das PYTHON Framework NIFTY (Numerical Information Field Theory) prädestiniert. In seiner ersten Release-Version war NIFTY jedoch noch nicht für Magnetfeldrekonstruktionen und die benötigten Größenordnungen geeignet. Um die Datenmengen verarbeiten zu können wurde daher zunächst D2O als eigenständiges Werkzeug für Daten-Parallelisierung entwickelt. Damit kann parallelisierter Code entwickelt werden, ohne das die eigentliche Entwicklungsarbeit behindert wird. Da im Grunde alle numerischen Disziplinen mit großen Datensätzen, die sich nicht in Teilmengen zerlegen lassen davon profitieren können, wurde D2O als eigenständiges Paket veröffentlicht.
Darüber hinaus wurde NIFTY so umfassend in seinem Funktionsumfang und seiner Struktur überarbeitet, sodass nun unter anderem auch hochaufgelöste Magnetfeldrekonstruktionen durchgeführt werden können. Außerdem ist es jetzt mit NIFTY auch möglich Karten der thermischen Elektronendichte und des interstellaren Staubes auf Basis neuer und gleichzeitig auch sehr großer Datensätze zu erstellen. Damit wurde der Weg zu einer nicht-parametrischen Rekonstruktionen des GMF geebnet.

Summary

This thesis deals with the reconstruction of the magnetic field of the Milky Way (GMF for *Galactic Magnetic Field*). A detailed description of the magnetic field is relevant for several problems in astrophysics. First, it plays an important role in how the structure of the Milky Way develops as the currents of interstellar gas and cosmic rays are deflected by the GMF. Second, it interferes with the measurement and analysis of radiation from extra-galactic sources. Third, it deflects ultra-high energetic cosmic rays (UHECR) to such an extent that the assignment of measured UHECR to potential sources is not possible without a correcting calculations. Fourth, the GMF can be used to study a cosmic dynamo process including its internal structures. In contrast to the GMF, normally only the outer magnetic field of stars and planets is accessible and measurable.

As much as the GMF has an impact on a variety of effects, it is just as difficult to determine. The reason for this is that the magnetic field cannot be measured directly, but only by its influence on various physical observables. Measurements of these observables yield their total accumulated value for a certain line of sight. Due to the fixed position of the solar system in the Milky Way, it is therefore a challenge to map the measured effect of the magnetic field to a spatial depth.

Measurements of the intensity and polarization of radio and microwaves, both for the entire sky and for individual stars whose position in space is known, serve as a source of information. Based on physical processes such as synchrotron emission and Faraday rotation, the GMF can be deduced. However, this requires three-dimensional density maps of other constituents of the Milky Way, such as thermal electrons or interstellar dust. Physical processes like dispersion and dust absorption are crucial for the creation of these auxiliary maps.

To reconstruct the GMF on the basis of existing measurement data, there are basically two approaches. On the one hand, the phenomenological approach of *parametric magnetic field models* can be used. This involves defining the structure of the magnetic field using analytical formulas with a limited number of parameters. These models include the general morphology of the magnetic field, such as galaxy arms and field reversals, but also local characteristics like nebulae in the solar system's neighbourhood. If a set of measurement data is given, one tries to find those model parameter values that are in concordance with the observables as closely as possible. For this purpose, within the course of this doctoral thesis IMAGINE, the *Interstellar MAGnetic field INference Engine* was developed. Due to parametric model's relatively small number of parameters, a fit is also possible with robust all-sky maps, even if they do not contain any depth information. However, there is the problem of arbitrariness in the approach of parametric models: there is a large number of models of different complexity available, which on top of that often contradict each other. In the past, the reconstructed parameter's uncertainty was often underestimated.

In contrast, a rigorous Bayesian analysis, as for example developed in this doctoral thesis with IMAGINE, provides a reliable analysis.
On the other hand, in addition to parametric models the GMF can also be reconstructed following a non-parametric approach. In this case, each space voxel has two independent degrees of freedom for the magnetic field. Hence, this type of reconstruction places much higher demands on the amount and quality of data, the algorithms, and the computing capacity. Due to the high number of degrees of freedom, measurement data are required which contain direct (parallax measurements) or indirect (by means of the Russel diagram) depth information. In addition, strong priors are necessary for those areas of space that are only weakly covered by the data. Simple Bayesian methods are no longer sufficient for this. Rather, information field theory (IFT) is now needed to combine the various sources of information correctly and to obtain reliable uncertainties. The PYTHON framework NIFTY (Numerical Information Field Theory) is predestined for this task. In its first release version, however, NIFTY was not yet natively capable of reconstructing a magnetic field and dealing with the order of magnitude of the problem's data. To be able to process given data, D2O was developed as an independent tool for data parallelization. With D2O parallel code can be developed without any hindrance of the actual development work. Basically all numeric disciplines with large datasets that cannot be broken down into subsets can benefit from this, which is the reason why D2O has been released as an independent package.
In addition, NIFTY has been comprehensively revised in its functional scope and structure, so that now, among other things, high-resolution magnetic field reconstructions can be carried out. With NIFTY it is now also possible to create maps of thermal electron density and interstellar dust on the basis of new and at the same time very large datasets. This paved the way for a non-parametric reconstruction of the GMF.

Chapter 1

Introduction

1.1 The Milky Way

Our solar system is part of the Milky Way, a spiral galaxy, illustrated in Fig. 1.1. The shape of spiral galaxies is dominated by a disk, which typically possesses several spiral arms and contains most of the galaxy's stars, gas and dust. The spirals are caused by gravity (Lin and Shu, 1964) and the resulting density waves (Francis and Anderson, 2009). The matter density within the disk follows a double exponential fall off profile

$$\rho(r, z) \approx \rho_0 \exp\left(\frac{-r}{r_0}\right) \exp\left(\frac{-|z|}{z_0}\right) \tag{1.1}$$

with r and z being Galacto-centric cylindric coordinates, and ρ_0 the density in the Galactic center. Within the disk, one can distinguish a stellar disk and a disk made up of gas and dust. For both the characteristic radius is $r_0 \approx 3.5\,\mathrm{kpc}$, while the characteristic height is $z_0^{\mathrm{stellar}} \approx 330\,\mathrm{pc}$ and $z_0^{\mathrm{dust}} \approx 160\,\mathrm{pc}$, respectively (Maoz, 2007). Further constituents of the Milky way are the *Galactic center*, the *bulge*, a *gas and star halo*, and a *dark halo*. At the Galactic center there is a compact stationary object with almost no luminosity, a mass of $4 \times 10^6\,\mathrm{M}_\odot$, and a size around 1 astronomical unit (AU). We assume that this object, called *Sagittarius A**, is a supermassive black hole. Around the Galactic center there is a spheroidal stellar bulge with a radius of $r^{\mathrm{bulge}} \approx 1\,\mathrm{kpc}$. The matter density within the bulge scales with the inverse cubic power of the distance to the center. The gas and star halo follows a cubic fall off profile, too, and 90% of the halo's stars lie within 30 kpc around the Galactic center (Harris, 1996). Using the virial theorem Zwicky (1937) concluded that the visible mass in the Galaxy is not sufficient to explain the stellar velocities. Today we are certain that only a small fraction of the Galaxy's total mass is visible (Battaglia et al., 2005, Battaglia et al., 2006, Kafle et al., 2014). A popular model for this dark matter is a dark halo following a Navarro-Frenk-White (NFW) profile:

$$\rho(r) = \frac{\rho_0}{\frac{r}{r_0}\left(1 + \frac{r}{r_0}\right)^2} \tag{1.2}$$

Fitting a NFW profile to stellar velocities leads to an estimate for the dark halo mass of 6×10^{11} to $3 \times 10^{12}\,\mathrm{M}_\odot$. Since the luminous matter amounts to $9 \times 10^{10}\,\mathrm{M}_\odot$, 95% of the Galaxy's mass

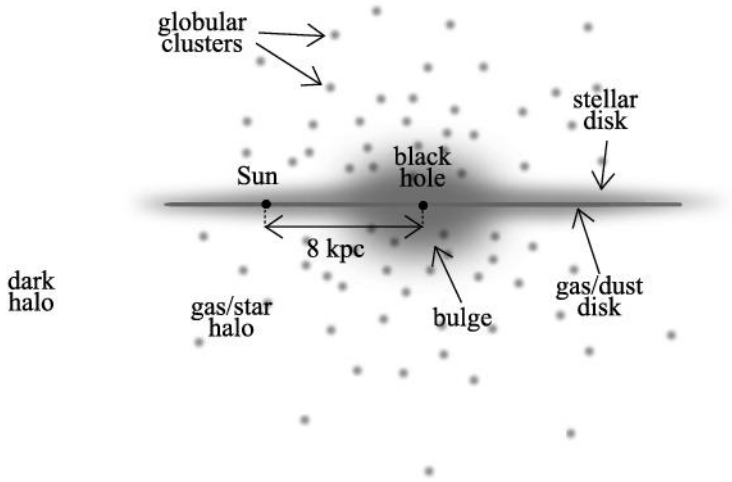


Figure 1.1: An illustration of the Milky Way taken from Maoz (2007).

are not visible (Battaglia et al., 2005, Battaglia et al., 2006, Kafle et al., 2014).
The solar system is located $z_\odot = 30\,\mathrm{pc}$ above the mid plane and roughly $r_\odot = (8.0 \pm 0.5)\,\mathrm{kpc}$ away from the Galactic center Maoz (2007). As the characteristic radius for the disk is $r_0 \approx 3.5\,\mathrm{kpc}$ the sun belongs to the outer regions of the Galaxy.

1.1.1 The Galactic Magnetic Field

The Galactic magnetic field (GMF) has a significant influence on the structure of the Milky way, as its energy density is on a par with that of the turbulent gas or cosmic rays (CR). To be specifc, the interstellar medium (ISM) continuously gets sheared due to the differential rotation in the Galatic disk. Furthermore, supernovae cause turbulent gas circulation. Thus, the energy density of the magnetic field is constantly replenished by the accompanying dynamo processes. At the same time, magnetic energy is lost through Galactic outflows and winds, but also by relaxation and reconnection processes of the magnetic field. The equilibrium between these opposing processes often leads to characteristic magnetic field configurations: The magnetic field orientation may follow the spiral structure of the galaxy, but field reversals and complex three-dimensional structures are also possible. Charged particles can move freely along magnetic field lines. However, the Lorentz force suppresses movements that traverse orthogonal to a magnetic field. Through this mechanism, the galactic magnetic field influences the structure of the ISM of the Milky Way. Likewise, the opposite influence is as important: from transport phenomena such as the outflow of hot gas or relativistic particle populations from the Galactic disk into the circum- and extra-Galactic space, we can deduce where Galactic magnetic field lines open up into the extra-galactic space. The thermal conductivity of the ISM along and across the magnetic field lines differs dramatically. Therefore, phases of the ISM with different temperatures are partially isolated from each other if not connected by field lines. Due to their influence on the gas movements, the magnetic fields are also involved in the angular momentum transport and the

radial gas flow within the Galactic disk. They therefore play an important role in the question of where new stars form and how gas masses are transported to the central black hole of the Galaxy. In addition to its great influence on the structure and dynamics of the ISM, the GMF is also important for a number of problems in astro- and astroparticle physics. The magnetic field deflects ultra-high-energy cosmic rays (UHECR) that are measured on the basis of the particle showers they cause in the Earth's atmosphere (Dawson et al., 2017, Farrar et al., 2013, Giacinti et al., 2013). This makes it difficult to identify astronomical sources that are still unknown. Relativistic electrons traveling through the GMF generate polarized radio synchrotron radiation. In addition, dust grains align with the magnetic field which causes thermal dust emission to be polarized. Together, this radiation contaminates the sensitive measurements of polarization in the cosmic microwave background (CMB), which is a tracer for early universe phenomena like primordial sound and gravitational waves. The study of extragalactic sources is disturbed by GMF-induced Faraday rotation as this effect alters the polarization direction of radiation from extragalactic photon sources (Bell and Enßlin, 2012, Bell et al., 2011, Oppermann et al., 2011, 2012, 2015). It is desirable to be able to distinguish the contribution of the Galactic Faraday rotation well from the extra-Galactic one, since the latter is to be used for the study of magnetic fields in intergalactic space. Thus, it is very desirable to have a description of the GMF that is as accurate as possible. Fig. 1.2 illustrates the entangled denpendencies between the constituents of the Galaxy and which component of the GMF influences them, respectively. Since the number of influential factors is considerable, below we first give an overview of the involved Galactic constituents, then discuss the relevant physical processes, and finally discuss the chain of inference.

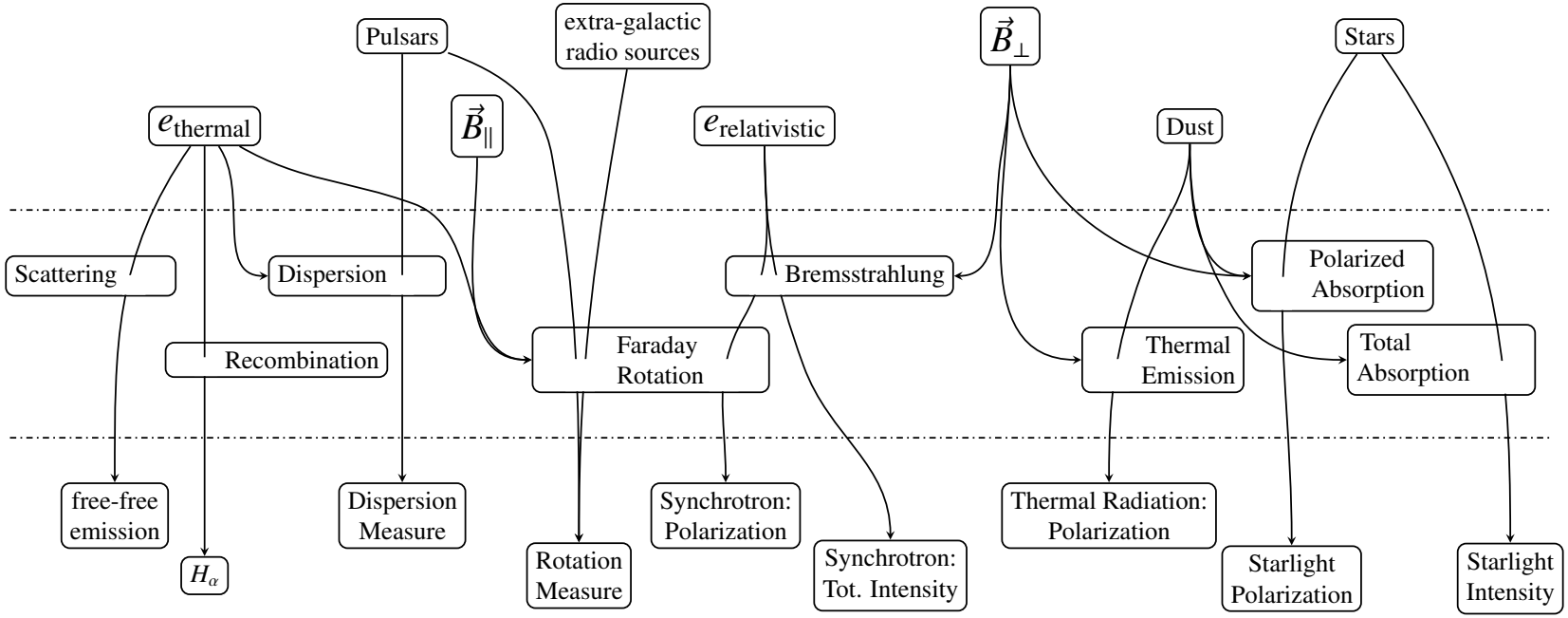


Figure 1.2: Illustration of the logical dependencies between constituents of the Galaxy (e.g. Pulsars), the involved physical effects (e.g. Faraday rotation), and the measurable observables (e.g. dispersion measure), respectively.

1.1.2 Constituents of Interest

As discussed above, the GMF plays a role in various astrophyiscal processes. Hence, measuring those gives us insight in the structure of the GMF. The involved constituents of the Milky way are briefly discussed in the following.

1.1.2.1 Thermal Electrons

In contrast to relativistic cosmic rays, the free electrons in the thermally ionized interstellar plasma are called *thermal electrons*. Their distribution is influenced by turbulent motions and shock fronts, e.g. from supernovae. On a large scale the morphology of the interstellar plasma should follow the disk and spiral arms of the Milky Way. However, as discussed in Sec. 1.1.1, the thermal electrons are heavily influenced by the GMF and vice-versa. In the past, huge effort has been made to construct at least parametric models of the thermal electron density. The first rather complex 3D approach was made by Taylor and Cordes (1993). In the course of this, radio observations of ionized hydrogen were used to get location information about the spiral arms. Pulsar dispersion measure data, c.f. Sec. 1.2.2, was used for constraining the model's parameters. As of today, the most widely used model is the NE2001 model by Cordes and Lazio (2002) which is based on Taylor and Cordes (1993). Its improvements are based on a larger DM data set that became available in the meantime, on the fact that H_α measurement data has now been used, and that information about the local solar neighborhood was taken into account. At the same time NE2001 has a major weak point: it is highly heuristic and contains inconsistencies like cut-out regions that just wouldn't fit the data. As a consequence, NE2001 never got published in a peer-reviewed journal; and yet its preprint version is often quoted. But in turn, NE2001 was also improved: YMW16 (Yao et al., 2017) serves as one example. Being highly non-trivial, the ways to infer the thermal electron density is discussed in Sec. 1.3.1.

1.1.2.2 Cosmic Rays

The number distribution of cosmic rays is dominated by relativistic protons, followed by 10 % alpha-particles and less than that massive nuclei (Schlickeiser, 2002). The abundance roughly matches the element abundance in the solar system, except for lithium, beryllium and boron, because of spallation processes when crossing interstellar matter. Their energy density (1 eV/cm^3) is comparable to the one of the GMF ($B^2/(2\mu_0) \approx 0.3$ eV/cm^3). CRs primarily get accelerated by first-order Fermi processes in strong shocks caused by supernovae and then traverse the Galaxy. In addition to diffusion, CRs are affected by convection with the plasma and also interaction with background photons must be taken into account (Matarrese et al., 1986). Furthermore, while propagating through the ISM CRs resonantly excite Alfvén waves. Those Alfvén waves in turn scatter CRs which partly isotropizes the velocity distribution in the wave's rest frame. As a result, low-energy cosmic rays stream down the wave's gradient (Zweibel, 2013).
While propagating through the Galaxy, CRs produce secondary particles like electrons, positrons, (anti-)protons, and also heavier nuclei via hadronic interaction. While also pions are produced, gamma-rays are emitted during their decay. Comparing observations with modeled primary and calculated secondary fluxes, one can particularily infer the density of cosmic ray electrons.

As discussed in Sec. 1.2.3, this is crucial information when extracting information regarding the GMF from synchrotron data. At the same time, the propagation of CRs is influenced by the GMF as they gyrate around the field lines of the GMF and therefore are confined by it. This illustrates that for a sophisticated deduction of the GMF also CR dynamics must jointly be considered. Both fields should be inferred together.

1.1.2.3 Pulsars

While burning, stars produce ever heavier elements. However, nuclear fusion is exothermic only until iron is reached. Because of this, heavy stars ($> 12\,M_\odot$) steadily build up an iron core that collapses once it reaches the Chandrasekhar limit ($1.4\,M_\odot$). Beyond this limit, the electron degeneracy pressure in the core is insufficient to counteract the gravitational self-attraction. When collapsing, protons and electrons merge into neutrons via inverse beta decay (Lieb and Yau, 1987). During this supernova explosion the outer shells are blown away into a ionized gas nebula and a highly compact remnant remains: a neutron star. According to established models Neutron stars have masses from $1.44\,M_\odot$ up to $3\,M_\odot$. Below that, there is no collapse and they become a white dwarf. Above that, the models predict that the supernova becomes a black hole. Despite their mass, neutron stars have a radius of about 10 km, which is only twice of their Scharzschild radius. At the same time, the angular momentum is preserved during the collapse and as a consequence, the neutron star is rapidly spinning. The neutron star with the highest measured frequency so far is *PSR J1748-2446ad* spinning at 716 Hz (Hessels et al., 2006). In addition to the angular momentum, also the magnetic moment is preserved during the collapse, resulting in typical magnetic field strengths of 10^8 T. If the axis of the magnetic dipole is not aligned with the rotation axis, the ionized plasma gets excited by the ever-changing magnetic field and emits radiation. Since charged particles in a strong magnetic field can move freely only along the field lines, the radiation is aligned with the magnetic dipole axis (Gold, 1968, Pacini, 1968). As this axis is subject to the pulsar's rotation a stationary observer sees the radiation, if at all, as periodic pulses.

1.1.2.4 Dust

Stars and the ISM are in constant interaction: new stars form from interstellar clouds, and at the end of their lives, many stars eject a substantial fraction of their mass back into the ISM. During their active time those stars produced heavier elements from hydrogen and helium by nuclear fusion. As a consequence, the abundance of heavier elements in the ISM increases over time. A major fraction of these heavier atoms compose solid particles of less than 10^{-6} m in size (Whittet, 2002). Those dust grains make up roughly 1 % of the ISM's mass but are the main reason for its opacity. They are so effective at scattering, absorbing and re-emitting starlight that, despite their small mass fraction, they have a decisive influence on our view of Galaxy. By way of illustration, at visible wavelengths only one photon in every 10^{12} that has been emitted in the center of our galaxy is received by us. The energy absorbed by the dust is re-emitted in the form of infrared radiation, which accounts for 20 % of the total bolometric luminosity of the Milky Way (Whittet, 2002). Note that the dust grains cannot reasonably be assumed to be spherical.

Physically, non-spherical dust grains are much more reasonable as they are also made of, among other things, anisotropic crystalline material. For example, graphite particles have a minimum free energy when flattened (Draine, 1989).

1.2 Physical Effects

1.2.1 Emission Processes

The warm ionized ISM emits radiation due to both, recombination and scattering between electrons and ions. Since the most common element in the ISM is hydrogen, it is worthwhile to study its spectral lines. Two wildly used ones are the following: First, the H_α line with a wavelength of $\lambda = 656\,\mathrm{nm}$, which corresponds to the transition from the $n = 3$ to the $n = 2$ level. This spectral line can clearly be identified in the corresponding observational data, although being subject to dust. Second, the H_I hyperfine transition line of hydrogen's ground state with a wavelength of 21 cm. The latter has the advantage that the scattering cross section with the ISM is rather low. By means of a Doppler analysis, H_I yields information about the speed of the corresponding hydrogen clouds. Combining this with models of the Galaxy's velocity profile one can also infer the cloud's location in space (Kerr, 1969).
In addition to recombination effects, there is also an important scattering effect. The thermal electrons in the ISM scatter off each other and off ions. Because of this acceleration, the electrons emit bremstrahlung; the so-called free-free emission. In contrast to the spectral lines discussed above, the spectrum of free-free emission is continuous. The emitted power per unit frequency bin is inversely proportional to the frequency squared. Similarly to H_α, at free-free emission pairs of electrons and ions, respectively, need to interact. Hence, both types of emission processes are proportional to the free electron density squared. Thereby, the quantity of interest is its line-of-sight integral, called emission measure (EM)

$$\mathrm{EM} = \int_0^d \mathrm{d}l\, n_e^2, \tag{1.3}$$

as discussed in, e.g., Reynolds et al. (1974) and Finkbeiner (2003).

1.2.2 Dispersion

When electromagnetic radiation passes through free electrons, e.g., those of the interstellar plasma, it induces oscillations of them. This leads to a dielectric constant, and therefore to a refractive index. The effective speed of light $c_{\mathrm{phase}}(\nu)$ in this medium is frequency dependend and is given by

$$c_{\mathrm{phase}}(\nu) = c\sqrt{1 - \frac{\nu_p^2}{\nu^2}}, \tag{1.4}$$

where ν_p is the plasma frequency, and c is the speed of light in vacuum. The plasma frequency reads

$$\nu_p = \sqrt{\frac{n_e e^2}{\pi m_e}}, \tag{1.5}$$

where n_e is the electron density, e is the elementary charge, and m_e is the electron mass. Hence, the speed of light increases with the radiation's frequency. In the case of pulsars this means, that high frequencies in the spectrum of a pulse reach the observer before the low ones. Given Eq. (1.5) and the electron densities in the warm ionized gas in the ISM, the plasma frequency stays below 100 kHz. We analyze the radiation of pulsars in the MHz and GHz regime which means that Eq. (1.4) can be linearized in ν_p^2 and thus also in n_e. Using this linear approximation, given a source at a distance d, the time for the radiation to arrive at the observers place is

$$t = \frac{d}{c} + \frac{k_{\mathrm{DM}}}{\nu^2} \int_0^d \mathrm{d}l \; n_e \tag{1.6}$$

with

$$k_{\mathrm{DM}} = \frac{e^2}{2\pi m_e c}. \tag{1.7}$$

The line-integral over n_e is called dispersion measure (Rybicki and Lightman, 2008):

$$\mathrm{DM} = \int_0^d \mathrm{d}l \; n_e \tag{1.8}$$

Eq. (1.6), or its derivative with respect to ν, respectively, can be fitted to the observed pulses' frequency profile. By this, if the distance d is known, one can infer DM for the specific positions of the pulsars in space.
Fig. 1.3 shows the results of a sky map simulation conducted with Hammurabi X using the YMW16 (Yao et al., 2017) Galactic thermal electron density model, as used in Chap. 2.

1.2.3 Synchrotron Emission

When charged particles move orthogonally to a magnetic field, they are deflected by the Lorentz force. As a consequence they gyrate around the field lines and emit synchrotron radiation. Due to relativistic beaming effects the radiation is not emitted isotropically, but rather within a narrow forward-oriented cone. Embedded in the GMF, relativistic electrons emit synchrotron radiation in the MHz and GHz range. For a relativistic velocity v, the gyration frequency reads

$$\nu_{\mathrm{g}} = \frac{eB}{2\pi\gamma m_e c}, \tag{1.9}$$

with the Lorentz factor

$$\frac{1}{\gamma} = \sqrt{1 - \frac{v^2}{c^2}}. \tag{1.10}$$

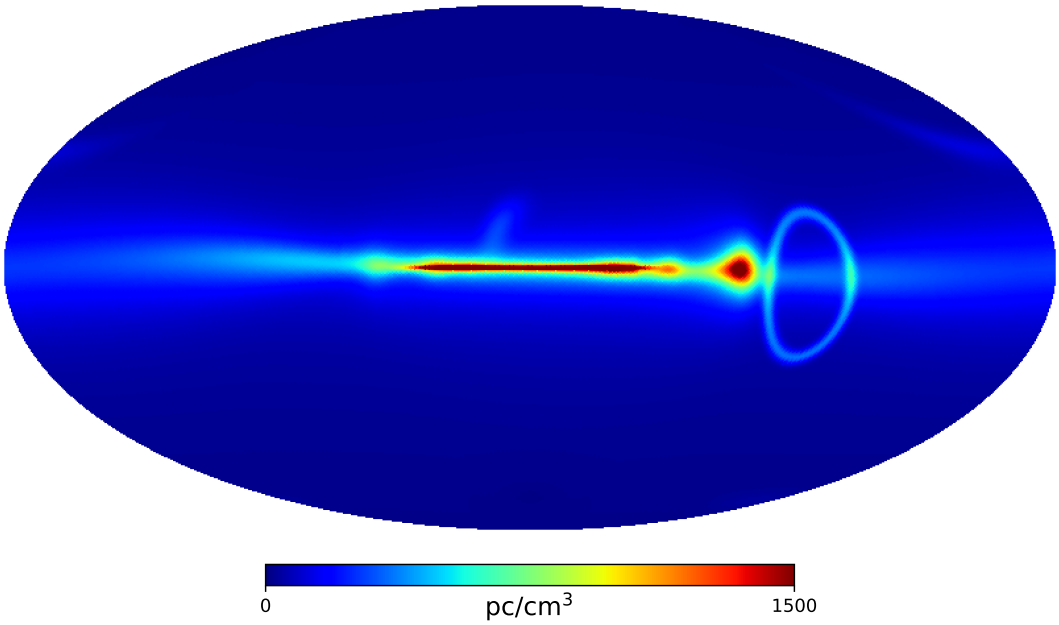


Figure 1.3: Map of the Galactic dispersion measure produced with HAMMURABI X based on the YMW16 (Yao et al., 2017) thermal electron model. In addition to the Galactic disk, also local features of the model can be seen; all in front the Gum Nebula that appears as a noticeable ring.

The intensity of the synchrotron radiation is determined by the intensity of the acceleration the electron experiences. $a_\perp$ may denote the part of the acceleration that is orthogonal to the electron's velocity vector; it can be written as

$$a_\perp = 2\pi\nu_\mathrm{g} v_\perp \tag{1.11}$$

whereby $v_\perp$ is the component of the electron's velocity that is perpendicular to the magnetic field. As shown by, e.g. Rybicki and Lightman (2008), the power of the emitted synchrotron radiation is given by

$$P = \frac{2e^2}{3c^3}\gamma^4 a_\perp^2 = \frac{2}{3}\frac{\gamma^2 e^4}{m_e^2 c^5} B^2 v^2. \tag{1.12}$$

When assuming that the electrons' velocity distribution is isotropic, averaging over all possible orientations yields

$$P = \left(\frac{2}{3}\right)^2 \frac{\gamma^2 e^4}{m_e^2 c^5} B^2 v^2. \tag{1.13}$$

Synchrotron radiation is to a high degree linearly polarized, to be precise, orthogonally with respect to the magnetic field. This makes it a valuable source of information for $B_\perp$; that part of the magnetic field which is orthogonal to the lines-of-sight.
Fig. 1.4 shows the results of a synchrotron emission simulation conducted with Hammurabi X, as used in Chap. 2. Except for an overall factor, the two maps of total intensity at 1 GHz and 30 GHz are structurally identical; note the different color scales. In contrast, the maps showing the polarized intensity differ heavily. This is due to Faraday depolarization, which will be discussed in the upcoming section: Sec. 1.2.4. As illustrated in Fig. 1.2, polarized intensity maps of synchrotron emission are therefore examples for observables that are shaped by two physical processes.

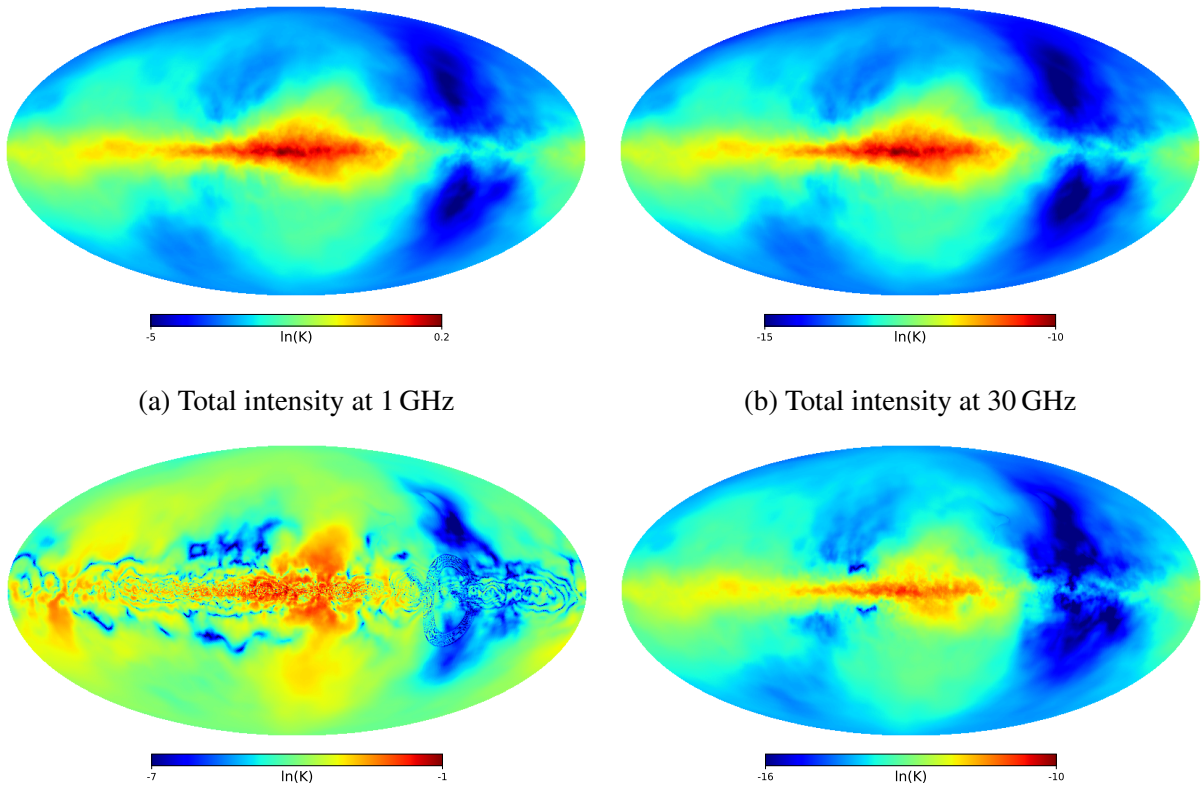
1.2.4 Faraday Rotation & Depolarization

In the presence of free electrons and a magnetic field parallel to the direction of propagation, the polarization of an electromagnetic wave is rotated; this effect is known as Faraday rotation. Note that linear polarization can be regarded as the superposition of according left- and right-handed circular polarization. The mechanism behind Faraday rotation is that the wave's electric field induces gyration movements of the electrons. With the magnetic field being present, the electric conductivity, thus the dieletric constant, and therefore the individual speed of light of the two circular polarization modes depends on their rotational sense. The left- and right-handed component of a linearly polarized electric field vector are rotated by the angle (Schlickeiser, 2002)

$$\phi_{\mathrm{r/l}}(\nu) = \int_0^L \mathrm{d}l\, \frac{2\pi\nu}{c} \sqrt{1 - \frac{\nu_\mathrm{p}^2}{\nu(\nu \pm \nu_\mathrm{c})}} \tag{1.14}$$

after traversing a distance L. This depends on the plasma frequency

$$\nu_\mathrm{p} = \sqrt{\frac{n_e e^2}{\pi m_e}}, \tag{1.15}$$



(a) Total intensity at 1 GHz

(b) Total intensity at 30 GHz

(c) Polarized intensity at 1 GHz

(d) Polarized intensity at 30 GHz

Figure 1.4: Comparison of synchrotron emission maps of total and polarized intensity at 1 GHz and 30 GHz based on the YMW16 (Yao et al., 2017) model and the WMAP LSA (Page et al., 2007) model for the thermal electron density and the Galactic magnetic field, respectively. The WMAP LSA model's parameters were set to $B_0 = 1.2\,\mu$G, $\psi_0 = 27.0°$, $\psi_1 = 0.9°$, and $\chi_0 = 25.0°$, following Page et al. (2007). To the regular WMAP LSA model, an isotropic random magnetic field was added with a characteristic strength of $0.8\,\mu$G, cf. Sec. 1.4 and Sec. 2.4.3 for details.

and the cyclotron frequency

$$\nu_{\mathrm{c}} = \frac{eB}{2\pi m_e c}. \tag{1.16}$$

Hence, the polarization angle of the linear polarized wave got rotated by

$$\Delta\phi = (\phi_r - \phi_l)/2. \tag{1.17}$$

In the limit of $\nu \gg \nu_{\mathrm{c}}$ and $\nu \gg \nu_{\mathrm{p}}$ the square root in Eq. (1.14) can be linearized which yields for $\Delta\phi$

$$\Delta\phi = \frac{e^3\lambda^2}{2\pi m_e^2 c^4} \int_0^L n_e \vec{B} \cdot \mathrm{d}\vec{l}. \tag{1.18}$$

Finally, taking out the wavelength-dependence, one gets to the definition of Faraday depth:

$$\Phi = \frac{e^3}{2\pi m_e^2 c^4} \int_0^L n_e \vec{B} \cdot \mathrm{d}\vec{l} \tag{1.19}$$

Fig. 1.5 illustrates the effect of Faraday rotation for the YMW16 (Yao et al., 2017) and WMAP LSA (Page et al., 2007) models. The dependency on the thermal electron density in Eq. (1.19) can be clearly seen, as the Gum Nebula is as traceable in Fig. 1.5 as in Fig. 1.3.
There are several effects how Faraday rotation reduces the measured degree of polarization. We discuss the two most important. First, as discussed above, Faraday rotation depends on the wavelength. If a source emits polarized but not perfectly monochromatic radiation, the measured degree of polarization gets reduced as every detector has a finite frequency window. This effect is important for frequencies $\nu \lesssim 1\,\mathrm{GHz}$. Second, if there are many sources (or one elongated source) along the line-of-sight, the distance from the observer to each source and therefore the Faraday depth is different, respectively. Hence, even if all sources initially emitted radiation with the same polarization angle, the measured Faraday rotation will be smeared out. The sum of all those effects is called Faraday depolarization. Fig. 1.4 illustrates the effect of Faraday depolarization and its dependence on the wavelength, cf. Eq. (1.18).

1.2.5 Dust Absorption and Emission

Dust absorption and emission can provide valuable information on the Galactic magnetic field. However, they have not been used in this work, hence the underlying concepts should be discussed only briefly. In this section we mainly follow Whittet (2002).
Extinction occurs whenever electromagnetic radiation propagates through a medium that contains small particles. In principle there are two extinction processes: absorption and scattering. When dust absorbs light, it incorporates its energy, which is converted into heat. This thermal energy causes the dust to emit thermal radiation. If the dust has an anisotropic geometry and is uniformly aligned, partially absorbed as well as thermally emitted light has a non-vanishing polarization.
As a start, we analyze the case of spherical dust. Although this is a great simplification, also aspherical dust can effectively be approximated by spherical dust if it is not aligned or polarization

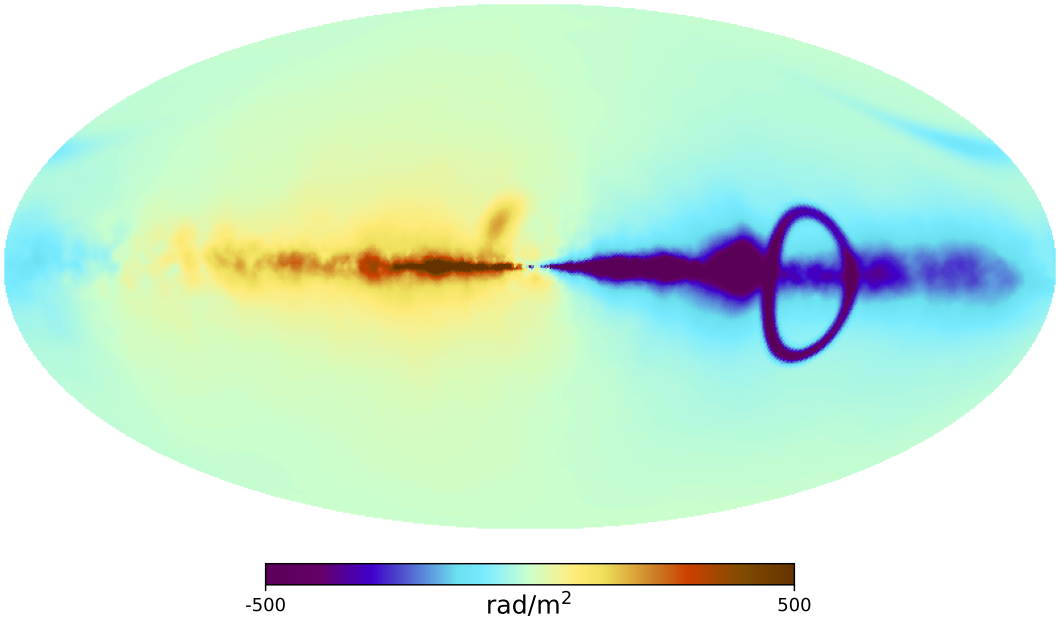


Figure 1.5: Map of the Galactic rotation measure produced with HAMMURABI X based on the YMW16 (Yao et al., 2017) model and the WMAP LSA (Page et al., 2007) model for the thermal electron density and the Galactic magnetic field, respectively. The WMAP LSA model's parameters were set to $B_0 = 1.2\ \mu$G, $\psi_0 = 27.0°$, $\psi_1 = 0.9°$, and $\chi_0 = 25.0°$, following Page et al. (2007). To the regular WMAP LSA model, an isotropic random magnetic field was added with a characteristic strength of 0.8 μG, cf. Sec. 1.4 and Sec. 2.4.3 for details.

is not considered. Assuming dust with a density of n and an extinction cross section C_{ext}, the attenuation of the starlight intensity per line of sight increment dL reads

$$\frac{\mathrm{d}I}{I} = -nC_{\text{ext}}\mathrm{d}L. \tag{1.20}$$

Integrating Eq. (1.20) over the full line-of-sight of a star with original brightness I_0, one gets

$$I = I_0 e^{-\tau}, \tag{1.21}$$

whereby τ is the optical depth of extinction

$$\tau = \int \mathrm{d}L\, nC_{\text{ext}}. \tag{1.22}$$

Usually the extinction is given in magnitudes A:

$$A = -2.5 \log\left(\frac{I}{I_0}\right) \tag{1.23}$$

The absorption of starlight by dust can lead to its partial linear polarization. Likewise, thermal dust emission can be polarized linearly, too. There are mainly two reasons for this. First, the dust grains have an isotropic geometry, but are in themselves – like graphite – optically anisotropic. For a net polarization, the optical axes would have to align uniformly, which is very unlikely to happen because of their isotropic geometry. The second possible reason that can lead to polarizing dust absorption is that the grains have an anisotropic geometry. It is the subject of active research into why anisotropic dust aligns itself in a magnetic field. A simple model for this is explained in the following.
Interstellar dust rotates due to random collisions. While doing so, elongated and disk-shaped dust particles rotate around the prinicpal axis of inertia, cf. Fig. 1.6. Because of the Barnett effect, the atomic magnetic dipoles within the dust grains align themselves uniformly, causing a net magnetic moment. Hence, the rotation axes of the dust particles align themselves with the magnetic field. This alignment of the dust ensures anisotropy of the extinction cross section and thus partial polarization of translucent light. Measuring the intensity for the different polarization angles, one gets a minimum and a maximum intensity of I: I_{min} and I_{max}, respectively. Based on that, the degree of polarization P is usually expressed as a percentage

$$P = 100\%\, \frac{I_{\text{max}} - I_{\text{min}}}{I_{\text{max}} + I_{\text{min}}}. \tag{1.24}$$

Based on a certain model for a dust geometry distribution, it can be calculated how strong the inherent rotation of the dust, the induced magnetic moment and thus the alignment to the GMF is. In addition, the extinction cross section can be calculated. Thus, the strength of the magnetic field orthogonal to the line of sight can be determined from the measured polarized dust extinction.
Touching the topic of dust only briefly, we only note that the extinction cross section depends on the light's wavelength: the shorter its wavelength, the more it gets absorbed, leading to the so-called reddening.

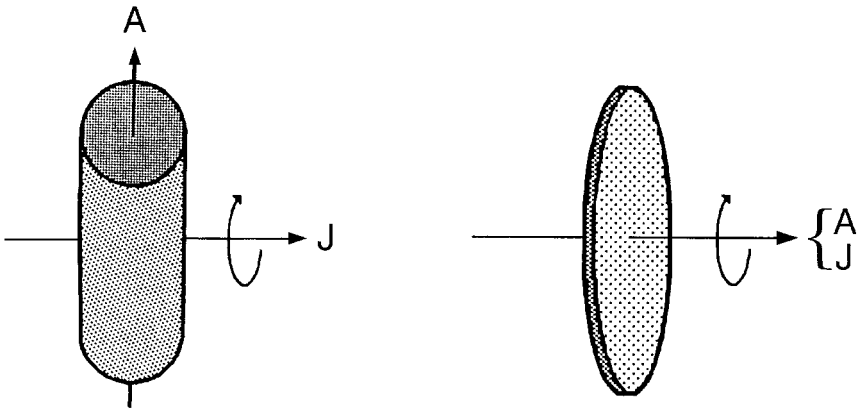


Figure 1.6: Illustration of the orientation of angular momentum J with respect to the grain's symmetry axis A; taken from Whittet (2002).

1.3 Chain of Inference

As shown in Fig. 1.2 the Galactic thermal electron density plays a crucial role for reconstructing the GMF, especially because it significantly affects Faraday rotation.

1.3.1 Inferring the Thermal Electron Density

For the reconstruction of the thermal electron density there are several sources of information available: the most prominent are dispersion pulsed radiation, free-free emission and hydrogen emission lines. However, the inference is mainly based on dispersion, but not the latter two, as we discuss in the following: First, free-free emission is a problematic source of information because it does not dominate the sky's electromagnetic energy spectrum at any frequency. Hence, it must be inferred by combining data at different frequencies using spectral information (Bennett et al., 2003). For this in turn, the temperature of the thermal electrons T_e must be known. Practically, one uses the mean value of the temperature of the ionized warm medium (8000 K). However, this is only a rough estimate as T_e is expected to vary in space and especially along a line-of-sight.
Second, in contrast to free-free emission, Hydrogen emission lines are easier to extract from the data due to the distinct shape of the spectral lines. However, radiation with a wavelength of 656 nm is subject to dust absorption. Hence, one needs a reliable dust map to account for the extincted radiation. Thus, in principle free-free and H_α can be used as long as the interrelated information is available. Nonetheless, conceptually a huge disadvantage remains: measurement data of free-free emission as well as emission lines is available in the form of all-sky maps, which as such do not contain any depth information. In contrast, with pulsars one has probes distributed in space which is a prerequisite for a sophisticated 3D reconstruction. This is what makes dispersion measure data from pulsars so valuable, though the measurement of their distance is challenging. For this it is advisable to use parallaxes, even if this involves uncertainties of several ten percent. Distressingly enough, in practice for the inference of the thermal electron density, often (e.g. Cordes and Lazio (2002)) distance information is used that itself is based on

a dispersion measure analysis that in turn includes an assumed certain thermal electron density model. This is highly critical circular reasoning yielding unreliable results.
Using only pulsar data with independent distance estimates, Schnitzeler (2012) finds that with the data available at that time there is no support for excessively complex thermal electron models. Hence, up to then, no detailed features could be resolved. In 2016, Greiner et al. (2016) demonstrated how to make a non-parameteric reconstruction based on a Pulsar driven dispersion measure analysis. If the 3D position of pulsars is roughly known, a true 3D reconstruction becomes possible, whereby matter density power spectrum and Galactic profile are learned on the fly. It is very likely that the SKA will yield data with which a highly informative reconstruction of the Galactic thermal electron density will become possible.

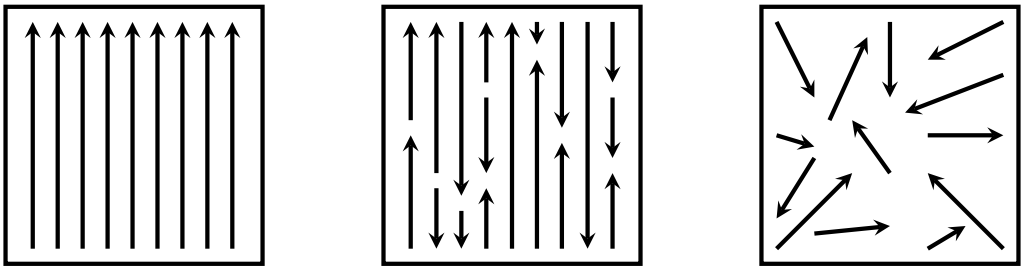
1.3.2 Inferring the Magnetic Field

Once one decided for a map of the Galactic thermal electron density field, the effect of Faraday rotation on the polarization of pulsar emission and extra-galactic radio sources can be analyzed. Note that in contrast to sources within the Galaxy, extra-Galactic sources have the downside of not providing any depth information. With respect to the Galactic magnetic field, as discussed in Sec. 1.2.4, Faraday rotation provides insight in $B_{\parallel}$, the component of the GMF which is parallel to the observer's lines-of-sight. Mainly a disturbance contribution, Faraday rotation also has an influence on the polarization angle of the synchrotron radiation that is emitted by relativistic electrons and cosmic rays. For $B_{\perp}$, the direction of the GMF that runs orthogonal to the observer's lines-of-sight, synchrotron radiation is the primary source of information, as discussed in Sec. 1.2.3. However, just as one needs the thermal electron map for Faraday rotation analyses, one needs a map of the relativistic electrons $e_{\text{relativistic}}$ when utilizing synchrotron radiation. In practice, those maps are based on either excessively simple models, or on forward simulations codes like CRPropa3 (Alves Batista et al., 2016), Picard (Kissmann, 2014), Galprop (Moskalenko, 2012), or Dragon2 (Evoli et al., 2017) Hereby, the problem is that those codes need a (at least large-scale) GMF as input. This means that in principle one has to perform a joint analysis of the GMF and $e_{\text{relativistic}}$, which is much harder than taking a map for $e_{\text{relativistic}}$ for granted[1]. Finally, it should be noted that thermal dust emission and starlight absorption can be used as sources of information to constrain $B_{\perp}$, too.

1.4 Models of the Galactic Magnetic Field

The GMF can be subdivided into three components (Jaffe et al., 2010) using the following systematics: a large-scale, an isotropic random, and an anisotropic field, cf. Fig. 1.7. The first one has correlation lengths on kpc scales and is likely caused by a Galactic dynamo. Thereby, a small initial seed field is amplified by the conversion of mechanical energy of turbulent gas flows (Brandenburg and Subramanian, 2005). The isotropic random or turbulent field is assumed to

[1] Due to the chaotic nature of cosmic ray trajectories, it is not possible to do backward evaluations of the particle propagator. Since only forward evaluations can be used during the inference, this means that the number of applicable analysis methods is heavily restricted.



Ordered | Anisotropic random / Straited | Isotropic random

Figure 1.7: Illustrations of the three components of the Galactic magnetic field: an ordered, an anisotropic random or straited, and an isotropic random field.

originate from supernova outflows and turbulent motions of the magnetized interstellar plasma. Its strength as well as orientation varies from place to place in space. The third component is referred to as anisotropic, ordered random, or straited random field. This special kind of magnetic field is created if a magnetized medium that possesses an isotropic field gets heavily compressed, e.g. by a supernova shock. There are a variety of parametric GMF models in the literature. While the simplest ones have an axisymmetric spiral shape, the complicated models combine the three aforementioned models. See Sec. 2.4.3 for more information.

1.5 Information Theory

In the field of statistics there are two different approaches to calculate probabilities and uncertainties. On the one hand there is the frequentistic approach, where the probability of an event is determined by repetition of a corresponding experiment (Hogg and Tanis, 2010). By doing so, the challenge is to formulate clearly which assumptions were made when constructing the entire event-space, which is considered possible. On the other hand, there is the Bayesian approach that is used in this work, where probabilities are consistently formulated as subjective degrees of belief. This makes it possible to formulate all assumptions, but also previous knowledge about an experiment or model in terms of probability densities. For a detailed discussion see Cox (1946) and Jaynes and Baierlein (2004).
The key element of Bayesian inference is Bayes' theorem, which is derived from the product rule for conditional probabilities and for two statements A and B reads as follows:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}. \tag{1.25}$$

The posterior probability $P(B|A)$ is thus the product of the likelihood $P(A|B)$ and Prior $P(B)$ divided by the evidence $P(A)$. Subsequential updating of the posterior probability is possible by concatenated application of Bayes theorem. Furthermore, Lemm (2003) showed that Bayesian logic can be combined with field theory. Later, Enßlin et al. (2009) named this approach Information Field Theory (IFT). Using IFT, it is possible to infer on the basis of measurement data entire

fields of quantities of interest, for example two-dimensional celestial maps or three-dimensional volume reconstructions of the electron density as well as the galactic magnetic field.
In addition to the fact that the use of Bayesian inference is generally recommended, concretely it offers significant advantages for this work. In both parametric and non-parametric magnetic field reconstruction, the treatment of statistical uncertainty is very important, which can be easily achieved with Bayesian inference. In the case of parametric reconstruction, this is reflected in the example of the galactic variance. Further, the concept of evidence makes it possible to compare two completely different models. In the case of non-parameteric magnetic field reconstructions, the correct treatment of uncertainties and error propagation of, for example, distances determined with paralax measurements is very important, as these uncertainties are very high, namely often throughout in the double-digit percentage range. Further information is given in Sec. 2.2 and Sec. 4.2.1.

1.6 Outline of this thesis

As discussed in Sec. 1.4, modelling the GMF can be divided into two classes: the parametric and the non-parametric approaches. At present, there is a vivid scientific community building parametric magnetic field models and fitting them to the available measurement data. In Chap. 2 we present Imagine, a framework for comprehensible GMF model inference. Thereby, handling uncertainties properly is especially important for GMF models that also include random GMF components. For that purpose, we present the concept of Galactic variance in Sec. 2.3. After describing the Imagine framework in Sec. 2.4 we show in Sec. 2.5 how the framework eases Bayesian parameter estimation and model comparison. This chapter has been submitted as an independent publication to the journal *Astronomy & Astrophysics*. The arxiv print can be found here: https://arxiv.org/abs/1801.04341.
The long-term perspective for information theoretically reliable magnetic field models are non-parametric models. Combined with a Bayesian analysis, preferably formulated in the language of information field theory, they are the only way to truly let the data speak and precisely control which extra information is added to the inference. The difficulty with non-parametric reconstructions, however, lies in the high computational effort that has to be made. Thus, as long as the reconstructions would have to be calculated on a single shared-memory computer, memory size and processor performance are limiting factors. The structure of the inference problem causes that otherwise common methods like subdividing the data into smaller chunks cannot be applied here. This made it necessary to develop a data parallelisation solution that takes into account the needs of the given class of inference. In Chap. 3 we present our solution to this problem: d2o, a Python module for cluster-distributed multi-dimensional numerical arrays. We show how one can use d2o to develop parallelized code without the need to have prior knowledge of technologies such as OpenMPI or OpenMP, and without being distracted by them. This chapter, as well as Chap. A have been published as an independent publication in the *Journal of Big Data* (Steininger et al., 2016).
To implement inference algorithms efficiently the inference framework NIFTy was developed. The need to make it possible to do non-parametric reconstructions of magnetic fields, was the

trigger for a complete rebuild of NIFTy. Among other things, NIFTy 3 now supports arbitrarily complex tensor fields and thus particularly the reconstruction of non-parametric three-dimensional magnetic fields. Using d2o as the basis of parallelization, cf. Sec. 4.4.6, now high-performance computing clusters can be used despite the complex field structures that are involved. But in general, cf. Sec. 4.4, there are a number of other improvements that make NIFTy 3 an even more valuable system. This chapter has been submitted as an independent publication to the journal *PLOS ONE*. The arxiv print can be found here: https://arxiv.org/abs/1708.01073.
Finally, we conclude this work and give a brief outlook on future challenges.

Chapter 2

Inferring Galactic magnetic field model parameters using IMAGINE - An Interstellar MAGnetic field INference Engine

This chapter is additionally used as a journal publication submitted to Astronomy & Astrophysics *(Steininger et al., 2018).*
I am the principal researcher of the research described in this chapter. My contributions include the primal idea for the design and the implementation of the presented software framework, the primal idea for the presented likelihood, the conception and conduction of the performance and consistency tests, and the conception and conduction of the productive runs of the pipeline. Together with Torsten A. Enßlin (TE), Tess Jaffe (TJ), Ellert van der Velden (EV), Jiaxin Wang (JW), Marijke Haverkorn (MH), Jörg R. Hörandel (JH), Jens Jasche (JJ) and Jörg P. Rachen (JR) I developed the primal idea of IMAGINE and developed and refined the work's objective. TE, TJ, EV, JW, MH, JH, JJ, and JR helped to focus the work such that it has a high added-value in real-world applications. EV and JW helped implementing and working out the conceptual structure of the software package. Maksim Greiner helped elaborating the debugging of the presented likelihood and, together with JJ, the sampling. TJ helped conceptioning the consistency tests and supported the productive runs. I wrote this chapter for the most part. TE, TJ, EV and JW helped drafting the manuscript by language editing. Additionally, EV contributed approximately 30% to Sec. 2.4.2, and 50% to Sec. 2.4.5. JW contributed approximately 50% to Sec. 2.4.3, and 70% to Sec. 2.4.4. TJ contributed approximately 70% to Sec. 2.6. Furthermore, TE also fulfilled the role of a principal investigator as he is my PhD supervisor. All authors read, commented, and approved the final manuscript.

Abstract

The Galactic magnetic field (GMF) has a huge impact on the evolution of the Milky Way. Yet currently there exists no standard model for it, as its structure is not fully understood. In the past many parametric GMF models of varying complexity have been developed that all have been fitted to an individual set of observational data complicating comparability. Our goal is to systematize parameter inference of GMF models. We want to enable a statistical comparison of different models in the future, allow for simple refitting with respect to newly available data sets and thereby increase the research area's transparency. We aim to make state-of-the-art Bayesian methods easily available and in particular to treat the statistics related to the random components of the GMF correctly. To achieve our goals, we built IMAGINE, the *Interstellar Magnetic Field Inference Engine*. It is a modular open source framework for doing inference on generic parametric models of the Galaxy. We combine highly optimized tools and technology such as the MULTINEST sampler and the information field theory framework NIFTY in order to leverage existing expertise. We demonstrate the steps needed for robust parameter inference and model comparison. Our results show how important the combination of complementary observables like synchrotron emission and Faraday depth is while building a model and fitting its parameters to data. IMAGINE is open-source software available under the GNU General Public License v3 (GPL-3) at: https://gitlab.mpcdf.mpg.de/ift/IMAGINE

2.1 Introduction

The interstellar magnetic field in galaxies plays a key role in processes at various scales from star formation up to overall galactic evolution. Its energy density is comparable to that of the turbulent gas or cosmic rays (CRs), and therefore the dynamical feedback on the interstellar medium (ISM) must not be ignored. Galactic magnetic fields affect in- and outflows of the ISM that already exist as well as the formation of new ones. They influence the propagation of CRs, which gyrate along the field lines. Though these effects are all important, it is challenging to infer the field, since it is only accessible via indirect detection methods. Additionally, since our Solar System is located within the Galactic plane, the tracers of the Galactic magnetic field (GMF) in our own Milky Way are highly degenerate as they are line-of-sight integrated quantities. This also means that the view of the opposite side of the Galaxy is obstructed by the intervening ISM. Because of all this, the GMF is currently mainly modeled via heuristic parametric models that have physically motivated features. The degrees of freedom in those models are morphological properties, field strengths (of possibly individual spatial components) of the magnetic field and the strength and characteristics of random contributions. Significant progress has been made here, which is the reason why a rather large number of GMF models is available today. At the same time the available data becomes better and better. Hence, there is need for a standardized platform that allows systematic parameter estimation and model comparison for a continuously expanding abundance of models and data.

2.2 Bayesian Parameter Inference and Model Comparison

The GMF can naturally be thought of as a vector field with an infinite number of degrees of freedom: under the constraint of zero divergence the magnetic field can have an individual strength and direction at every point in space. This view corresponds to the most generic model possible, where the model's parameters are the field's degrees of freedom. To infer the GMF one must simplify this most generic model, for example, by discretizing space. Doing so reduces the model parameters to a finite but still huge number, namely twice the number of voxels of the considered volume. However, now one can try to concretely infer the magnetic field voxel by voxel, a method known as non-parametric modelling. Generally speaking, constraining those non-parametric models is certainly hard, because the huge number of degrees of freedom often are counteracted by a limited amount of data. Because of this, one often builds a simpler model with a heavily reduced number of parameters, which therefore only covers a tiny slice in the full parameter space but still represents the most important features of the modeled quantity. In the case of the GMF, various models have been developed that differ greatly in their complexity: the number of parameters varies between only a few and up to 40. Given a model and observational data one must find an estimate for a set of the model's parameters that explains the observed data well. However, in addition to the parameter estimation of a given model, there is also the task of comparing the plausibility of different models. In the case of GMF inference this is especially important since so far there is no standard model available.
In terms of Bayesian inference, parameter estimation and model comparison can be described by the following components: a given model m that has a set of parameters θ shall be constrained by data d. This means, that we are interested in the posterior probability density $P(\theta|d, m)$. Bayes' theorem provides us with a calculation prescription

$$P(\theta|d, m) = \frac{P(d|\theta, m)P(\theta|m)}{P(d|m)}, \tag{2.1}$$

where $P(d|\theta, m)$ is the likelihood of the data, $P(\theta|m)$ is the parameter prior, and $P(d|m)$ is the model's evidence. The latter guarantees the posterior's normalization and is given by

$$\mathcal{Z} = P(d|m) = \int_{\Omega_\theta} P(d|\theta, m)P(\theta|m)\mathrm{d}\theta. \tag{2.2}$$

For parameter estimation with one model, the evidence can be neglected, hence it is sufficient to maximize the product of the likelihood and the prior. However, for comparing different models, e.g., m_1 and m_2, one needs normalized posteriors to form the ratio

$$R = \frac{P(m_1|d)}{P(m_2|d)} = \frac{P(d|m_1)P(m_1)}{P(d|m_2)P(m_2)} = \frac{\mathcal{Z}_1 P(m_1)}{\mathcal{Z}_2 P(m_2)}. \tag{2.3}$$

Often there is no strong a priori reason for preferring one model over the other which corresponds to setting the model prior ratio $P(m_1)/P(m_2)$ to unity. In this case, the model's evidence is the only source of information for model selection.

2.3 Galactic Variance

The likelihood $P(d|\theta, m)$ describes the probability to measure the data d if reality was given by θ and m. By modeling the physical system this probability can be explicitly calculated for certain sets (θ, m). For this, one uses a forward simulation code to compute observables like sky-maps of Faraday rotation, synchrotron emission, and thermal dust emission. Given measured data, by modeling the noise characteristics of the detector, a probability can be assigned to the calculated maps, which is in principle a standard approach. However, when analyzing parametric models of the GMF one must be careful at this step because of how those models describe small scale structure of the magnetic field. Generally speaking, parametric models specify the large scale structure of the magnetic field explicitly by parameterizing the geometry of its components – for example, the disk and possibly its arms, the halo, X-shaped components, et cetera – and the field strength therein. Together, these components form the so-called regular field. Small scale structure, in contrast, is modeled in terms of its statistical properties rather than an explicit realization. This means, that when for a given parameter set θ a model instance is created, a random magnetic field is generated and added to the regular field. Depending on the model, the random magnetic field obeys, for example, a certain power spectrum, is locally proportional to the regular field, or shows a certain degree of anisotropy. As a consequence, the set (θ, m) corresponds not only to one, but rather infinitely many possible field realizations. For the calculation of a likelihood this means that the measured observables must be compared with the ensemble average, which in practice is the simulated mean of a yet finite set of observable realizations that result from the magnetic field realizations. In theory one can work out the effect of various types of random fields on the used observables; for example, the total intensity of synchrotron emission does not depend on whether the structure of the magnetic field is ordered or completely random. Hence, one could use fudge factors to calculate the observable's mean directly without having to create numerous samples. However, to do a proper uncertainty quantification one must not neglect the so-called *Galactic variance*, a term introduced in Jaffe et al. (2010). This variance measures how strong the influence of the random magnetic field on the individual pixels of an observable's sky-map is. Regions where the influence is high, that is where the observable's variance is high, must be down-weighted when being compared to measured data, in contrast to regions were the randomness of the magnetic field has little influence on the observable's randomness. This makes it again necessary to calculate instances of (θ, m) to be able to construct an estimate for the Galactic variance. See Sec. 2.4.6 for details.

2.4 The IMAGINE Framework

As mentioned in Sec. 2.1, the number of available GMF models and the abundance and quality of observational data are continuously increasing. The goal of IMAGINE is to provide scientists with a standardized framework to analyze the probability distributions of model parameters based on physical observables. In doing so, Bayesian statistics is used to judge the mismatch between measured data and model prediction. It is important to note that IMAGINE's inference is not limited to magnetic field models. Rather, IMAGINE creates an instance of the Milky Way based on a set

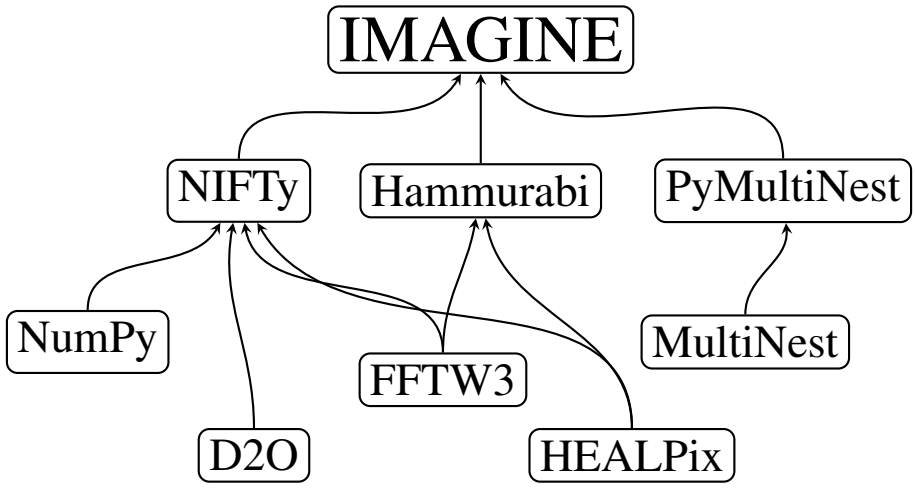


Figure 2.1: The building blocks of the IMAGINE framework.

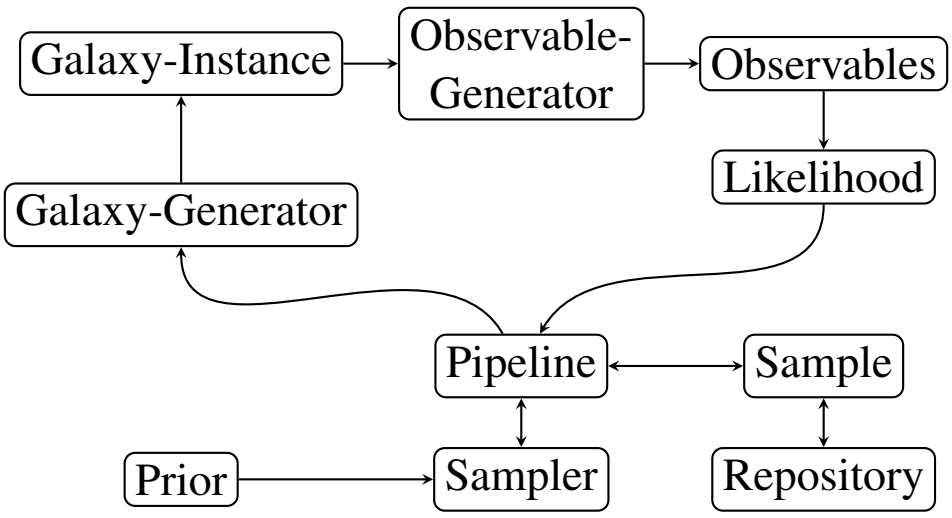


Figure 2.2: The structure of the IMAGINE data processing and interpretation.

of parameters. It is irrelevant for the framework whether the parameters are controlling the appearance of the GMF or, for example, the properties of the free electron density or the dust density. Nevertheless, for the time being, we focus on the GMF and keep all other components fixed.
It is desirable to have a flexible and open framework available when doing parameter inference. The magnetic field in particular must be analyzed indirectly via observables like synchrotron emission, Faraday rotation, dust absorption, or thermal dust emission since there is no direct detection method. This implies that the inference depends on the assumptions that were made regarding further constituents of the Milky Way, for example the free electron density, the population of cosmic rays, or the dust density. Hence, it is very likely that once the self-consistent analysis of a magnetic field model is finished, new insights regarding one or more other components make it necessary to redo the calculations with the new set-up. An example for this is the NE2001 model for the Galaxy's free electron density (Cordes, 2004). Today, updated versions like the YMW16 model (Yao et al., 2017) are available, and it would be very interesting to update parameter estimates from the past. In practice, either this does not happen at all or only with a huge time delay; inference pipelines are usually not made public and the originator may not have the necessary resources anymore. A standardized and open inference framework can help here to speed up scientific progress and make scientific results more transparent.
Imagine is built on the programming language Python to ensure flexibility, and several external libraries for numerical efficiency, cf. Fig. 2.1. Here, Python is primarily used as *glue* to connect individual components and external libraries. A strictly object-oriented design makes it easy to extend its functionality from existing base-classes. The configuration of the inference runs is also done in Python. No configuration files are used as the needs for future derived custom classes can not be foreseen today. Instead, the scientist instantiates the individual components in the main Python script which are ultimately embraced by the Imagine-pipeline.

2.4.1 Components and Overall Structure

The structure of Imagine is shown in Fig. 2.2 and discussed here. The `Pipeline` object plays the key-role as it embraces all other objects and orchestrates their function calls. Its partner is the `Sampler`, with a functional interface for likelihood evaluations. The pipeline hides physical units and scales from the sampler. This means that the former exposes the latter N variables ranging from 0 to 1, each. In this way, the sampler can operate very generically on this unit cube $[0 \dots 1]^N$ without the need to know any internal details on the Galaxy models.
The likelihood evaluation inside the pipeline consists of the following steps. The `Sampler` yields a point from $[0 \dots 1]^N$. Hence, first, the `Galaxy-Generator` maps those variables to physical parameters. Note that N does not need to be the full number of all parameters a model has. All parameters that are not marked as *active* in the `Pipeline` are set to their individually configurable default value. The `Galaxy-Generator` then uses these parameters to generate a certain `Galaxy` model realization. This means to set up all constituents of the abstract Galaxy model including, e.g., the regular and the random magnetic field, the thermal electron density field, the dust-density field, et cetera. Next, the `Observable-Generator`, for example Hammurabi (Waelkens et al., 2009), processes the Galaxy instance and computes physical `Observables`,

like sky-maps of the Faraday depth, synchrotron emission, or thermal dust emission. Those simulated quantities are then compared with measured data by the `Likelihood`, which in turn consists of sub-likelihoods for the individual observables. Together, parameters, Galaxy model, observables and likelihood values form a `Sample`. Finally, the pipeline can be configured to store those `Samples` in a repository for post-processing and caching before the likelihood value is returned to the sampler. Together with the prior, the sampler can then determine which variable configuration should be evaluated next.
As described in Sec. 2.3 the GMF models may consist of a random field component to model the small scale structure stochastically. To deal with the resulting Galactic variance, instead of a single simulation, a set of realizations is created for a certain parameterization. The members of that set are processed in parallel by the `Observable-Generator` such that horizontal scaling, i.e. using multiple computers as a cluster, can be exploited to compensate for the massively increased computational costs one has compared to approaches which ignore the Galactic variance. For this purpose, the Imagine framework uses the software packages NIFTy 3 (Steininger et al., 2017) and D2O (Steininger et al., 2016) for convenient data processing and efficient data parallelization, respectively. D2O is based on the *Message Passing Interface* standard (MPI) (Message Passing Interface Forum, 1994, 1998) and in particular on mpi4py (Dalcín et al., 2005). In combination with OpenMP threading (Dagum and Menon, 1998) of the `Observable-Generator` and the accompanying vertical scaling, Imagine efficiently exploits the parallel architecture of a modern high performance computing cluster as a whole as well as its nodes.

2.4.2 Using Sampling Methods for Uncertainty Quantification

The goal of the Imagine framework is to provide deep probabilistic insights into Galaxy models given observational data. Because of the complexity of the problem, it is not sufficient to calculate point estimates like a maximum a-posteriori approximation. We expect very counter intuitive interdependencies among the model parameters and hence need a thorough uncertainty quantification in order to correctly interpret the observations.
To achieve this, Imagine uses Markov Chain Monte Carlo (MCMC) methods as described by Gelman et al. (2014). As depicted in Sec. 2.2, we seek to perform parameter estimation for a given model as well as Bayesian hypothesis testing when comparing models. Because of its modularity, the Imagine framework can easily make use of the full arsenal of the Bayesian methodology, since it is straightforward to plug in different MCMC libraries and to write interfaces for new ones.
Over the years, various sampling methods based on MCMC have been created. In the following, we briefly discuss the concepts of *Metropolis-Hastings*, *Hamiltonian Monte Carlo* and *Nested sampling*.

2.4.2.1 Metropolis-Hasting Sampling

The Metropolis-Hastings (MH) algorithm (Hastings, 1970, Metropolis et al., 1953) creates a biased random walk through the parameter space. If the random walk is ergodic and its transition probabilities obey detailed balance, $P\left(\vec{x} \to \vec{x'}\right) P(\vec{x}) = P\left(\vec{x'} \to \vec{x}\right) P(\vec{x'})$, the samples gener-

ated by the random walk follow the probability distribution $P(\vec{x})$. Typically, this is achieved by combining a suggestion step with symmetric transition probabilities between any pair of locations from an unbiased random walk with a rejection step that ensures detailed balance, $P_{\text{accept}} = \min\left\{1, P(\vec{x}_{\text{proposed}})/P(\vec{x}_{\text{old}})\right\}$.
During the walk the samples in the chain must decorrelate from the starting position. Hence, the efficiency of an MCMC algorithm is crucial. Choosing a small step length for that purpose indeed means a lower rejection ratio. However, because of the small steps the chain does not move. In contrast, a large step length yields a high rejection ratio and therefore a chain that does not move, either. This relationship gets worse with higher dimensions. An approach to achieve high acceptance rates is Hamiltonian Monte Carlo sampling.

2.4.2.2 Hamiltonian Monte Carlo Sampling

Hamiltonian Monte Carlo (HMC) sampling (also known as Hybrid Monte Carlo sampling) is a unique MCMC algorithm that introduces an auxiliary Gaussian random variable $\vec{p}$ of the same dimensionality as the original parameters $\vec{x}$, cf. Betancourt (2017), Brooks et al. (2011).
The auxiliary variable plays the role of a momentum, the original parameters the role of a position in equations of motion from Hamiltonian mechanics. The negative log-probability corresponds to an energy. A new position in parameter space of position and momentum is generated by integrating the Hamiltonian equations of motion in time. This new position is then treated as the result of a proposal step in the sense of the MH algorithm. Since the Newtonian equations of motions conserve energy the proposed parameters should be accepted 100% of the time, while at the same time being far away from the initial parameters to ensure decorrelation of $\vec{x}$. This makes HMC sampling much more efficient in exploring the parameter space than MH sampling. Although this makes an HMC sampler move much faster than an ordinary MH sampler it has a downside: it requires the gradient field of the desired probability density function (PDF). Especially when dealing with a high number of dimensions, this can pose a problem if finite differencing must be used for gradient computation. Furthermore, some GMF models exhibit discontinuities that result in non-smooth likelihood landscapes, which makes gradients even more problematic. Hence, the Imagine pipeline primarily uses nested sampling which does not require gradient information and allows for model comparison, cf. Sec. 2.2, too.

2.4.2.3 Nested Sampling

Nested sampling is an MCMC method developed by Skilling (2006), that is capable of directly estimating the relation between the likelihood function and the prior mass. It is unique in the fact that nested sampling is specifically made for usage in Bayesian problems, giving the evidence as its primary result instead of the posterior probability.
Nested sampling works with a set of *live-points*. In each iteration, the point that has the lowest likelihood value gets replaced by a new one with a higher likelihood value. As this method progresses, the new points sample a smaller and smaller prior volume. The algorithm thus traverses through nested shells of the likelihood.

2.4.3 Magnetic Field Models

There are many parametric field models in the literature, from relatively simple axisymmetric spirals to complex multi-component models. In addition to defining the parametrized structure of a magnetic field model, estimates for the values of those parameters must be made. Usually, the term *model* is used for both the analytical structure of the magnetic field and for a certain parameter fit. Note that in the context of Imagine, *model* refers to the analytical structure only, since the goal is to investigate its parameter space. It is more straightforward to denote two samples from the same parameter space as belonging to the same *model* instead of constituting distinct models themselves, especially when doing Bayesian model comparison.
In addition to the models' intrinsic complexities, the analyses in the literature also vary with respect to how many observables and datasets were used in the optimization. An example for a rather simple magnetic field model that was fitted to only one observable is the WMAP logarithmic-spiral-arm (LSA) model (Page et al., 2007). In a Galacto-centric cylindrical frame this regular GMF model is given as

$$\begin{aligned} \vec{B}(r,\phi,z) &= B_0 \left[\sin(\psi)\cos(\chi)\vec{\hat{r}} + \cos(\psi)\cos(\chi)\vec{\hat{\phi}} + \sin(\chi)\vec{\hat{z}} \right] , \\ \psi &= \psi_0 + \psi_1 \ln\left(\frac{r}{R_0}\right) , \\ \chi &= \chi_0 \tanh\left(\frac{z}{z_0}\right) , \end{aligned} \tag{2.4}$$

where ψ represents the pitch angle of the magnetic field spiral arm which varies according to ψ_1 and a logarithmic dependency on the radial distance r. R_0 is the distance between the Galactic center and the Sun, and ψ_0 defines the local regular field orientation. The parameter χ corresponds to the off-disk tilting of the Galactic field, and z_0 characterizes the vertical scale height of the poloidal field strength modulation. This simple LSA model for the coherent field was fitted to synchrotron polarization data at 23 GHz by Page et al. (2007). Since the observable intensity of the synchrotron radiation depends on both B_0 and the cosmic ray electron (CRE) density in a degenerate way, only the other three parameters were fitted.
At the more complicated end is the Jansson and Farrar (2012) model (JF12 hereafter) with dozens of parameters describing independent spiral arm segments for regular and random fields and thin and thick disks, an X-shaped halo, and more. JF12 was optimized against both Faraday rotation measures (RM) and synchrotron total and polarized intensity. The model of Jaffe et al. (2013) (and references therein, Jaffe13 hereafter) is in between in terms of number of parameters, with fewer fitted parameters compared to JF12 though originally optimized against the same observables.
Some analyses in the literature include only a coherent field component, while some additionally study the random component from the turbulent ISM in a variety of ways. The JF12 model includes an analytic expression for the average amount of each observable that would result from the given turbulence model. Jaffe13 is notable in that it uniquely includes the effect of the Galactic variance described in Sec. 2.3 explicitly in the likelihood. That analysis used a set of numerical realizations of each model to quantify not only the average amount of emission but

also its variations for a given point in parameter space, which is a necessary step for an unbiased likelihood analysis as described in Sec. 2.4.6.
A further complication to this sort of analysis is how to treat the anisotropy in the random component. As described in Jaffe et al. (2010), from an observational point of view, the GMF can be divided into three components: coherent, isotropic random, and a third variously called the ordered random, the anisotropic random, or the striated component. This third component is expected to arise in the turbulent ISM due to both shocks and shears on large scales. The JF12 model includes a scalar fudge-factor to adjust the synchrotron polarization amplitude from the coherent field to estimate this striated component. In contrast, Jaffe13 explicitly models it by projecting the numerically simulated isotropic random component onto the coherent component to generate an additional anisotropic component. These are complementary methods to model phenomenologically the effect of anisotropic, turbulent, magnetohydrodynamical processes that are computationally expensive to model physically.
On an abstract level, the regular and random components of a magnetic field model are independent. Because of this, Imagine distinguishes them such that the user can combine any regular with any random field model. This is made possible not least through recent developments related to Imagine's primary observable generator Hammurabi.

2.4.4 Hammurabi

The Hammurabi code (Waelkens et al., 2009) was built for simulating Galactic polarized foreground emission, absorption, and polarization rotation. Its core functionality is to produce 2D observables in terms of HEALPix[1] maps (Górski et al., 2005) based on 3D physical field configurations in the Galaxy, e.g., the magnetic, cosmic ray and free electron fields. To analyze various different models, Hammurabi is able to construct physical fields both analytically and numerically. Both regular and random fields covering Galactic scales can be generated with built-in field generators. The observables are produced through line-of-sight integration, including synchrotron and polarized dust emission, Faraday depth, and dispersion measure. In the course of the integration, radiative transfer and polarization rotation are evaluated by accumulating absorption and rotation effects backwards from the observer to the emitter. Technically speaking, the line-of-sight integration is conducted on a set of nested HEALPix shells. Given R as the maximum simulation radius, the n^{th} shell out of N total shells covers the radial distance from $2^{(n-N-1)}R$ to $2^{(n-N)}R$, except for the first shell which starts at the observer. The angular resolution in each shell is set by HEALPix's N_{side} parameter. The n^{th} shell is by default set up with $N_{\text{side}} = 2^{(n-1)}M$, where M represents the lowest simulation resolution at the first shell. Accumulation of observables among shells is carried out by standard HEALPix interpolation. Within each shell, physical quantities are estimated from inside out on discrete radial bins, where the radial bin number is proportional to the radial thickness of the corresponding shell. Since the observables and the physical fields are constructed and evaluated in different coordinate frames, a trilinear interpolation method is used to retrieve information from the physical fields during the line-of-sight integration.

[1] http://healpix.sourceforge.net

2.4.4.1 Random Magnetic Field Generation

While exploring a magnetic field model's parameter space, the likelihood must be evaluated very often. Hence, Hammurabi and especially its random field generator must be swift to preserve computational feasibility. To accomplish Imagine's scientific goals, Hammurabi was recently redesigned; the new version is called Hammurabi X[2] hereafter.
In addition to numerous small to medium sized improvements, Hammurabi X provides two novel solutions for random magnetic field configurations on global, i.e. Galactic, and local, i.e. Solar neighborhood scales, respectively. In the case of global field generation, the focus lies on computational efficiency. Hence, a triple Fourier transform approach is used to do anisotropy enforcement, field strength rescaling and divergence cleaning. For a given power spectrum, $P(k)$, a random magnetic field, $\vec{\tilde{B}}(\vec{k})$, is created in the harmonic Fourier base. The first Fourier transform translates $\vec{\tilde{B}}(\vec{k})$ into the spatial domain $\vec{B}(\vec{x})$. There, anisotropy that may depend on the alignment of the regular magnetic field is introduced. Additionally, a template field strength scaling can be included in terms of a function $S(\mathbf{x})$ as

$$\vec{B}(\vec{x}) \rightarrow \vec{B}(\vec{x}) \sqrt{S(\vec{x})}. \tag{2.5}$$

An example for such a scaling function is

$$S(\vec{x}) = S(r, \phi, z) = \exp\left(-\frac{r}{h_r} - \frac{|z|}{h_z}\right), \tag{2.6}$$

where h_r and h_z are the characteristic scales of the radial and vertical profiles, respectively. The second Fourier transform translates the re-profiled field $\vec{B(x)}$ back into harmonic space, where a Gram-Schmidt procedure is used to clean up the divergence:

$$\vec{\tilde{B}} \rightarrow \frac{\vec{\tilde{B}} - (\vec{k} \cdot \vec{\tilde{B}})\vec{k}/k^2}{\left|\vec{\tilde{B}} - (\vec{k} \cdot \vec{\tilde{B}})\vec{k}/k^2\right|} |\vec{\tilde{B}}| \,. \tag{2.7}$$

Finally, a last Fourier transform is applied to retrieve the desired $\vec{B}(\vec{x})$. Hence, the anisotropic random magnetic field is drawn from a one-dimensional power spectrum which in contrast corresponds to statistical homogeneity and isotropy. Breaking the isotropy with subsequent divergence cleaning results in a field that does not precisely obey the original power spectrum $P(k)$ anymore. In contrast to the global method, for local scale simulations a strict method including vector decomposition of the power spectrum tensor is available in Hammurabi X. This method is not prone to the inaccuracies described above. Details with respect to the local field generator are beyond the scope of this paper but are available in the release publication of Hammurabi X (Wang et al., in prep.).

[2] https://bitbucket.org/hammurabicode/hamx

2.4.5 Observables

Magnetic fields cannot be measured directly. Instead, their properties need to be inferred indirectly via *observables* (also referred to as *tracers*). The most commonly used observables include Faraday rotation, synchrotron radiation, dust absorption and emission to probe properties of the GMF, as well as dispersion measure to probe the thermal electron density. These observables are briefly described below.

2.4.5.1 Faraday Rotation

Faraday rotation can be described as a double refraction effect when linearly polarized light travels through a magnetized, ionized medium. The polarization angle of the Faraday rotation is given by

$$\theta = \theta_0 + \Phi\lambda^2, \tag{2.8}$$

with θ being the observed polarization angle, θ_0 the original polarization angle, Φ the Faraday depth and λ the wavelength of the light ray. The Faraday depth is given by a line-of-sight integral over a distance l_0 to an observer,

$$\frac{\Phi}{\mathrm{rad\ m^{-2}}} = 0.812 \int_{l_0}^{0} \frac{n_e(l)}{\mathrm{cm^{-3}}} \frac{B_\parallel(l)}{\mu\mathrm{G}} \frac{\mathrm{d}l}{\mathrm{pc}}, \tag{2.9}$$

with $n_e(l)$ and $B_\parallel(l)$ being the thermal electron density and strength of the parallel magnetic field, respectively, at distance l away from the observer. Φ is positive (negative) when the magnetic field is pointing towards (away from) the observer by convention. Assuming the emitted polarization angle θ_0 is constant for a specific source, the Faraday depth gives information about the average strength of the line-of-sight (i.e., parallel) component of the magnetic field.

2.4.5.2 Synchrotron Radiation

The synchrotron radiation that is used for the GMF inference is caused by the acceleration of relativistic electrons within this very magnetic field. This linearly polarized electromagnetic radiation is emitted radially to the acceleration. Its intensity is given by

$$I_s \propto N(E)\, B_\perp^x, \tag{2.10}$$

with $N(E)$ being the density of relativistic electrons in the relevant energy range, E. The index x depends on the energy spectrum of these electrons, typically $x \approx 1.8$. Even though the intensity of synchrotron radiation is degenerate with other emission components, like free-free and spinning dust in the microwave band, Stokes Q and U still provide information regarding the magnetic field. The other components are assumed to be unpolarized. The random components of the GMF depolarize the synchrotron radiation; see the classic paper by Burn (1966). The strength of this depolarization depends on the degree of ordering in the field, which can be written as $B_{\perp,r}^2/B_\perp^2$ with $B_{\perp,r}$ being the regular part of $B_\perp$. Using the Stokes I, Q, and U together, we

can calculate the strength of the magnetic field perpendicular to the line-of-sight $B_\perp$ (using the intensity I) and the fraction of the total magnetic field that is regular $B^2_{\perp,r}/B^2_\perp$ (using the polarized intensity PI). This makes it a useful tool for studying the random component of magnetic fields. In addition, the lines-of-sight for an extended source with a per se constant polarization angle traverse space with a different field configuration each. This results in varying polarization angles within the instrument beam, known as Faraday beam depolarization which provides further information.

2.4.5.3 Dust Absorption and Emission

Starlight polarization is caused by rotating dust grains absorbing certain polarizations of light. In a magnetic field, a dust grain tends to align its long axis perpendicular to the direction of the local magnetic field (see Davis and Greenstein (1951) and references therein). If the field is perpendicular to the line-of-sight, certain polarizations of the light-ray get blocked, viz. dust absorption of background starlight. The resulting observed light-ray is thus polarized, which gives information about the direction of the magnetic field perpendicular to the line-of-sight between the observer and the star.
The approach above works well for low-density dust clouds. In high-density dust clouds, the probability that a light-ray gets completely absorbed along the way is fairly high. However, dust heats up if it absorbs a lot of radiation, which in return will be re-emitted in the infrared. This emitted infrared light is also polarized according to the dust grain's geometry, viz. polarized thermal dust emission. Since as already mentioned the dust grains are aligned in the magnetic field, the polarized dust emission provides complementary information about the direction of $B_\perp$.

2.4.5.4 Dispersion Measure

When a neutron star forms in the course of a supernova collapse the preserved angular momentum causes the neutron star to rotate rapidly. Along the neutron star's magnetic axis, a highly focused beam of radiation is emitted, and the rotational and magnetic axes are not necessarily the same. Since the beam is highly focused, from an observer's point of view this may result in a blinking pattern, which is why those stars are called pulsars. The group and phase velocity of the emitted radiation are not the same in the interstellar medium because of its ionized components, mainly free electrons. Because of this, higher frequencies arrive earlier than lower ones. This extra time delay added at a frequency ν is given by

$$t(\nu) = \frac{e^2}{2\pi m_e c}\frac{\mathrm{DM}}{\nu^2}, \tag{2.11}$$

with DM being the so-called dispersion measure. The DM itself is given by the line-of-sight integral,

$$\mathrm{DM} = \int_0^{l_0} n_e(l)\,\mathrm{d}l. \tag{2.12}$$

If one has information on the thermal electron density, the DM solely depends on the distance l_0 between the source and the observer.
The dispersion measure, although it does not give any information on magnetic field properties, is still a very important observable. With DM, the thermal electron density can be inferred, which in turn is needed for the inference of Faraday rotation, as described in Ekers et al. (1969). Using a combination of Faraday rotation, synchrotron radiation, starlight polarization and dispersion measure data is key for inferring the constituents of the Galaxy.

2.4.6 Likelihood

The likelihood is the probability $P(d|\theta, m)$ to obtain the data d from a measurement under the assumption that reality is given by the model m that in turn is configured by the parameters θ. It is the key element to rate the probability of a stochastic sample. Assuming the generic case of a measurement with linear response function R of a signal s which involves additive noise n, the corresponding equation for the data d reads

$$d = R(s) + n. \tag{2.13}$$

If the measurement device is assumed to exhibit Gaussian noise characteristics with a covariance matrix N, i.e.

$$n \hookleftarrow \mathcal{G}(n, N) = \frac{1}{|2\pi N|^{1/2}} \exp\left(-\frac{1}{2} n^\dagger N^{-1} n\right) \tag{2.14}$$

the log-likelihood for a simulated signal that is the result of the evaluation of a model m with parameters θ, i.e. $s' = m(\theta)$, to have produced the measured data d is

$$\mathcal{L}(d|s') = -\frac{1}{2} \left(d - R(s')\right)^\dagger N^{-1} \left(d - R(s')\right) - \frac{1}{2} \ln\left(|N|\right). \tag{2.15}$$

In the context of Imagine, as discussed in Sec. 2.3, the GMF models posses random components that are described by (m, θ) only stochastically. Marginalizing over those random degrees of freedom results in a modification of the effective covariance term in Eq. (2.15), namely that the Galactic variance must be added to the data's noise covariance. During the further discussion we consider the following quantities:

- The individual GMF samples within an ensemble of size N_{ens} are named B^i, with $i \in [1, N_{\text{ens}}]$.
- The process of creating observables from B^i is encoded in the response R.
- The simulated observables are denoted by $c^i = R(B^i)$.
- The measured observable's data is named d.

Denoting furthermore the data's noise covariance by A, the Galactic covariance by C, and the dimensionality of observables by N_{dim} the log-likelihood reads

$$\mathcal{L}(d|c) = -\frac{1}{2}(d - \bar{c})^\dagger (A + C)^{-1} (d - \bar{c}) - \frac{1}{2} \ln\left(|A + C|\right) \tag{2.16}$$

with the ensemble mean of c

$$\bar{c} = \frac{1}{N_{\mathrm{ens}}} \sum_{i=1}^{N_{\mathrm{ens}}} c^i. \tag{2.17}$$

As discussed in Sec. 2.3 the Galactic covariance C reflects the fact that the observables posses an intrinsic variance because of the random parts of the GMF. For example, the higher the intrinsic variance, the more the likelihood will be flattened by the $(A + C)^{-1}$ term. This means that the likelihood is less responsive to deviations from the ensemble mean for regions of high variance. Hence, there is the risk of overestimating random field contributions, since they are favored by the likelihood. However, this is compensated by the second summand in Eq. (2.16): the covariance matrix' log-determinant $\ln(|A + C|)$. In Eq. (2.15) the covariance matrix and thus its determinant are constant and therefore can be neglected as we are not interested in the absolute scales of the likelihood. In contrast, for Eq. (2.16) we have to consider it as this determinant varies from point to point in parameter space.
The Galactic covariance C is not known, hence, we must estimate it. A classic approach for C is to evaluate the dyadic product of the samples' deviations from their mean:

$$C_{\mathrm{cl}} = \frac{N_{\mathrm{dim}}}{N_{\mathrm{ens}}} \sum_{i=1}^{N_{\mathrm{ens}}} (c^i - \bar{c})(c^i - \bar{c})^\dagger = \frac{1}{N_{\mathrm{ens}}} \sum_{i=1}^{N_{\mathrm{ens}}} u^i u^{i\dagger} \tag{2.18}$$

with

$$u^i = \sqrt{N_{\mathrm{dim}}} \left(c^i - \bar{c} \right). \tag{2.19}$$

Since the number of samples in an ensemble is much smaller than the number of dimensions this classical estimator for the covariance matrix is insufficient. Most of its eigenvalues are zero, making an operator-inversion impossible. Hence, it is better to use a sophisticated estimator using a shrinkage target (e.g., a diagonal matrix) and a shrinkage factor. Here, we use the *Oracle Approximating Shrinkage* (OAS) estimator by Chen et al. (2011):

$$C = \mu\rho \, \mathbb{1} + (1 - \rho) \, C_{\mathrm{cl}}. \tag{2.20}$$

The specific quantities needed to compute the OAS estimator are

$$\mu = \frac{1}{N_{\mathrm{dim}}} \mathrm{tr}\left(C_{\mathrm{cl}}\right) = \frac{1}{N_{\mathrm{dim}} N_{\mathrm{ens}}} \sum_{i=1}^{N_{\mathrm{ens}}} u^{i\dagger} u^i \tag{2.21}$$

$$a = \mathrm{tr}\left(C_{\mathrm{cl}}^\dagger C_{\mathrm{cl}}\right) = \frac{1}{N_{\mathrm{ens}}^2} \sum_{i=1}^{N_{\mathrm{ens}}} \sum_{j=1}^{N_{\mathrm{ens}}} \left(u^{i\dagger} u^j\right)^2 \tag{2.22}$$

$$r = \min\left\{1, \frac{(1 - 2/N_{\mathrm{dim}})\, a + N_{\mathrm{dim}}^2 \mu^2}{(N_{\mathrm{ens}} + 1 - 2/N_{\mathrm{dim}})\,(a - N_{\mathrm{dim}} \mu^2)}\right\}. \tag{2.23}$$

In the likelihood one needs to apply the inverse of the sum of A and C, $(A + C)^{-1}$. Since we do not know a basis in which $A + C$ is diagonal, the inversion of this operator is a nontrivial task.

However, because of its structure, we can use the Sherman-Morrison-Woodbury matrix identity (Sherman and Morrison, 1950, Woodbury, 1950) by re-sorting

$$A + C = (A + \mu r\, \mathbb{1}) + (1 - r)C_{\mathrm{cl}} = B + VV^{\dagger} \tag{2.24}$$

with

$$B = A + \mu r\, \mathbb{1} \quad \text{and} \quad V = \sqrt{\frac{1-r}{N_{\mathrm{ens}}}}\, U. \tag{2.25}$$

Namely,

$$(B + VV^{\dagger})^{-1} = B^{-1} - B^{-1}V(\mathbb{1} + V^{\dagger}B^{-1}V)^{-1}V^{\dagger}B^{-1}. \tag{2.26}$$

With this formula only a matrix of size N_{ens}^2 instead of N_{dim}^2 must be inverted.
For computing the log-determinant $\ln(|A + C|)$ one could use the result of the OAS estimator and apply the generalized form of the matrix determinant lemma (Harville, 2008) to it. Its structure is closely related to the Sherman-Morrison-Woodburry matrix identity: it turns the problem into the calculation of the determinant of a matrix of size N_{ens}^2 instead of N_{dim}^2. For our case it reads:

$$|A + C| = \left|B + VV^{\dagger}\right| = |B| \cdot \left|\mathbb{1} + V^{\dagger}B^{-1}V\right| \tag{2.27}$$

However, the OAS estimator has been designed for and is good at approximating covariance matrices in terms of quadratic forms; using it for determinant estimation yields rather poor results. And in fact, it can be shown that it is not possible to construct a general purpose estimator from covariance matrix samples if the number of samples is lower than the number of dimensions (Cai et al., 2015). Nevertheless, heuristic as well as Bayesian estimators have been developed trying to cover special cases, as for example the case of sparse or diagonally dominated covariance matrices (Fitzsimons et al., 2017, Hu et al., 2017). For the time being we approximate the determinant $|A + C|$ by its diagonal:

$$\ln(|A + C|) \approx \frac{1}{N_{\mathrm{dim}}}\mathrm{tr}\left[\ln\left(A + \frac{1}{N_{\mathrm{ens}}}\sum_{i=1}^{N_{\mathrm{ens}}}\left(c^i - \bar{c}\right)^2\right)\right]. \tag{2.28}$$

This approximation serves the purpose of regularizing the random magnetic field strength. Future improvements could include the usage of one of the widely used shrinkage estimators as discussed in Hu et al. (2017). They work similarly to the OAS estimator, though exhibiting shrinkage coefficients and targets tailor made for covariance determinant approximation. For those, then Eq. (2.27) can be used for efficient computation. In either case, the inversion of the covariance matrix as well as the calculation of its determinant can be done explicitly, if approximately, which therefore allows us to evaluate the ensemble likelihood in Eq. (2.16) efficiently.

2.5 Application

In the following we discuss possible usage scenarios of the Imagine pipeline. Regardless of parameter estimation or model comparison, first, a Galaxy model must be set up. Below we

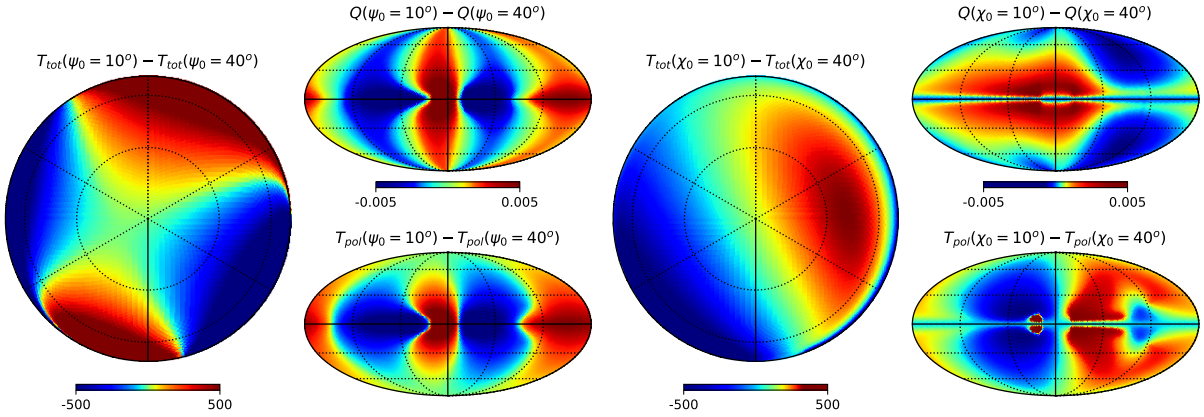


Figure 2.3: Simulated synchrotron emission difference maps (including 408 MHz total intensity $T_{\rm tot}$ at northern hemisphere, 30 GHz Stokes Q and polarized intensity $T_{\rm pol}$ in mK) with different ψ_0 or χ_0 settings. ψ_0 has influence mainly along Galactic longitude while χ_0 affects more the latitude direction.

will use Imagine to analyze the following scenario. Our Galaxy model consists of the WMAP logarithmic-spiral-arm (LSA) magnetic field model (Page et al., 2007) in combination with an isotropic Gaussian random field as described in Sec. 2.4.4.1. In Hammurabi X, the random field's normalization is chosen such that its RMS field strength at the Sun's position is given by τ. We denote the spectral index of the random field's power spectrum as α. Furthermore, we choose the YMW16 model (Yao et al., 2017) for the thermal electron density. Here, our goal is to infer the parameters of the magnetic field model, so the thermal electron density we assume to be fixed. For the input data, we consider polarized synchrotron emission at 1.41 GHz (Stokes Q and U) following Wolleben et al. (2006), 408 MHz (Stokes I) and at 30 GHz (Stokes Q and U) following (Planck Collaboration et al., 2016a), and the Faraday depth map following Oppermann et al. (2012).

2.5.1 Mock Data Based Tests

It is advisable, before starting a large likelihood exploration, to check if the chosen observables (tracers) are sensitive to the model parameters that are about to be inferred. In principle, all observables used here are sensitive to the GMF configuration especially near the Solar neighborhood. In terms of the WMAP LSA model, the influence of ψ_0 on polarized synchrotron emission is expected to be the most noticeable feature, cf. Fig. 2.3. By definition of the model, ψ_1 has greater influence than ψ_0 and χ_0 on the field's configuration when $r < R_0/e$. We therefore expect the observables to be more sensitive to ψ_1 at low Galactic latitudes where line-of-sight integration accumulates information through the Galactic center. However, Faraday depolarization at low Galactic latitudes and low frequencies diminishes constraining power of polarized synchrotron emission on ψ_1.

2.5.1.1 Mock Data Configuration

During the development of the IMAGINE framework, initial mock data tests were performed on the base of the JF12 model and are described in van der Velden (2017). One result of that work was an increased appreciation for the difficulty working with such a complex model. The first (ω_1) is solely a regular WMAP field, while the second (ω_2) additionally possesses a random component as described in Sec. 2.4.4.1. To test the pipeline with a parameter set that is as realistic as possible, we used the best fit estimates for the WMAP LSA model given in Page et al. (2007), except for ψ_1 which would be 0.9°. To conduct proper tests it is helpful if the mock data generating parameter values are not located at the boundaries of parameter space, so we set ψ_1 to 7.95°. Since B_0 is not given in Page et al. (2007) we use Beck and Krause (2005) as a reference and set it to 6 μG. Furthermore, for ω_2 we set the random magnetic field's strength around the Sun τ to 2 μG. The spectral index is set to $\alpha = 1.7 \approx 5/3$ (Kolmogorov). The precise mock data parameter values and the boundaries of the tested parameter volume are given as follows

$$B_0 = 6.00 \in [0.3, 11.7]\ \mu\text{G} \tag{2.29}$$
$$\chi_0 = 25.0 \in [1.0, 49.0]^\circ \tag{2.30}$$
$$\psi_0 = 27.0 \in [6.0, 48.0]^\circ \tag{2.31}$$
$$\psi_1 = 7.95 \in [0, 15.9]^\circ \tag{2.32}$$
$$\tau = 2.00 \in [0.2, 3.8]\ \mu\text{G} \tag{2.33}$$
$$\alpha = 1.7 \in [0.2, 3.2] \tag{2.34}$$

After processing the mock magnetic fields with Hammurabi X, we add individual random noise samples with the variances given in Oppermann et al. (2012), Planck Collaboration et al. (2016a), Wolleben et al. (2006) to the calculated observables. For the Oppermann Faraday depth map there is an uncertainty map available which is based on a Bayesian Wiener filter reconstruction. Since the pixel-wise noise is uncorrelated on small scales, we downscale the uncertainty map to $N_{\text{side}} = 32$ to estimate the total noise power correctly. Since we produce the sample simulations with $N_{\text{side}} = 32$ as well, no further adaption of this noise map is necessary. For the Planck and Wolleben synchrotron (Stokes Q and U in each case) we take a constant statistical uncertainty of 2.12 μK (Planck Collaboration et al., 2016b, Tab. 10) and 12 mK (Wolleben et al., 2006, Sec. 5.2). For the Stokes I map at 408 MHz an uncertainty map is given. We downgrade all four data sets to our simulation resolution of $N_{\text{side}} = 32$.

For the inference below, the ensemble size was set to $N_{\text{ens}} = 64$. Our tests showed that for the resolution $N_{\text{side}} = 32$ this is the ensemble size where the classical covariance term in the ensemble likelihood becomes dominant over the shrinkage target, i.e. r falls below 0.5, cf. Sec. 2.4.6. To make likelihood maximization and sampling possible, it is also necessary to stabilize the likelihood by fixing the ensemble member's random seed. This introduces a bias, which we found, however, to be already negligible in the case of $N_{\text{ens}} = 64$ compared to the emerging Galactic variance. In the future, one could try to enhance existing sampling techniques already including simulated annealing (Kirkpatrick et al., 1983) to become capable of treating the noisy likelihood surface directly.

	B_0 [μG]	ψ_0 [°]	ψ_1 [°]	χ_0 [°]
Mock values	6.0	27.0	7.95	25.0
Reconstruction	5.999	26.99	7.943	25.003

Table 2.1: Log-likelihood maximizing parameter values for mock data ω_1 inferred with a Nelder-Mead optimizer; showing the first significant digit of deviation.

2.5.1.2 Regular Magnetic Field

First, we consider the first mock data set ω_1 that does not contain random field components. For this data set we perform one-dimensional likelihood scans through the parameter space, as this is a systematic way to check the observables' sensitivity with respect to the model parameters. In doing so, we vary only one parameter at a time while keeping all others fixed to the mock data's generating values.

Fig. 2.4 shows how well the different observables yield peaks in the likelihood. Since there is no random magnetic field, the ensemble likelihood simplifies to a standard χ^2 likelihood. Several comments are in order. First, one sees that the total log-likelihood exhibits clear peaks very near to the *true* mock data values for all four WMAP LSA parameters. Second, as expected, B_0 shows the strongest dependence, followed by ψ_0 and χ_0; ψ_1 affects the observables as well but much more weakly than the other three parameters. Third, it is remarkable that for all parameters the total log-likelihood is dominated by synchrotron emission Stokes Q & U at 30 GHz and Stokes I at 408 MHz. Faraday rotation also adds some information, but synchrotron data at 1.41 GHz yields four to six orders of magnitude weaker signals in the log-likelihood. This is because of the signal-to-noise ration which is better for the Planck than for the Wolleben data set. Furthermore, due to the depolarization effects that have a huge impact on low-frequency polarized synchrotron data, we see sharp peaks for 1.41 GHz synchrotron data, as the morphology of the observable map tremendously changes when varying the magnetic field. If the GMF were regular, this would allow us to constrain the GMF parameters very precisely. However, the presence of random magnetic fields and Faraday depolarization effects render this frequency uninformative for this analysis. We therefore exclude the 1.41 GHz data from the subsequent analysis.

After examining the one-dimensional parameter scans, we then check whether it is possible to infer the input parameters from the mock data set with simple minimization. Tab. 2.1 shows the values a Nelder-Mead minimizer (Nelder and Mead, 1965) yields when operating with the mock data set ω_1. As mentioned above, only Faraday depth and synchrotron data at 408 MHz and 30 GHz were used according to the insights we drew from the parameter scans.

The minimizer is able to reliably find the correct parameter values, which suggests that the likelihood surface is well-behaved throughout the parameter space volume and not only along the optimum-intersecting axes. Note that in general it is advisable to use a gradient-free minimization scheme like Nelder-Mead due to possible non-smooth transitions that are particularly part of more complex magnetic field models.

Finally, we use PyMultiNest (Buchner et al., 2014) to explore the likelihood surface of the

mock data ω_1. Fig. 2.5 shows the marginalized probability density functions as well as pairwise correlation plots. As expected, B_0 is inferred with the highest precision; followed by ψ_0 and χ_0, and finally ψ_1. In the course of this, the addition of mock noise causes the inferred parameter means to be shifted with respect to the true values. Different random seeds for the noise yield varying offsets. The likelihood is insensitive to these deviations, however, since we knew the true noise covariance matrix and take it into account. With the high signal-to-noise ratios, the 2σ intervals are narrow and cover the ω_1's *true* parameter values. This means that the likelihood is consistent with the process of mock data creation and mock noise generation.

2.5.1.3 Regular and Random Magnetic Field

In the following we repeat the steps from the previous section for mock data set ω_2: scanning the parameter space, finding optimal parameter values with Nelder-Mead minimization and doing a full sampling with PyMultiNest. Fig. 2.5 shows that including a random magnetic field reduces the sensitivity of the ensemble likelihood considerably. In contrast to Fig. 2.4, now the log-likelihood values vary over one to three instead over eight orders of magnitude. As before, the signal for B_0 is strongest, followed by ψ_0 and χ_0, and finally ψ_1. With respect to the parameters of the random magnetic field component we see that τ, the parameter for the random magnetic field's strength, and α, the random field's spectral index, exhibit a slight peak at their true values. However, as foreseen in Sec. 2.4.6, τ's likelihood flattens significantly for large values. The fact that for α the likelihood has its maximum near the true mock data value is the incidental result of combining contrarily biased Faraday rotation and synchrotron radiation likelihoods. It should be noted that such shifts are not unexpected, since the mock data include a single realization of the Galactic and noise variance that can cause such chance alignment with slightly shifted parameters. Finally, we note that Faraday rotation data would not be able to constrain τ and α reasonably. The total likelihood's shape around the true mock data value is rather flat for τ and α. Their influence on the likelihood is comparably small in this mock scenario, but would increase with the strength of the random field component; here the setting is $B_0 = 6\,\mu$G vs. $\tau = 2\,\mu$G. ψ_1, τ, and α get traced by the observables – at least slightly – which is why we keep them for the further inference. At this point the importance of this sensitivity analysis becomes evident, as we can draw the following conclusions: If we find a parameter which has completely negligible or even misleading influence on the likelihood it should be excluded from inference. It would solely increase the dimensionality of the problem and with respect to minimizers and samplers behave in the best case as a noisy contribution and therefore disturb convergence.
For completeness, we visualize the importance of the regularizing determinant in Eq. (2.16). Fig. 2.6 shows that without the determinant the ensemble likelihood favors too high random field strengths and spectral indices.
As in Sec. 2.5.1.2, we continue by inferring the parameter values of ω_2 using a Nelder-Mead minimizer. Tab. 2.2 shows the results of the optimization whereby we see that as expected the accuracy is significantly lower than without a random magnetic field component, cf. Tab. 2.1. One can also see the trend that τ gets overestimated, which already became apparent in the parameter scan, cf. Fig. 2.5.
Finally, Fig. 2.7 shows the marginal plots based on a PyMultiNest run on the mock data set ω_2.

	B_0 [μG]	ψ_0 [°]	ψ_1 [°]	χ_0 [°]	τ [μG]	α
Mock values	6.0	27.0	7.95	25.0	2.0	1.7
Reconstruction	6.063	26.86	8.38	23.9	2.11	1.698

Table 2.2: Log-likelihood maximizing parameter values for mock data ω_2 inferred with a Nelder-Mead optimizer; showing two significant digits of deviation.

First, we recognize that the Galactic variance caused rather broad uncertainties. Nevertheless, the uncertainty intervals are highly reasonable: for example, although the sample mean value for B_0 lies rather precisely at 6 μG, the maximum likelihood value is significantly shifted to the right. Furthermore, one sees that the Galactic variance washes out almost all predictive power on ψ_1. The fact that the sample mean matches the mock data's generating parameter is mainly due to the fact that the true value is at the center of the prior volume. As seen before, τ gets overestimated, while 2 μG still lies within the 2σ interval. Interestingly enough, looking at the joint probability density plot for τ and α, one sees that for larger τ also larger α become more likely. This is on the one hand an indicator for an unsurprising degeneracy between the total strength and the spectral index. On the other hand, we expect that the predictive power on one of the parameters can be increased by fixing the other by the use of strong prior information.
Note, that a naive χ^2 likelihood which, unlike the ensemble likelihood, does not reflect the Galactic variance massively underestimates the uncertainties introduced by the random magnetic field. Fig. 2.8 shows the result of PyMultiNest maximizing a χ^2 likelihood on the ω_2 mock data set. The spectral index is pushed to a small value of $\alpha = 0.202$ making the random magnetic field rather white. As a consequence, in the ensemble mean the influence of the random magnetic field maximally cancels out as the set of samples in the ensemble is finite. The other parameters then heavily over-fit the variations in the mock-data which come from its specific random magnetic field realization. This illustrates the importance of taking the Galactic variance into account when doing model parameter inference.

2.5.1.4 Model Comparison

One strength of sampling methods like MultiNest is that they produce an estimate for the *evidence*. As discussed in Sec. 2.2, the evidence is crucial for model selection. To illustrate the procedure, we set up the following scenario: Given the prevailing mock data set ω_2, we compare two models that are both trivial versions of the WMAP LSA model. The only free parameter is now B_0. For model M_1 the values for the *hidden* parameters are equal to those of the mock data, while for model M_2 they are fixed to $\psi_0 = 3.0°$, $\psi_1 = 25.0°$, and $\chi_0 = 7.0°$. Tab. 2.3 shows that the log-evidence for M_1 is significantly higher than for M_2, corresponding to a massive Bayes factor of $R = 2.47 \cdot 10^{10}$. But besides the quality of fit the evidence also takes the model's complexity into account. The fewer parameters a model has, the smaller is its total parameter space volume. Hence, even if a rather complicated model has a better best-fit estimate than a simpler one, if over-fitting occurs its evidence value will be worse. Tab. 2.3 also shows the log-evidence

	Log-Evidence	B_0 [μG]	ψ_0 [°]	ψ_1 [°]	χ_0 [°]
Mock values		6.0	27.0	7.95	25.0
M_0	13.66 ± 0.20	6.10 ± 0.253	26.89 ± 3.42	8.43 ± 4.39	23.92 ± 6.12
M_1	15.09 ± 0.18	6.10 ± 0.212	27.0*	7.95*	25.0*
M_2	−8.84 ± 0.15	6.229 ± 0.232	3.0*	25.0*	7.0*

Table 2.3: Sample mean and log-evidence values from PyMultiNest for different WMAP LSA plus random magnetic field models based on the mock data set ω_2. At M_0 all four WMAP parameters are flexible; at M_1 and M_2 only B_0 is adjustable. An asterisk (*) indicates a fixed value. In any case, the random magnetic field's parameters were kept at their mock data's default value $\tau = 2$ μG, and $\alpha = 1.7$

for the full four-parameter WMAP LSA model (M_0). The log-evidence for M_0 lies in between those of M_1 and M_2; the Bayes factor between M_0 and M_1 is $R = 4.18$, which means that there is *substantial* evidence that M_1 is more likely (Jeffreys, 1998). Thus, one sees the penalty coming from M_0's larger parameter space volume compared to M_1. However, the improvements of a better parameter fit may compensate for this penalty as the comparison with M_2 shows.

2.5.2 Application to Real Data

In Sec. 2.5.1, we verified that the Imagine pipeline produces self-consistent results for the WMAP LSA model in combination with the chosen observables. Now we analyze the likelihood structure for the real synchrotron data at 408 MHz and 30 GHz (Planck Collaboration et al., 2016a), and the Faraday depth data (Oppermann et al., 2012)[3] Generally, it is advisable to thoroughly prepare the input data by masking regions in the sky obviously perturbed by local phenomena, for example supernova remnants. However, for this paper this is beyond the scope, as the goal is to illustrate the concepts behind Imagine rather than producing high-precision estimates. First, we use the synchrotron data to constrain the parameters of the purely ordered WMAP LSA model; so far no random fields are included in the magnetic field nor in the likelihood. The result is shown in Fig. 2.9.

The resulting uncertainties are quantitatively consistent with the mock-data results, cf. Fig. 2.5. Qualitatively speaking, B_0 is determined with the highest accuracy, followed by ψ_0 and χ_0, and finally ψ_1. Also the inferred magnetic field strength $B_0 = 4.44$ μG is of a reasonable order of magnitude (Han, 2006, Ruiz-Granados et al., 2010). $\psi_0 = 6.69°$ lies within the wide range of estimates one can find in literature, e.g., 8° (Beck, 2001, Han, 2006) and 35° (Page et al., 2007). However, $\psi_1 = 34.4°$ and $\chi_0 = 78.6°$ are far off from the best-fit values given in Page et al. (2007), namely $\psi_1 = 0.9°$ and $\chi_0 = 25°$. This, in combination with the very small uncertainties, indicates that an inference neglecting the influence of random components in the magnetic field as well as the likelihood is making matters too easy.

[3]For using Imagine in production it is advisable to use the raw data compiled by Oppermann et al. (2012) as this ensures that there is no alteration of the noise information by a Wiener filter.

When using Faraday depth instead of synchrotron data (Oppermann et al., 2012) for a parameter fit, cf. Fig. 2.10, the limited capabilities of the WMAP LSA model become clear. Even though the WMAP LSA model was designed for fitting synchrotron radiation but not Faraday depth data, it is nevertheless remarkable how incompatible they are. Not only are the estimates for $\psi_0 = 280^\circ$ and $\psi_1 = 330^\circ$ far off their reference values, the magnetic field strength is pushed to values near zero. The latter indicates a general incompatibility between the model and the data. Furthermore, the best fit value for B_0 is negative, which in our case means that the direction of the magnetic field is reversed compared to Page et al. (2007). Fig. 2.11 illustrates the issue. In the data, one can locate a dipole as well as a quadrupole moment both being aligned with the Galactic plane. Because of its simple structure, the WMAP LSA model cannot account for the double anti-axisymmetric quadrupole structure. That is expected, and if such a feature is needed one can use more complex models like JF12 (Jansson and Farrar, 2012) or Jaffe13 (Jaffe et al., 2013). But, beyond this, Fig. 2.11 in combination with Fig. 2.12 reveals that the likelihood peak at $\psi_1 = 330.2^\circ$ corresponds to a configuration where the model exhibits field reversals to fit the structure in the Galactic plane. Although the Faraday rotation map compares well to the data, such a parameter configuration is the result of a simple model fitted to a complicated dataset and is not necessarily the most physically realistic solution. This demonstrates another possible pitfall and also how important it is to incorporate physical priors for the model parameters when doing a real-life analysis. Imagine provides the structure for comprehensive studies that are not only built on powerful algorithms such as MultiNest that will find parameter estimates in any case but also regularizes them and points out problems in the reconstruction. Furthermore, irrespective of the quadrupole, for the reference parameter values also the dipole does not fit; it has the wrong sign. This means that the overall field orientation itself in the WMAP LSA model cannot be correct. This is a fact that does not become apparent when solely using synchrotron data, since even though synchrotron emission is sensitive to the magnetic field's direction, it is not to its orientation. Using the Imagine pipeline for parameter estimation, it is economic to include various data sets from different observables, since the Imagine data repository is open and will grow through collaborative contribution. Such obvious contradictions can then be avoided by a more comprehensive approach.

Going a step further, we try to find parameter estimates for the WMAP LSA plus random magnetic field model that we previously used for the mock data tests in Sec. 2.5.1. The results are shown in Fig. 2.13. With respect to the random magnetic field, we limit ourselves to the inference of τ, the strength of the random magnetic field. For the sampling we used a wide prior volume, especially for the angular parameters ψ_0, ψ_1 and $\chi_0 \in [0, 360]^\circ$, each. Note that only ψ_0 is a truly circular parameter, cf. Eq. (2.4), and thus was setup as such in PyMultiNest. Comparing the results of a fit based purely on synchrotron emission, given in Fig. 2.13, to the scenario in Fig. 2.9 with only an ordered field provides several insights. First of all, we see how approaches that neglect the Galactic variance tremendously underestimate uncertainties when doing parameter estimation. The estimate for B_0 is smeared out the least, but the predictive power for ψ_0, ψ_1 and χ_0 disappears when taking the Galactic variance into account correctly. In the light of the above, it is noteworthy how clear the prediction for the strength of the random magnetic field turns out to be. All in all, despite their weak predictive power, the results shown in Fig. 2.13 are consistent with those in Fig. 2.9. For the former, the regular magnetic field strength B_0 is smaller

compared to the latter as now the random magnetic field also contains magnetic field power in τ. Also note the reasonable anti-correlation between B_0 and τ. The overall order of magnitude of τ is compatible with Han (2006). Since ψ_0 is a circular parameter it is necessary to consider circular definitions of mean and standard-deviation (Watson, 1983), which yield $\psi_0 = 0.66 \pm 2.60°$. Hence, the estimate for ψ_0 points towards the same order of magnitude as $\psi_0 = 6.69°$ shown in Fig. 2.9. Furthermore, for χ_0 we see a peak around 80° which can be interpreted to correspond to the previous best fit value $\chi_0 = 78.6°$. The second peak around $\chi_0 = 260°$ is less clear and is likely to be a morphological degeneracy. As synchrotron emission is sensitive to the magnetic field's direction but not to its orientation, considering Eq. (2.4), we expect a diffuse degeneracy in χ_0, which gets disturbed by the factor $\tanh(z/z_0)$.
Repeating this analysis for Faraday rotation data from Oppermann et al. (2012), results shown in Fig. 2.14, underlines what has been seen in Fig. 2.10. The WMAP LSA model is inherently incompatible to Faraday rotation observations: B_0 is pushed to values near zero and an increasing τ solely broadens B_0's likelihood as discussed in Sec. 2.4.6 but does not add anything to the intrinsic quality of fit. The likelihoods for ψ_0, ψ_1 and χ_0 don't posses any clear peaks nor pairwise correlations.
All in all, this simple example of the WMAP LSA model augmented with a random magnetic field already illustrates how challenging it is to create models of the constituents of the Galaxy with consistent geometry and to find reliable estimates for their parameters. While the example in this section was rather academic due to the model's simplicity, the presented steps similarly apply when analyzing more complex models such as JF12 and Jaffe13.

2.6 Conclusion & Outlook

In this paper we presented IMAGINE, a framework for GMF model parameter inference. We have discussed the motivation behind Bayesian parameter inference and model comparison as well as the importance of the *Galactic variance*. We then described the modular structure and extensibility of the IMAGINE framework. Its most important building blocks are:

- state-of-the-art parametric GMF models,
- a varied set of complementary observables,
- the new and improved HAMMURABI X simulator, and
- the different sampling algorithms that can be used within IMAGINE.

In Sec. 2.5, we showed with mock data that the pipeline works self-consistently, we illustrated the concept of Bayesian model comparison, and we then applied the pipeline to real data. In the course of this, we showed the importance of multi-observable based parameter fitting. This analysis was, however, a simple proof-of-concept to demonstrate the capabilities of the IMAGINE pipeline. Now, more sophisticated analyses are in order to gain as much scientific insight from existing data sets and GMF models as possible. Since IMAGINE is uniquely suited to handle the random component of the GMF and its uncertainties correctly, it can be adapted into a powerful tool to study the turbulent ISM by, e.g., adding a structure function analysis to the likelihood

in order to constrain the turbulent spectral index. All those insights should be used to build improved models and to keep the models' best-fit parameter estimates up-to-date with respect to the ever improving data. Within this paper we solely inferred the parameters of the GMF while keeping the thermal electron density fixed. With an extended list of observables (e.g., the DM), more informative datasets, and better models, we can extend this work to a joint inference of the magnetic field, the thermal electron density, the cosmic ray population, and even the dust model parameters. The IMAGINE pipeline is ready to help tackle this challenge and is available at: https://gitlab.mpcdf.mpg.de/ift/IMAGINE

Acknowledgements

We thank François Boulanger, Martin Reinecke, Luiz F. S. Rodrigues, and Anvar Shukurov for fruitful discussions and valuable suggestions. Part of this work was supported by the *Studienstiftung des deutschen Volkes*. The original concept of IMAGINE arose from two International Team meetings[4] hosted by the International Space Science Institute in Bern. We also acknowledge support and hospitality of the Lorentz Center in Leiden, where the IMAGINE project was further discussed and refined.[5] We acknowledge the support by the DFG Cluster of Excellence "Origin and Structure of the Universe". The computations have been carried out on the computing facilities of the Computational Center for Particle and Astrophysics (C2PAP) and the Radboud University, Nijmegen, respectively. This research has been partly supported by the DFG Research Unit 1254 and has made use of the NASA/IPAC Infrared Science Archive, which is operated by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Some of the results in this paper have been derived using the HEALPIX package (Górski et al., 2005). The corner plots where made using the `corner` PYTHON package (Foreman-Mackey, 2016).

[4] http://www.issibern.ch/teams/bayesianmodel/

[5] http://www.lorentzcenter.nl/lc/web/2017/880/info.php3?wsid=880

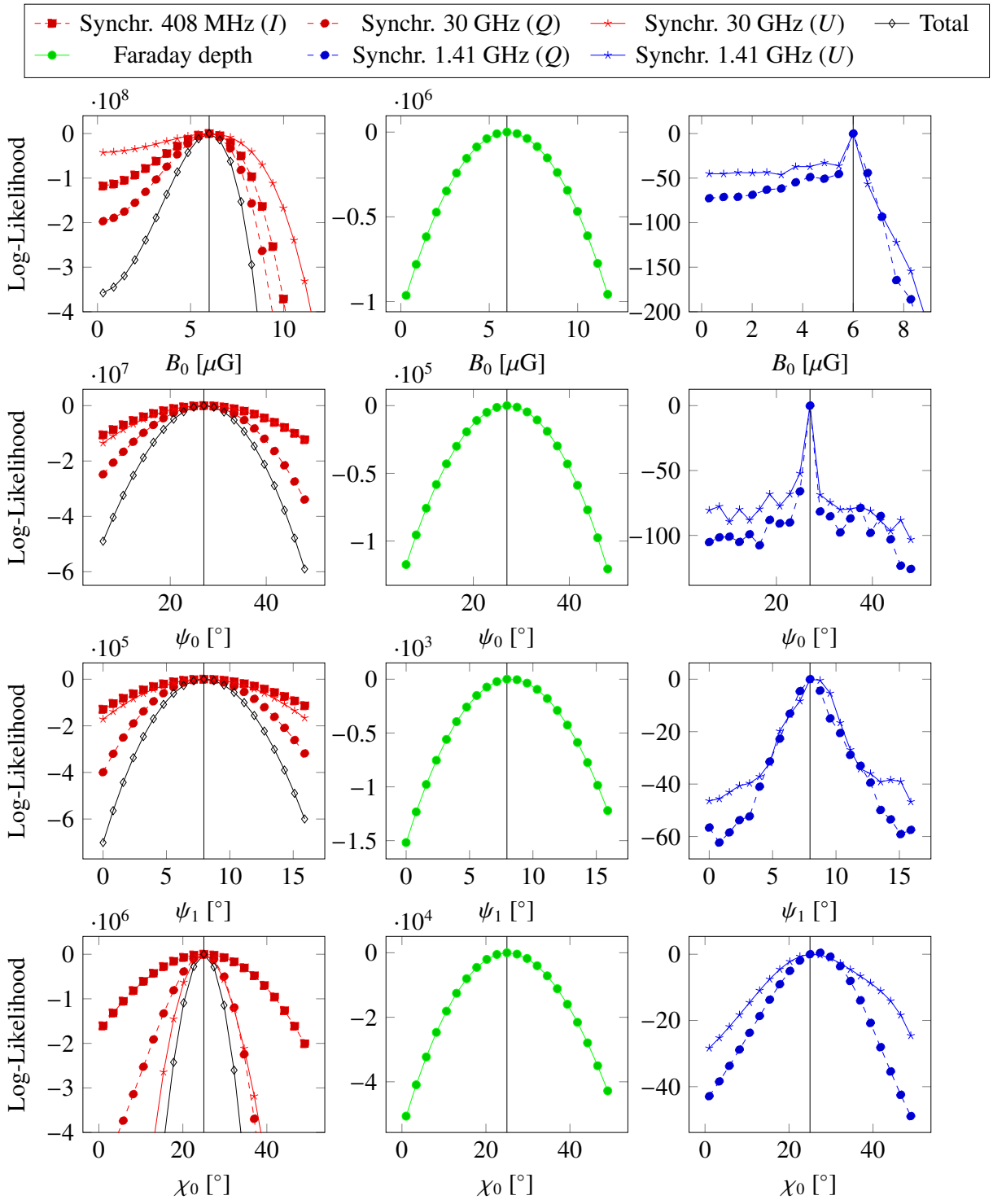


Figure 2.4: **Mock data ω_1:** Scans through the parameter space of the WMAP LSA regular magnetic field model, including simulated additive measurement noise according to Oppermann et al. (2012), Planck Collaboration et al. (2016a), Wolleben et al. (2006). The mock data input parameter values are at the very center of each abscissa and indicated by the vertical line. Since the Planck synchrotron data dominates the overall likelihood, we show the total likelihood in the leftmost plot. Note that the log-likelihood varies over several orders of magnitude.

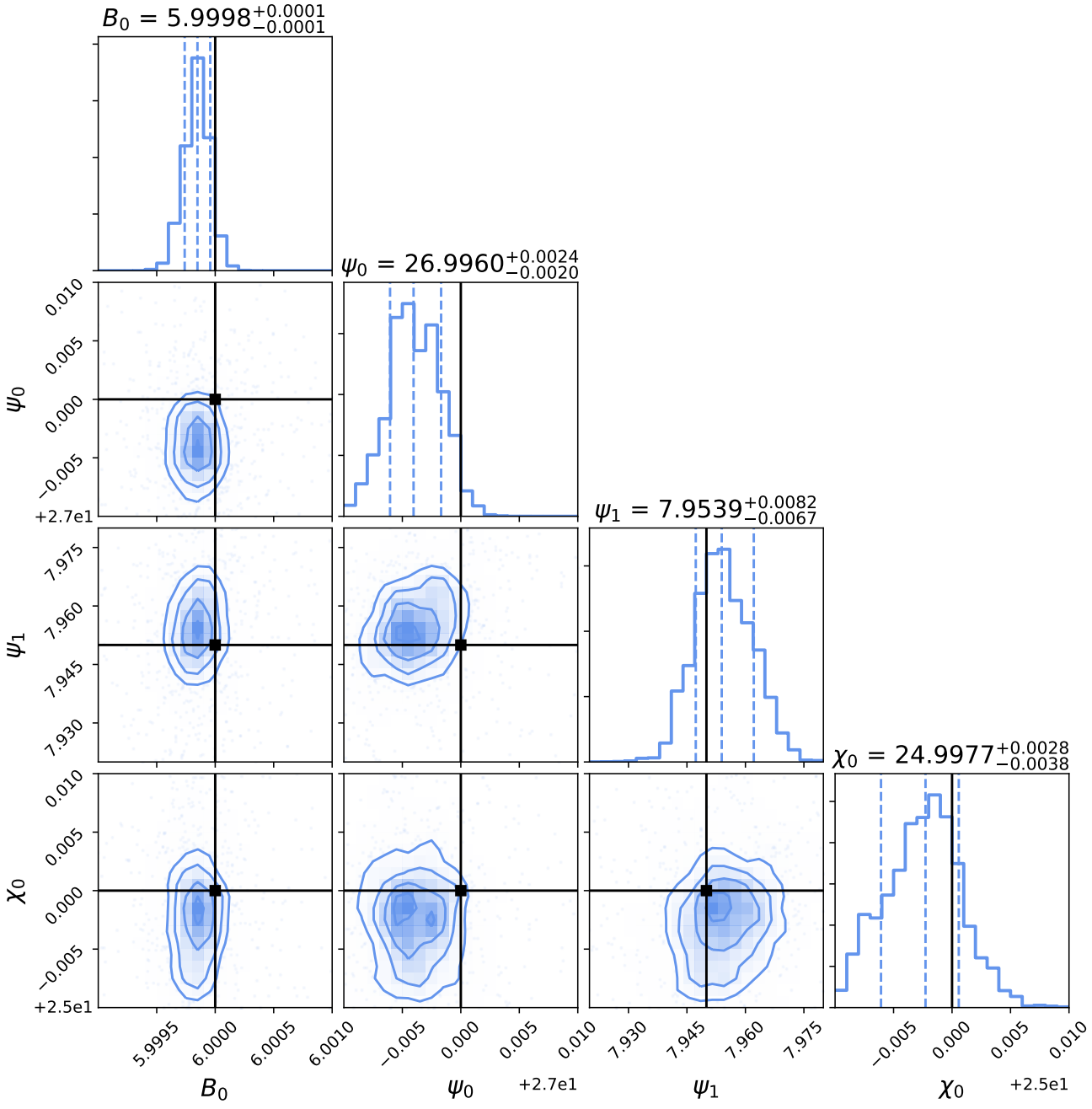
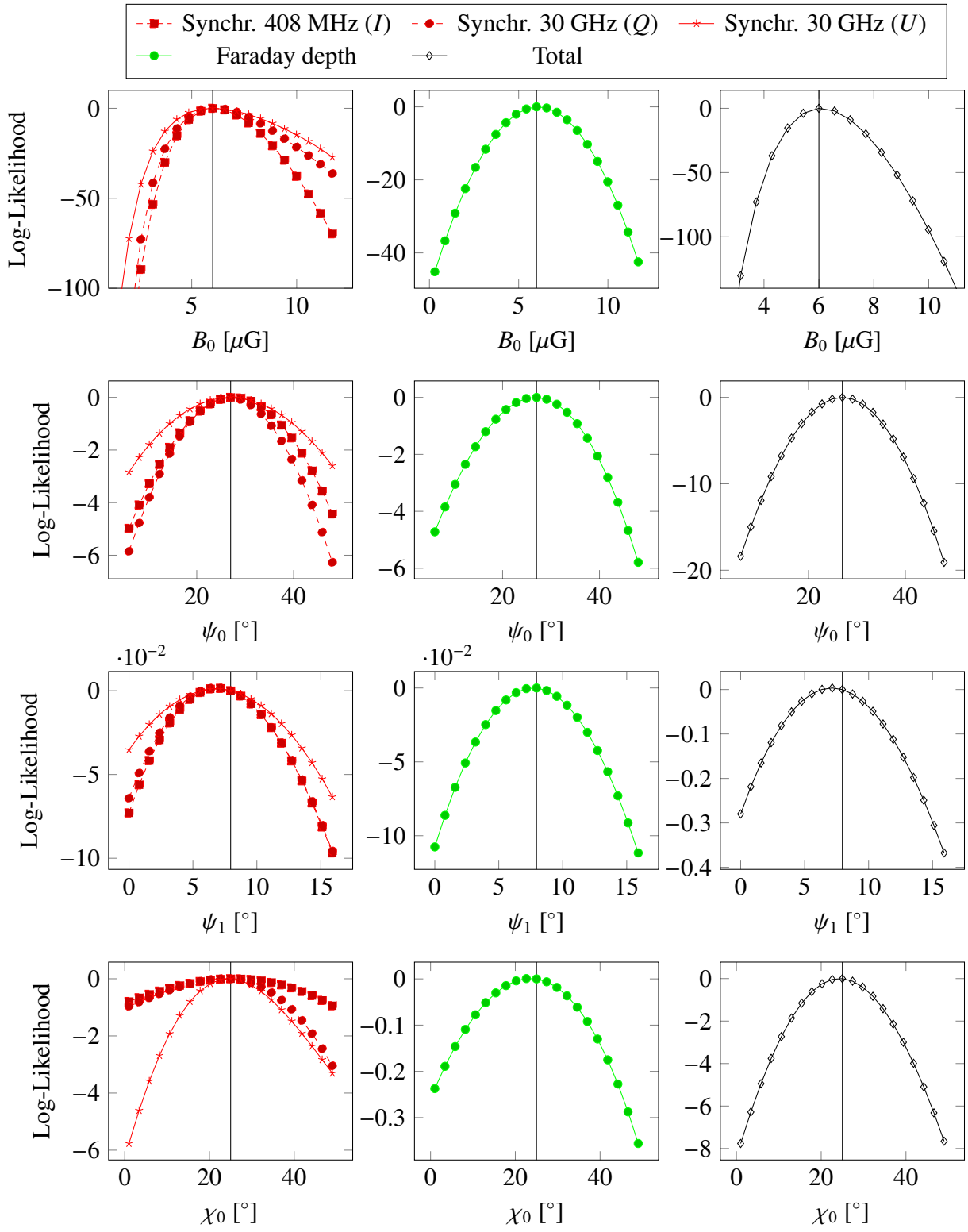


Figure 2.5: **Mock data ω_1:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to mock data ω_1. The dashed lines represent the 16%, 50% and 84% quantiles, respectively. The parameters for the mock data, indicated by the solid lines, were set to $B_0 = 6.0\,\mu$G, $\psi_0 = 27.0°$, $\psi_1 = 7.95°$, and $\chi_0 = 25°$.


Synchr. 408 MHz (I)
Synchr. 30 GHz (Q)
Synchr. 30 GHz (U)
Faraday depth
Total
Log-Likelihood
B_0 [μG]
ψ_0 [°]
ψ_1 [°]
χ_0 [°]
$\cdot 10^{-2}$

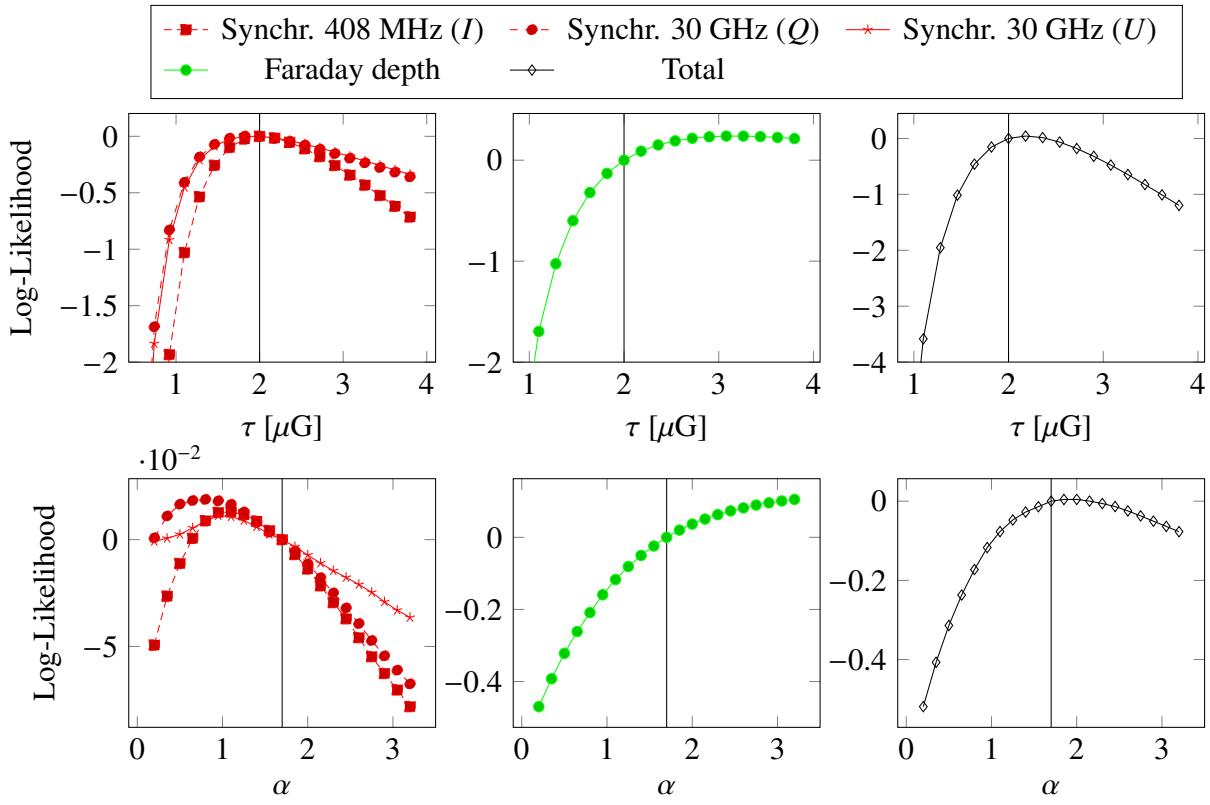


Figure 2.5: **Mock data ω_2:** Scans through the parameter space of the WMAP LSA regular plus isotropic random magnetic field model, including simulated additive measurement noise according to Oppermann et al. (2012), Planck Collaboration et al. (2016a). The mock data input parameter values are indicated by the vertical line. Note that the plot of the *total* likelihood does not include the synchrotron data at 1.41 GHz.

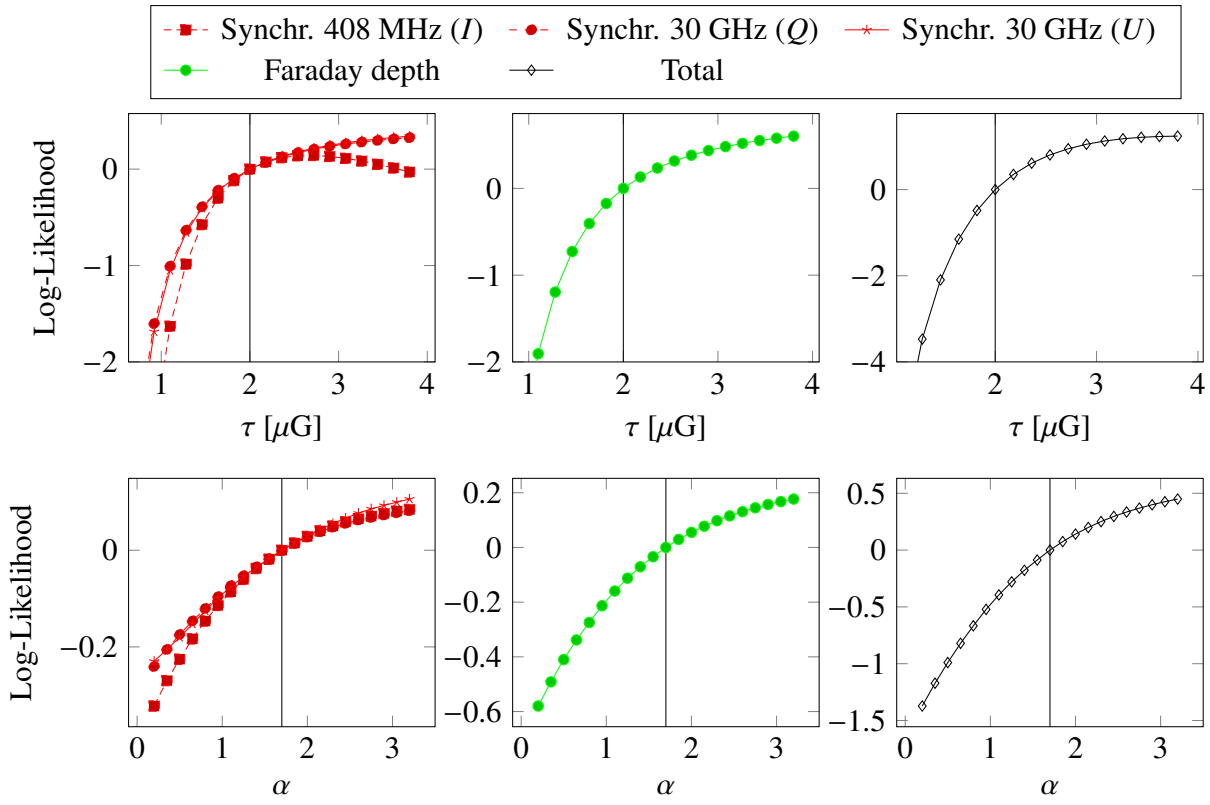


Figure 2.6: **Mock data ω_2, without determinant term:** Scans through the parameter space of the WMAP LSA regular plus isotropic random magnetic field model, including simulated additive measurement noise according to Oppermann et al. (2012), Planck Collaboration et al. (2016a), Wolleben et al. (2006). For these plots the ensemble likelihood was evaluated without the determinant term. The mock data input parameter values are indicated by the vertical line. Note that the plot of the *total* likelihood does not include the synchrotron data at 1.41 GHz.

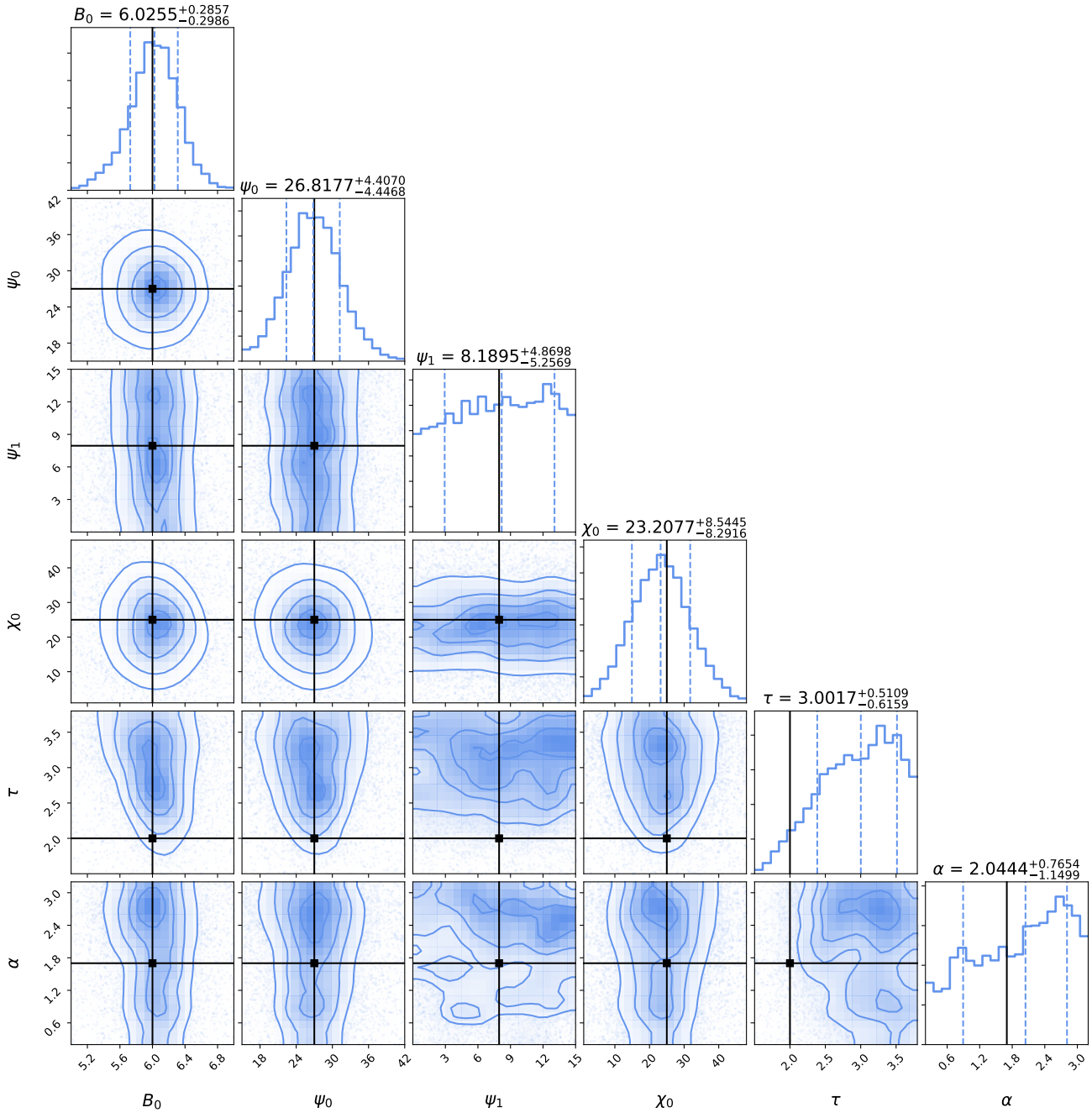


Figure 2.7: **Mock data ω_2:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to mock data ω_2. The dashed lines represent the 16%, 50% and 84% quantiles, respectively. The parameters for the mock data, indicated by the solid lines, were set to $B_0 = 6.0\,\mu$G, $\psi_0 = 27.0°$, $\psi_1 = 7.95°$, $\chi_0 = 25°$, $\tau = 2\,\mu$G, and $\alpha = 1.7$.

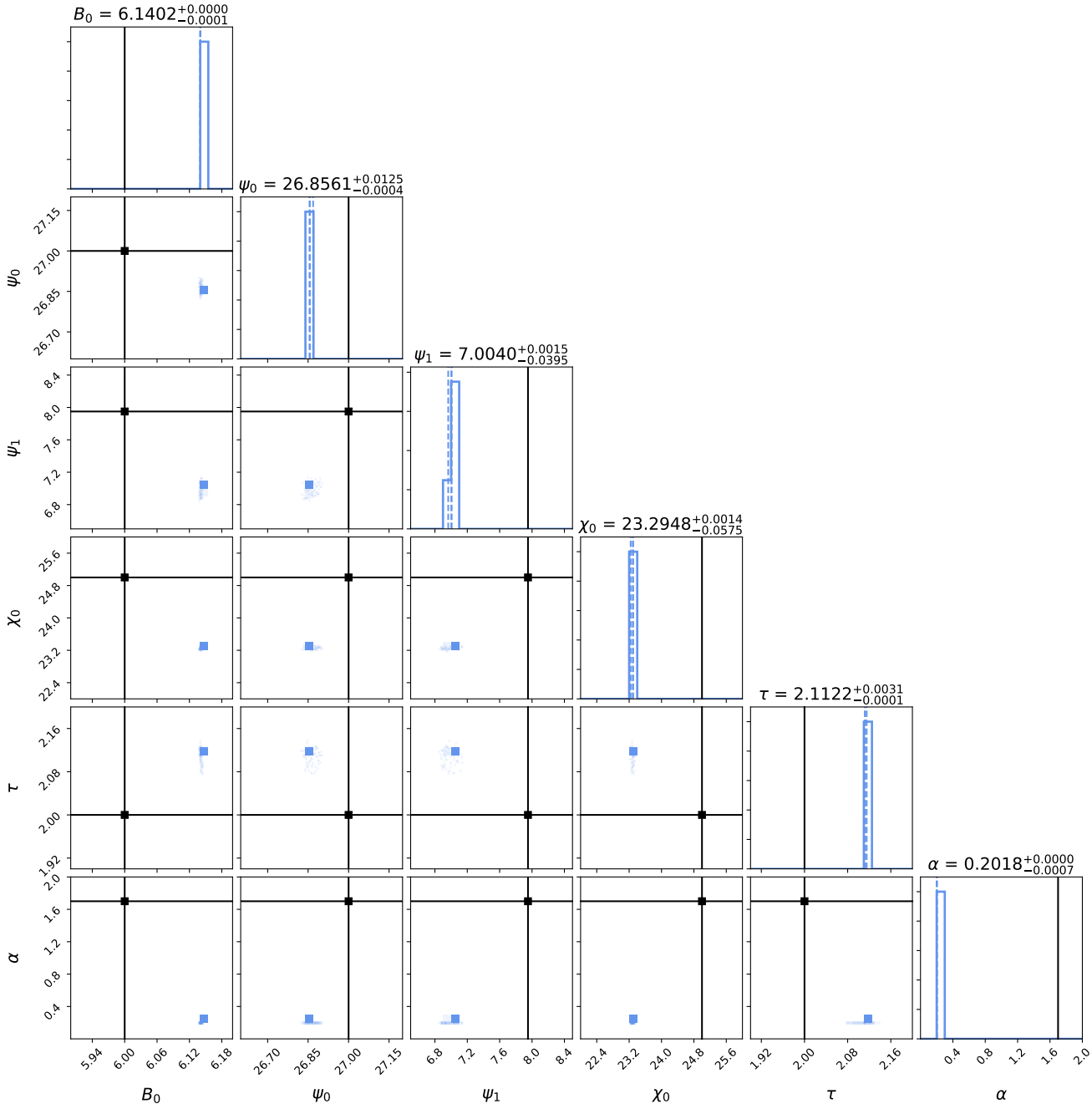


Figure 2.8: **Mock data ω_2 in combination with a χ^2 likelihood:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to mock data ω_2 using a simple χ^2 likelihood that not reflects the influence of the Galactic variance. The dashed lines represent the 16%, 50% and 84% quantiles, respectively. The parameters for the mock data, indicated by the solid lines, were set to $B_0 = 6.0\ \mu$G, $\psi_0 = 27.0°$, $\psi_1 = 7.95°$, $\chi_0 = 25°$, $\tau = 2\ \mu$G, and $\alpha = 1.7$.

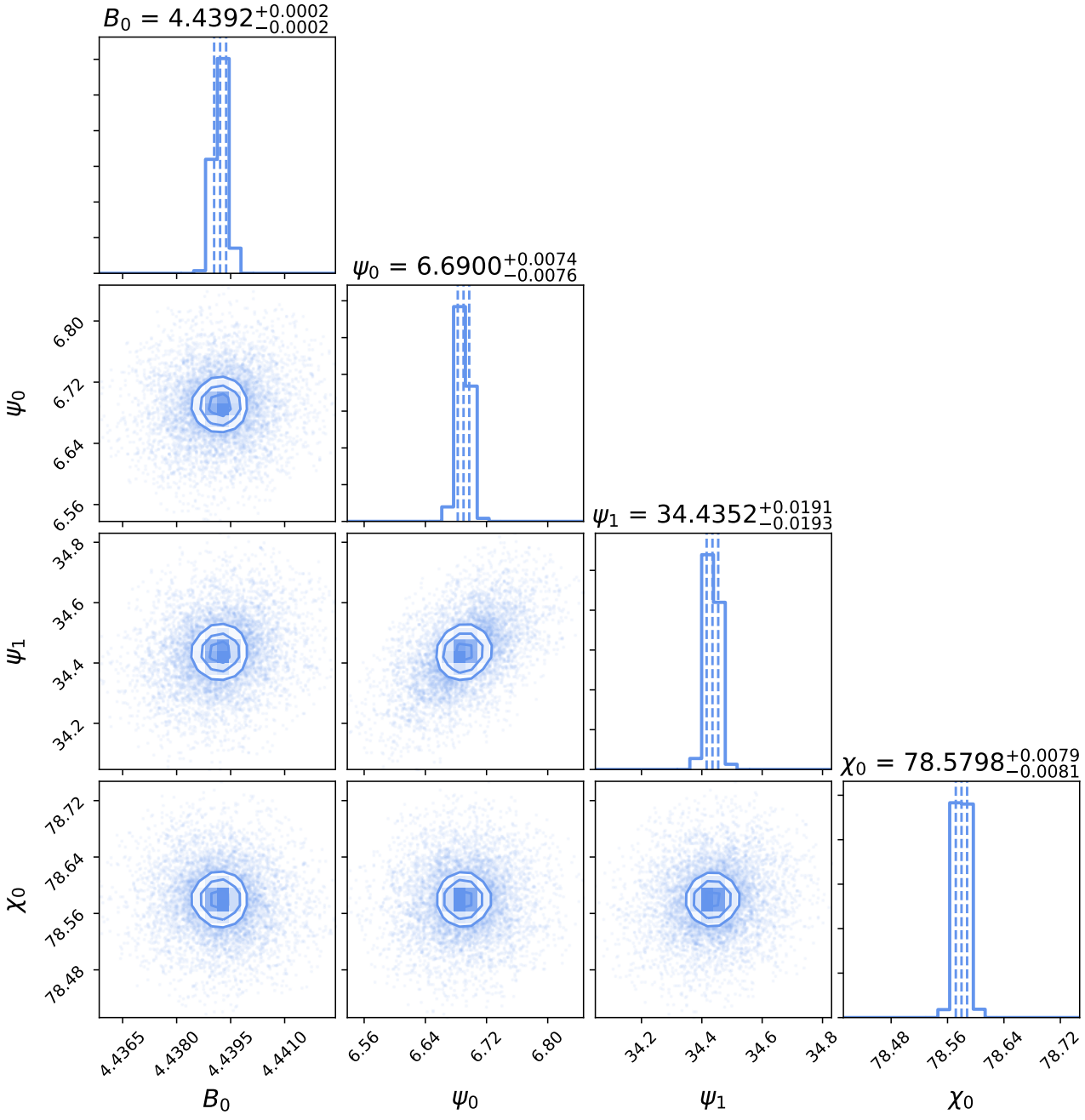


Figure 2.9: **Synchrotron data, purely ordered WMAP LSA magnetic field:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to 408 MHz and 30 GHz synchrotron data from Planck Collaboration et al. (2016a). The dashed lines represent the 16%, 50% and 84% quantiles, respectively.

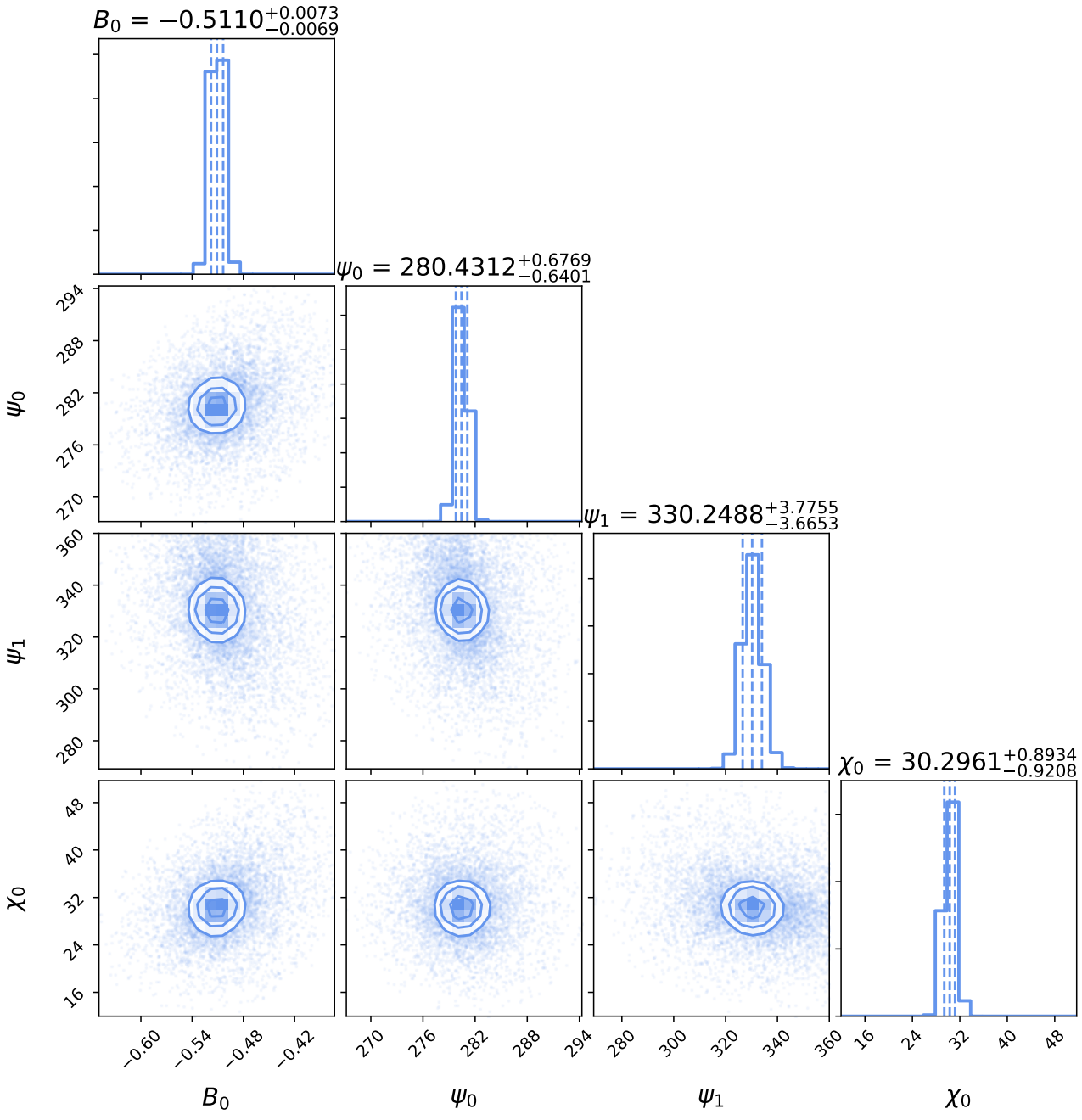
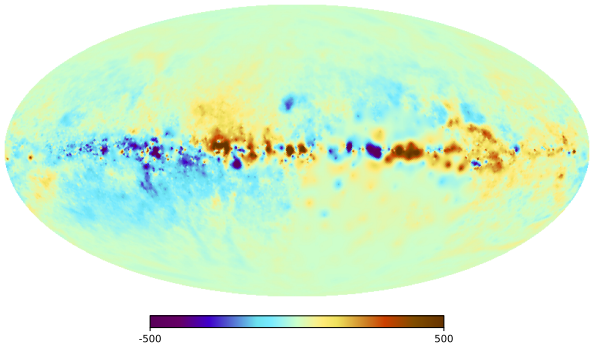
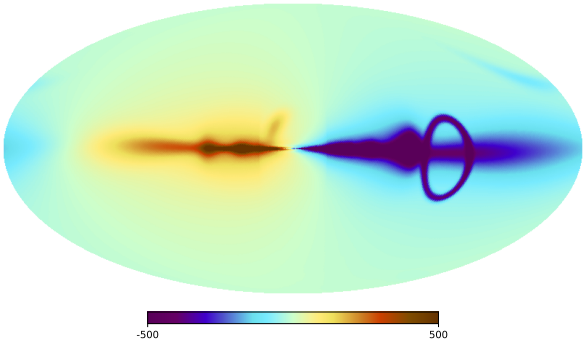


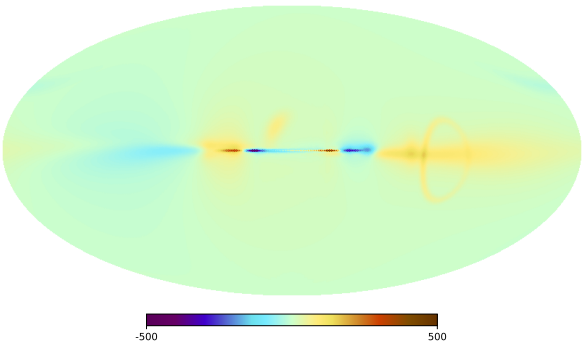
Figure 2.10: **Faraday rotation data, purely ordered WMAP LSA magnetic field:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to Faraday rotation data from Oppermann et al. (2012). The dashed lines represent the 16%, 50% and 84% quantiles, respectively.



(a) Map of the Galactic Faraday depth given by Oppermann et al. (2012) in rad/m^2.



(b) Map of the Galactic Faraday depth produced with HAMMURABI X based on the WMAP LSA model in rad/m^2. The model's parameters were set to $B_0 = 1.5\ \mu$G, $\psi_0 = 27.0°$, $\psi_1 = 0.9°$, and $\chi_0 = 25.0°$, following Page et al. (2007).



(c) Map of the Galactic Faraday depth produced with HAMMURABI X based on the WMAP LSA model in rad/m^2. The model's parameters were set to $B_0 = -0.51\ \mu$G, $\psi_0 = 280°$, $\psi_1 = 330°$, and $\chi_0 = 30.3°$.

Figure 2.11: Comparison of Faraday depth maps in rad/m^2.

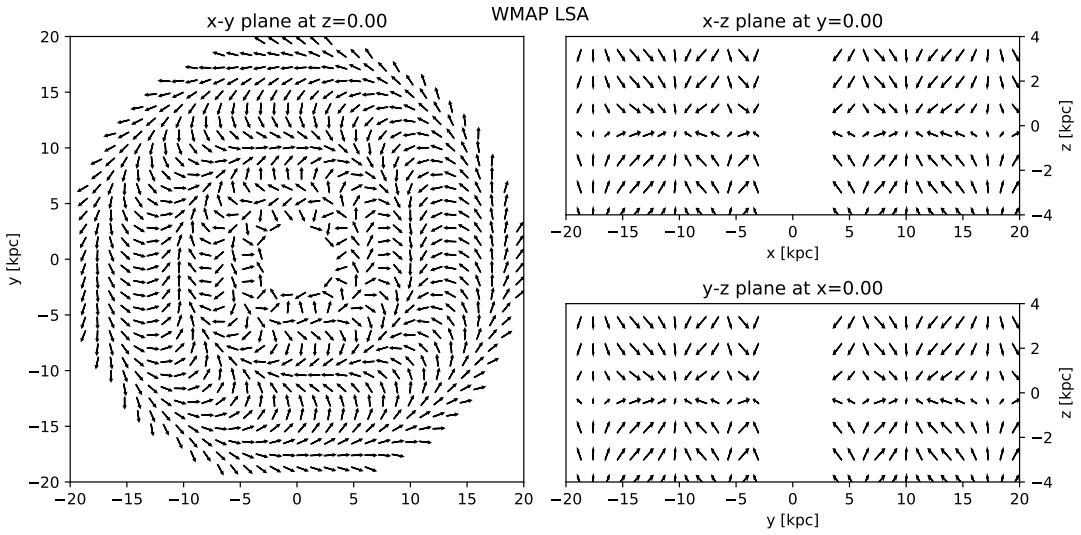


Figure 2.12: Streamplot of the WMAP LSA model for $B_0 = -0.51\ \mu$G, $\psi_0 = 280°$, $\psi_1 = 330°$, and $\chi_0 = 30.3°$.

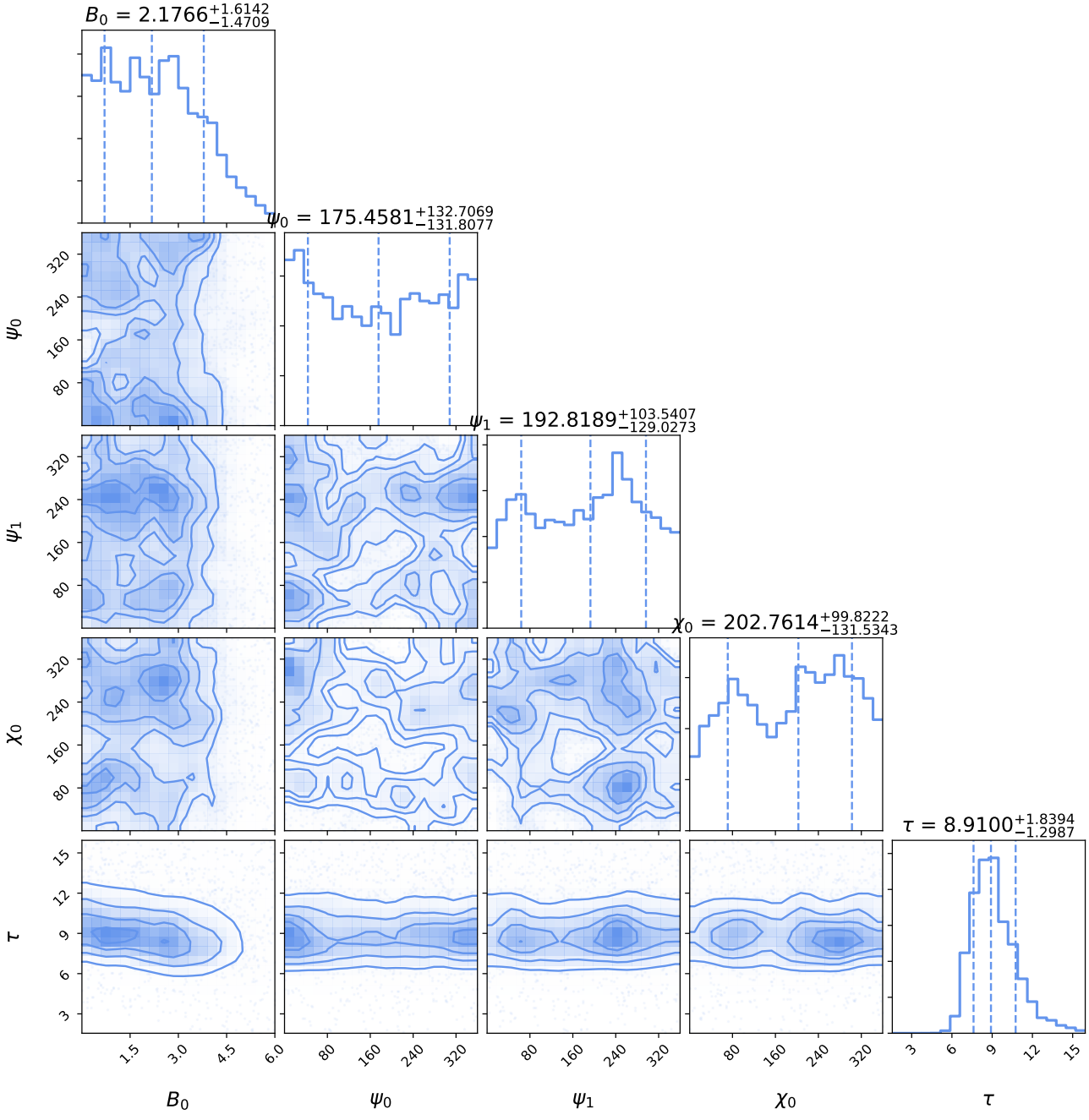


Figure 2.13: **Synchrotron radiation data, WMAP LSA plus random magnetic field:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to 408 MHz and 30 GHz synchrotron data from Planck Collaboration et al. (2016a). The dashed lines represent the 16%, 50% and 84% quantiles, respectively.

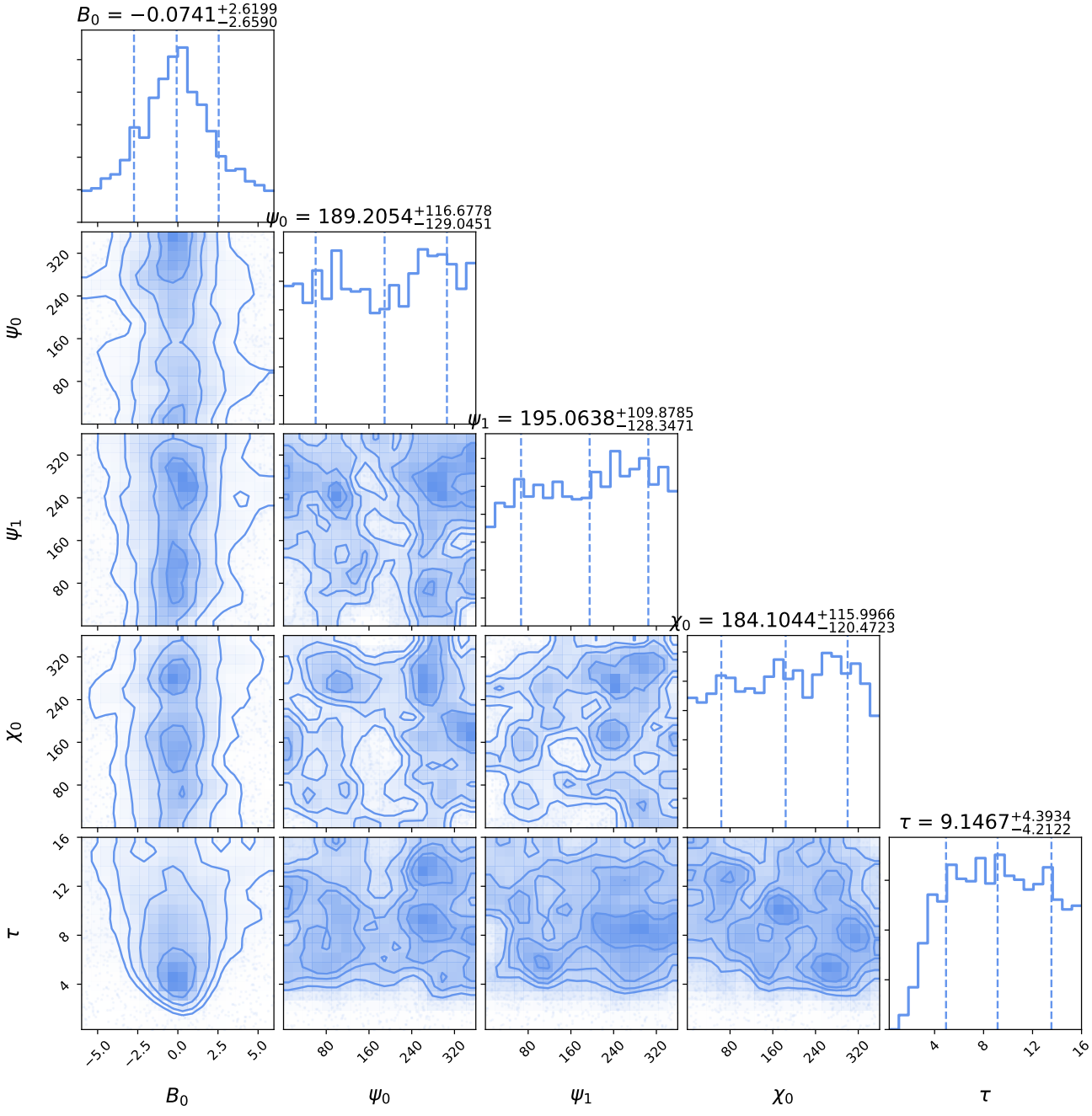


Figure 2.14: **Faraday rotation data, WMAP LSA plus random magnetic field:** Marginalized posterior plots and projected pairwise correlation plots from applying PyMultiNest to Faraday rotation data from Oppermann et al. (2012). The dashed lines represent the 16%, 50% and 84% quantiles, respectively.

Chapter 3

D2O - a distributed data object for parallel high-performance computing in Python

This chapter, as well as Appendix A, are additionally used as a journal publication in the SpringerOpen Journal of Big Data *(Steininger et al., 2016).*
I am the principal researcher of the research described in this chapter. My contributions include the primal idea, the implementation of presented software package, and the conduction of the performance tests. Furthermore, I wrote this chapter. Maksim Greiner (MG), Frederik Beaujean (FB) and Torsten Enßlin (TE) helped working out the conceptual structure of the software package, and drafting the chapter by doing language editing. FB also played a pivotal role for executing the performance tests. TE also fulfilled the role of a principal investigator as he is my PhD supervisor. All authors read, commented, and approved the final manuscript.

Abstract

We introduce D2O, a Python module for cluster-distributed multi-dimensional numerical arrays. It acts as a layer of abstraction between the algorithm code and the data-distribution logic. The main goal is to achieve usability without losing numerical performance and scalability. D2O's global interface is similar to the one of a `numpy.ndarray`, whereas the cluster node's local data is directly accessible for use in customized high-performance modules. D2O is written in pure Python which makes it portable and easy to use and modify. Expensive operations are carried out by dedicated external libraries like *numpy* and *mpi4py*. The performance of D2O is on a par with numpy for serial applications and scales well when moving to an MPI cluster. D2O is open-source software available under the GNU General Public License v3 (GPL-3) at `https://gitlab.mpcdf.mpg.de/ift/D2O`.

3.1 Introduction

3.1.1 Background

Data sets in simulation and signal-reconstruction applications easily reach sizes too large for a single computer's random access memory (RAM). A reasonable grid size for such tasks like galactic density reconstructions (Greiner et al., 2016) or multi-frequency imaging in radio astronomy (Junklewitz et al., 2016) is a cube with a side resolution of 2048. Such a cube contains $2048^3 \approx 8.6 \cdot 10^9$ voxels. Storing a 64-bit double for every voxel therefore consumes 64 GiB. In practice one has to handle several or even many instances of those arrays which ultimately prohibits the use of single shared memory machines. Apart from merely holding the arrays' data in memory, parallelization is needed to process those huge arrays within reasonable time. This applies to basic arithmetics like addition and multiplication as well as to complex operations like Fourier transformation and advanced linear algebra, e.g. operator inversions or singular value decompositions. Thus parallelization is highly advisable for code projects that must be scaled to high resolutions.
To be specific, the initial purpose of D2O was to provide parallelization to the package for Numerical Information Field Theory (NIFTY)(Selig et al., 2013), which permits the abstract and efficient implementation of sophisticated signal processing methods. Typically, those methods are so complex on their own that a NIFTY user should not need to bother with parallelization details in addition to that. It turned out that providing a generic encapsulation for parallelization to NIFTY is not straightforward as the applications NIFTY is used for are highly diversified. The challenge hereby is that, despite all their peculiarities, for those applications numerical efficiency is absolutely crucial. Hence, for encapsulating the parallelization effort in NIFTY we needed an approach that is flexible enough to adapt to those different applications such that numerical efficiency can be preserved: D2O.
D2O is implemented in Python. As a high-level language with a low-entry barrier Python is widely used in computational science. It has a huge standard library and an active community for 3rd party packages. For computationally demanding applications Python is predominantly used as a steering language for external compiled modules because Python itself is slow for numerics.
This article is structured as follows. Sec. 3.1.2 gives the aims of D2O, and Sec. 3.1.3 describes alternative data distribution packages. We dicuss the code architecture in Sec. 3.2 , the basic usage of D2O in Sec. 3.3, and the numerical scaling behavior in Sec. 3.4. Sec. 3.5 contains our conclusion and Sec. A.1 describes the detailed usage of D2O.

3.1.2 Aim

As most scientists are not fully skilled software engineers, for them the hurdle for developing parallelized code is high. Our goal is to provide data scientists with a numpy array-like object (cf. *numpy* (van der Walt et al., 2011)) that distributes data among several nodes of a cluster in a user-controllable way. The user, however, shall not need to have profound knowledge about parallel programming with a system like *MPI* (Message Passing Interface Forum, 1994, 1998) to achieve this. The transition to use *distributed_data_objects* instead of numpy arrays in exist-

ing code must be as straightforward as possible. Hence, D2O shall in principle run – at least in a non-parallelized manner – with standard-library dependencies available; the packages needed for parallel usage should be easily available. Whilst providing a global-minded interface, the node's local data should be directly accessible in order to enable the usage in specialized high-performance modules. This approach matches with the theme of *DistArray* (Enthought, 2016): "Think globally, act locally". Regarding D2O's architecture we do not want to make any a-priori assumptions about the specific distribution strategy, but retain flexibility: it shall be possible to adapt to specific constraints induced from third-party libraries a user may incorporate. For example, a library for fast Fourier transformations like *FFTW* (Frigo, 1999) may rely on a different data-distribution model than a package for linear algebra operations like *ScaLAPACK* (Blackford et al., 1997)[1]. In the same manner it shall not matter whether a new distribution scheme stores data redundantly or not, e.g. when a node is storing not only a distinct piece of a global array, but also its neighboring (ghost) cells (Dadone and Grossman, 2004).
Our main focus is on rendering extremely costly computations possible in the first place; not on improving the speed of simple computations that can be done serially. Although primarily geared towards *weak scaling*, it turns out that D2O performs very well in *strong-scaling* scenarios, too; see Sec. 3.4 for details.

3.1.3 Alternative Packages

There are several alternatives to D2O. We discuss the differences to D2O and why the alternatives are not sufficient for our needs.

3.1.3.1 DistArray

DistArray (Enthought, 2016) is very mature and powerful. Its approach is very similar to D2O: It mimics the interface of a multi dimensional numpy array while distributing the data among nodes in a cluster. However, DistArray involves a design decision that makes it inapt for our purposes: it has a strict client-worker architecture. DistArray either needs an *ipython ipcluster* (Pérez and Granger, 2007) as back end or must be run with two or more MPI processes. The former must be started before an interactive ipython session is launched. This at least complicates the workflow in the prototyping phase and at most is not practical for batch system based computing on a cluster. The latter enforces tool-developers who build on top of *DistArray* to demand that their code always is run parallelized. Both scenarios conflict with our goal of minimal second order dependencies and maximal flexibility, cf. Sec. 3.1.2. Nevertheless, its theme also applies to D2O: "Think globally, act locally".

3.1.3.2 scalapy (ScaLAPACK)

scalapy is a Python wrapper around *ScaLAPACK* (Blackford et al., 1997), which is "a library of high-performance linear algebra routines for parallel distributed memory machines" (Team, 2016b). The `scalapy.DistributedMatrix` class essentially uses the routines from ScaLA-

[1] FFTW distributes slices of data, while ScaLAPACK uses a block-cyclic distribution pattern.

PACK and therefore is limited to the functionality of that: two-dimensional arrays and very specific block-cyclic distribution strategies that optimize numerical efficiency in the context of linear algebra problems. In contrast, we are interested in *n*-dimensional arrays whose distribution scheme shall be arbitrary in the first place. Therefore scalapy is not extensive enough for us.

3.1.3.3 petsc4py (PETSc)

petsc4py is a Python wrapper around *PETSc*, which "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations" (Balay et al., 2015). Regarding distributed arrays its scope is as focused as scalapy to its certain problem domain – here: solving partial differential equations. The class for distributed arrays `petsc4py.PETSc.DMDA` is limited to one, two and three dimensions as PETSc uses a highly problem-fitted distribution scheme. We in contrast need *n*-dimensional arrays with arbitrary distribution schemes. Hence, petsc4py is not suitable for us.

3.2 Code Architecture

3.2.1 Choosing the Right Level of Parallelization

D2O distributes numerical arrays over a cluster in order to parallelize and therefore to speed up operations on the arrays themselves. An application that is built on top of D2O can profit from its fast array operations that may be performed on a cluster. However, there are various approaches how to deploy an algorithm on a cluster and D2O implements only one of them. In order to understand the design decisions of D2O and its position respective to other packages, cf. Sec. 3.1.3, we will now discuss the general problem setting of parallelization and possible approaches for that. Thereby we reenact the decision process which led to the characteristics D2O has today.

3.2.1.1 Vertical & Horizontal Scaling

Suppose we want to solve an expensive numerical problem which involves operations on data arrays. To reduce the computation time one can in principle do two things. Either use a faster machine – *vertical scaling* – or use more than one machine – *horizontal scaling*. Vertical scaling has the advantage that existing code does not need to be changed[2], but in many cases this is not appropriate. Maybe one already uses the fastest possible machine, scaling up is not affordable or even the fastest machine available is still too slow. Because of this, we choose horizontal scaling.

3.2.1.2 High- & Low-Level Parallelization

With horizontal scaling we again face two choices: high- and low-level parallelization. With high-level parallelization, many replicas of the algorithm run simultaneously, potentially on mul-

[2]This is true if scaling up does not involve a change of the processor architecture.

tiple machines. Each instance then works independently if possible, solving an isolated part of the global problem. At the end, the individual results get collected and merged. The python framework *pathos* (McKerns et al., 2012) provides functionality for this kind of procedure.
An example of high-level parallelization is a sample generator which draws from a probability distribution. Using high-level parallelization many instances of the generator produce their own samples, which involves very little communication overhead. The sample production process itself, however, is not sped up.
In low-level parallelization, several nodes work together on one basic task at a time. For the above sample generator, this means that all nodes work on the same sample at a time. Hence, the time needed for producing individual samples is reduced; they are serially generated by the cluster as a whole.

3.2.1.3 Downsides

Both of these approaches have their drawbacks. For high-level parallelization the algorithm itself must be parallelizable. Every finite algorithm has a maximum degree of intrinsic parallelization[3]. If this degree is lower than the desired number of processes then high-level parallelization reaches its limits. This is particularly true for algorithms that cannot be parallelized by themselves, like iterative schemes. Furthermore, there can be an additional complication: if the numerical problem deals with extremely large objects it may be the case that it is not at all solvable by one machine alone[4].
Now let us consider low-level parallelization. As stated above, we assume that the solution of the given numerical problem involves operations on data arrays. Examples for those are unary[5], binary[6] or sorting operations, but also more advanced procedures like Fourier transformations or (other) linear algebra operations. Theoretically, the absolute maximum degree of intrinsic parallelization for an array operation is equal to the array's number of elements. For comparison, the problems we want to tackle involve at least 10^8 elements but most of the TOP500 (Strohmaier et al., 2015) supercomputers possess 10^6 cores or less. At first glance this seems promising. But with an increasing number of nodes that participate in one operation the computational efficiency may decrease considerably. This happens if the cost of the actual numerical operations becomes comparable to the generic program and inter-node communication overhead. The ratios highly depend on the specific cluster hardware and the array operations performed.

3.2.1.4 Problem Sizes

Due to our background in signal reconstruction and grid-based simulations, we decide to use low-level parallelization for the following reasons. First, we have to speed up problems that one

[3]For the exemplary sample generator the maximum degree of parallelization is the total number of requested samples.

[4]In case of the sample generator this would be the case if even one sample would be too large for an individual machine's RAM.

[5]E.g. the positive, negative or absolute values of the array's individual elements or the maximum, minimum, median or mean of all its elements.

[6]E.g. the sum, difference or product of two data arrays.

cannot parallelize algorithmically, like fixed-point iterations or step-wise simulations. Second, we want to scale our algorithms to higher resolutions while keeping the computing time at least constant. Thereby the involved data arrays become so big that a single computer would be oversubscribed. Because of this, the ratio of *array size* to *desired degree of parallelization* does not become such that the computational efficiency would decrease considerably. In practice we experience a good scaling behavior with up to $\approx 10^3$ processes[7] for problems of size 8192^2, cf. Sec. 3.4. Hence, for our applications the advantages of low-level parallelization clearly outweigh its drawbacks.

3.2.2 D2O as Layer of Abstraction

Compared to high-level parallelization, the low-level approach is more complicated to implement. In the best case, for the former one simply runs the serial code in parallel on the individual machines; when finished one collects and combines the results. For the latter, when doing the explicit coding one deals with local data portions of the global data array on the individual nodes of the cluster. Hence, one has to keep track of additional information: for example, given a distribution scheme, which portion of the global data is stored on which node of the cluster? Keeping the number of cluster nodes, the size and the dimensionality of the data arrays arbitrary implies a considerable complication for indexing purposes. By this, while implementing an application one has to take care of two non-trivial tasks. On the one hand, one must program the logic of distributing and collecting the data; i.e. the data handling. On the other hand, one must implement the application's actual (abstract) algorithm. Those two tasks are conceptually completely different and therefore a mixture of implementations should be avoided. Otherwise there is the risk that features of an initial implementation – like the data distribution scheme – become hardwired to the algorithm, inhibiting its further evolution. Thus it makes sense to insert a layer of abstraction between the algorithm code and the data distribution logic. Then the abstract algorithm can be written in a serial style from which all knowledge and methodology regarding the data distribution is encapsulated. This layer of abstraction is D2O.

3.2.3 Choosing a Parallelization Environment

To make the application spectrum of D2O as wide as possible we want to maximize its portability and reduce its dependencies. This implies that – despite its parallel architecture – D2O must just as well run within a single-process environment for cases when no elaborate parallelization back end is available. But nevertheless, D2O must be massively scalable. This relates to the question of which distribution environment should be used. There are several alternatives:

- Threading and multiprocessing: These two options limit the application to a single machine which conflicts with the aim of massive scalability.
- (py)Spark (Zaharia et al., 2010) and hadoop (Apache Software Foundation, 2016): These modern frameworks are very powerful but regrettably too abstract for our purposes, as they

[7] This was the maximum number of processes available for testing.

prescind the location of individual portions of the full data. Building a numpy-like interface would be disproportionately hard or even unfeasible. In addition to that, implementing a low-level interface for highly optimized applications which interact with the node's local data is not convenient within pySpark. Lastly, those frameworks are usually not installed as standard dependencies on scientific HPC clusters.

- MPI (Message Passing Interface Forum, 1994, 1998): The *Message Passing Interface* is available on virtually every HPC cluster via well-tested implementations like *Open-MPI* (Gabriel et al., 2004), *MPICH2* (Team, 2016a) or *Intel MPI* (Corporation, 2016). The open implementations are also available on commodity multicore hardware like desktops or laptops. A Python interface to MPI is given by the Python module *mpi4py* (Dalcín et al., 2005). MPI furthermore offers the right level of abstraction for hands-on control of distribution strategies for the package developers.

Given these features we decide to use *MPI* as the parallelization environment for D2O. We stress that in order to fully utilize D2O on multiple cores, a user does not need to know how to program in MPI; it is only necessary to execute the program via MPI as shown in the example in Sec. 3.3.5.

3.2.4 Internal Structure

3.2.4.1 Composed Object

A main goal for the design of D2O was to make no a-priori assumptions about the specific distribution strategies that will be used in order to spread array data across the nodes of a cluster. Because of this, D2O's distributed array – `d2o.distributed_data_object` – is a composed object; cf. Fig. 3.1.
The *distributed_data_object* itself provides a rich user interface, and makes sanity and consistency checks regarding the user input. In addition to that, the *distributed_data_object* possesses an attribute called `data`. Here the MPI processes' local portion of the global array data is stored, even though the *distributed_data_object* itself will never make any assumptions about its specific content since the distribution strategy is arbitrary in the first place. The *distributed_data_object* is the only object of the D2O library that a casual user would interact with.
Every *distributed_data_object* possesses an instance of a `d2o.distributor` subclass for all tasks that require knowledge about the certain distribution strategy. This object stores all the distribution-scheme and cluster related information it needs in order to scatter (gather) data to (from) the nodes and to serve for special methods, e.g. the array-cumulative sum. The *distributed_data_object* builds its rich user interface on top of those abstracted methods of its distributor.
The benefit of this strict separation is that the user interface becomes fully detached from the distribution strategy; may it be block-cyclic or slicing, or have neighbor ghost cells or not, et cetera. Currently there are two fundamental distributors available: a generic *slicing*-[8] and a *not-*

[8]The slicing is done along the first array axis.

distributor. From the former, three special slicing distributors are derived: *fftw*[9], *equal*[10] and *freeform*[11]. The latter, the *not*-distributor, does not do any data-distribution or -collection but stores the full data on every node redundantly.

3.2.4.2 Advantages of a Global View Interface

D2O's global view interface makes it possible to build software that remains completely independent from the distribution strategy and the used number of cluster processes. This in turn enables the development of 3rd party libraries that are very end-use-case independent. An example for this may be a mathematical optimizer; an object which tries to find for a given scalar function f an input vector $\vec{x}$ such that the output $y = f(\vec{x})$ becomes minimal. It is interesting to note that many optimization algorithms solely use basic arithmetics like vector addition or scalar multiplication when acting on $\vec{x}$. As such operations act locally on the elements of an array, there is no preference for one distribution scheme over another when distributing $\vec{x}$ among nodes in a cluster. Two different distribution schemes will yield the same performance if their load-balancing is on a par with each other. Further assume that f is built on D2O, too. On this basis, one could now build an application that uses the minimizer but indeed has a preference for a certain distribution scheme. This may be the case if the load-balancing of the used operations is non-trivial and therefore only a certain distribution scheme guarantees high evaluation speeds. While the application's developer therefore enforces this scheme, the minimizer remains completely unaffected by this as it is agnostic of the array's distribution strategy.

[9] The fftw-distributor uses routines from the pyFFTW (Frigo, 1999, Gomersall, 2016) package (Frigo, 1999) for the data partitioning.

[10] The equal-distributor tries to split the data in preferably equal-sized parts.

[11] The local data array's first axis is of arbitrary length for the freeform-distributor.

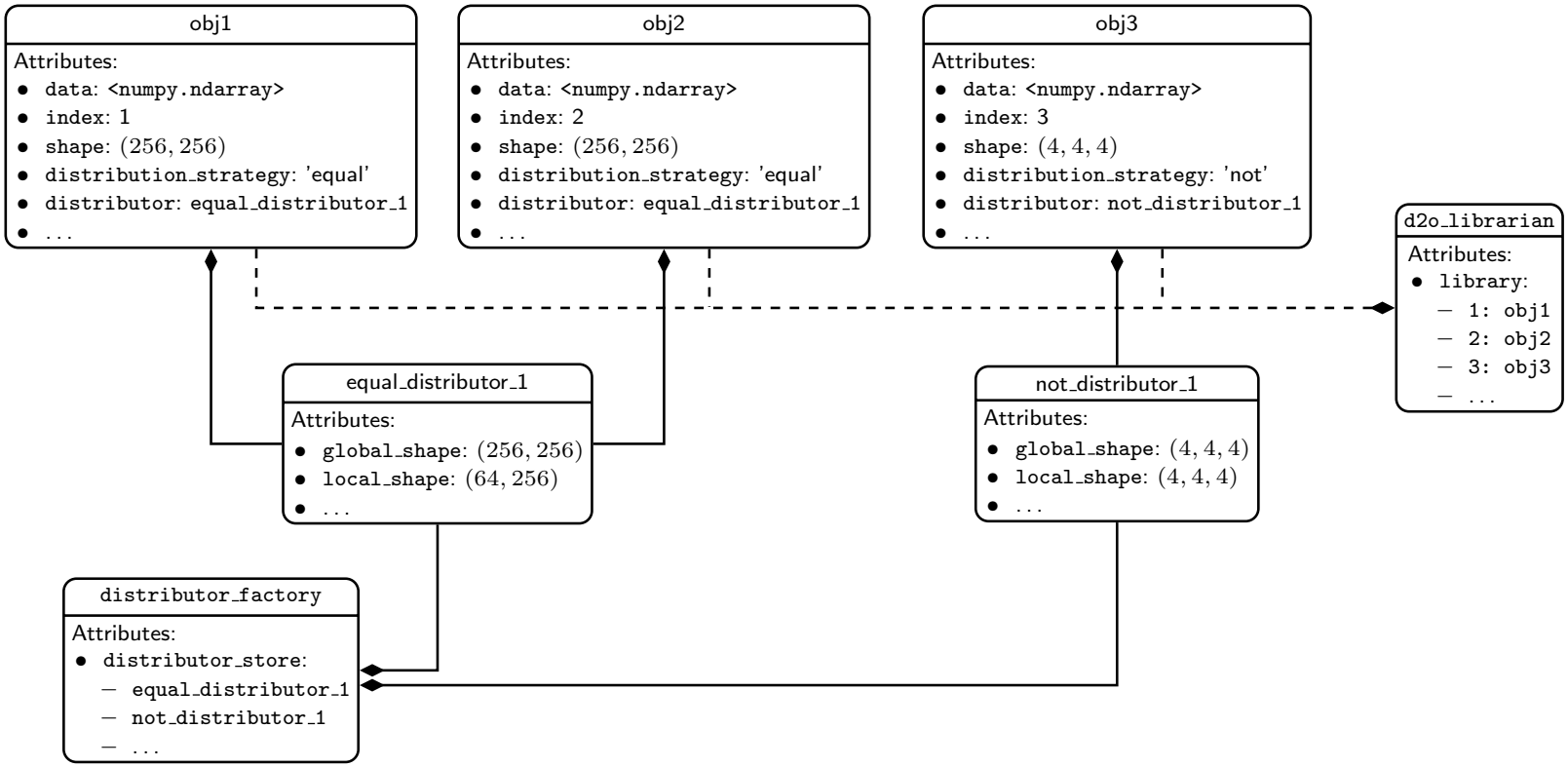


Figure 3.1: Here the main object composition structure of D2O is shown. *distributed_data_objects* are composed objects, cf. Sec. 3.2.4.1. All tasks that need information related to the distribution strategy are outsourced to a *distributor*. In this figure, three *distributed_data_objects* are shown where `obj1` and `obj2` share the same distributor. This is possible because they are essentially identical: they have the same global shape, datatype, and distribution strategy. Since it is expensive to instantiate new distributors, the *distributed_data_objects* get their instance from the `distributor_factory` that takes care of caching those distributors that have already been created. Furthermore, we illustrate the `d2o_librarian` that keeps weak references to the individual *distributed_data_objects* and assigns a unique cluster-wide identifier to them: the `index`.

3.3 Basic Usage

In the subsequent sections we will illustrate the basic usage of D2O in order to explain its functionality and behavior. A more extended discussion is given in Sec. A.1. Our naming conventions are:

- instances of the `numpy.ndarray` class are labeled `a` and `b`,
- instances of `d2o.distributed_data_object` are labeled `obj` and `p`.

In addition to these examples, the interested reader is encouraged to have a look into the *distributed_data_object* method's docstrings for further information; cf. the project's web page `https://gitlab.mpcdf.mpg.de/ift/D2O`.

3.3.1 Initialization

Here we discuss how to initialize a *distributed_data_object* and compare some of its basic functionality to that of a `numpy.ndarray`. First we import the packages.

```
In [1]: import numpy as np
In [2]: from d2o import distributed_data_object
```

Now we set up some test data using numpy.

```
In [3]: a = np.arange(12).reshape((3, 4))
In [4]: a
Out[4]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

One way to initialize a *distributed_data_object* is to pass an existing numpy array.

```
In [5]: obj = distributed_data_object(a)
In [6]: obj
Out[6]: <distributed_data_object>
        array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

The output of the `obj` call shows the local portion of the global data available in this process.

3.3.2 Arithmetics

Simple arithmetics and point-wise comparison work as expected from a numpy array.

```
In [7]: (2*obj, obj**3, obj >= 5)
Out[7]: (<distributed_data_object>
         array([[ 0,  2,  4,  6],
```

```
4                [ 8, 10, 12, 14],
5                [16, 18, 20, 22]]),
6         <distributed_data_object>
7         array([[   0,    1,    8,   27],
8                [  64,  125,  216,  343],
9                [ 512,  729, 1000, 1331]]),
10        <distributed_data_object>
11        array([[False, False, False, False],
12               [False,  True,  True,  True],
13               [ True,  True,  True,  True]], dtype=bool))
```

Please note that the *distributed_data_object* tries to avoid inter-process communication whenever possible. Therefore the returned objects of those arithmetic operations are instances of *distributed_data_object*, too. However, the D2O user must be careful when combining numpy arrays with *distributed_data_objects*. If one combines two objects with a binary operator in Python (like +, -, *, \, % or **) it will try to call the respective method (`__add__`, `__sub__`, etc...) of the *first* object. If this fails, i.e. if it throws an exception, Python will try to call the *reverse* methods of the *second* object (`__radd__`, `__rsub__`, etc...):

```
1 In [8]: a + 1; # calls a.__add__(1) -> returns a numpy array
2 In [9]: 1 + a; # 1.__add__ not existing -> a.__radd__(1)
```

Depending on the conjunction's ordering, the return type may vary when combining numpy arrays with *distributed_data_objects*. If the numpy array is in the first place, numpy will try to extract the second object's array data using its `__array__` method. This invokes the *distributed_data_object*'s `get_full_data` method that communicates the full data to every process. For large arrays this is extremely inefficient and should be avoided by all means. Hence, it is crucial for performance to assure that the *distributed_data_object*'s methods will be called by Python. In this case, the locally relevant parts of the array are extracted from the numpy array and then efficiently processed as a whole.

```
1  In [10]: a + obj # numpy converts obj -> inefficient
2  Out[10]: array([[ 0,  2,  4,  6], # note: numpy.ndarray
3                  [ 8, 10, 12, 14],
4                  [16, 18, 20, 22]])
5
6  In [11]: obj + a # obj processes a -> efficient
7  Out[11]: <distributed_data_object>
8          array([[ 0,  2,  4,  6],
9                 [ 8, 10, 12, 14],
10                [16, 18, 20, 22]])
```

3.3.3 Array Indexing

The *distributed_data_object* supports most of numpy's indexing functionality, so it is possible to work with scalars, tuples, lists, numpy arrays and *distributed_data_objects* as input data. Ev-

ery process will extract its locally relevant part of the given data-object and then store it; cf. Sec. A.1.3.

```
In [12]: obj
Out[12]: <distributed_data_object>
         array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [13]: obj[1] # extract a row
Out[13]: <distributed_data_object>
         array([4, 5, 6, 7])

In [14]: obj[1,-2] # extract single entry
Out[14]: 6

In [15]: obj[::2, 1::2] # slicing notation
Out[15]: <distributed_data_object>
         array([[ 1,  3],
                [ 9, 11]])

# sets data using slicing
In [16]: obj[::2, 1::2] = [[111, 222], [333, 444]]
In [17]: obj
Out[17]: <distributed_data_object>
         array([[  0, 111,   2, 222],
                [  4,   5,   6,   7],
                [  8, 333,  10, 444]])
```

By default it is assumed that all processes use the *same* key-object when accessing data. See Sec. A.1.4 for more details regarding process-individual indexing.

3.3.4 Distribution Strategies

In order to specify the distribution strategy explicitly one may use the "`distribution_strategy`" keyword:

```
In [18]: obj = distributed_data_object(
                  a, distribution_strategy='equal')
In [19]: obj.distribution_strategy
Out[19]: 'equal'
```

See Sec. A.1.1 for more information on distribution strategies.

3.3.5 Distributed Arrays

To use D2O in a distributed manner, one has to create an MPI job. This example shows how four MPI processes hold individual parts of the global data and how distributed read & write access works. The script is started via the command:

```
mpirun -n 4 python get_set_data.py
```

```
1  # get_set_data.py
2  from mpi4py import MPI
3  import numpy as np
4  from d2o import distributed_data_object
5  # Get the process' rank number (0,1,2,3) from MPI
6  rank = MPI.COMM_WORLD.rank
7
8  # Initialize some data
9  a = np.arange(16).reshape((4,4))
10 # Initialize the distributed_data_object
11 obj = distributed_data_object(a)
12
13 # Print the process' local data
14 print (rank, obj.get_local_data())
15 # extract data via slicing
16 print (rank, obj[0:3:2, 1:3].get_local_data())
17
18 b = -np.arange(4).reshape((2,2))
19 obj[2:4,1:3] = b # Write b into obj
20
21 # Print the process' local data
22 print (rank, obj.get_local_data())
23
24 # Consolidate the data
25 full_data = obj.get_full_data()
26 if rank == 0: print (rank, full_data)
```

The *distributed_data_object* gets initialized in line 11 with the following array:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]]))
```

Here, the script is run in four MPI processes; cf. `mpirun -n 4 [...]`. The data is split along the first axis; the print statement in line 14 yields the four pieces:

```
(0, array([[ 0,  1,  2,  3]]))
(1, array([[ 4,  5,  6,  7]]))
```

```
(2, array([[ 8,  9, 10, 11]]))
(3, array([[12, 13, 14, 15]]))
```

The second print statement (line 16) illustrates the behavior of data extraction; `obj[0:3:2, 1:3]` is slicing notation for the entries 1, 2, 9 and 10[12]. This expression returns a *distributed_data_object* where the processes possess the individual portion of the requested data. This means that the distribution strategy of the new (sub-)array is determined by and aligned to that of the original array.

```
(0, array([[1, 2]]))
(1, array([], shape=(0, 2), dtype=int64)) # empty
(2, array([[ 9, 10]]))
(3, array([], shape=(0, 2), dtype=int64)) # empty
```

The result is a *distributed_data_object* where the processes 1 and 3 do not possess any data as they had no data to contribute to the slice in `obj[0:3:2, 1:3]`. In line 19 we store a small 2x2 block `b` in the lower middle of `obj`. The process' local data reads:

```
(0, array([[ 0,  1,  2,  3]]))
(1, array([[ 4,  5,  6,  7]]))
(2, array([[ 8,  0, -1, 11]]))
(3, array([[12, -2, -3, 15]]))
```

Finally, in line 25 we use `obj.get_full_data()` in order to consolidate the distributed data; i.e. to communicate the individual pieces between the processes and merge them into a single numpy array.

```
(0, array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  0, -1, 11],
           [12, -2, -3, 15]]))
```

3.4 Performance and Scalability

In this section we examine the scaling behavior of a *distributed_data_object* that uses the *equal* distribution strategy. The timing measurements were performed on the C2PAP Computing Cluster (Universe, 2016)[13]. The software stack was built upon *Intel MPI 5.1*, *mpi4py 2.0*, *numpy 1.11* and *python 2.7.11*. For measuring the individual timings we used the Python standard library module *timeit* with a fixed number of 100 repetitions.

[12]This notation can be decoded as follows. The numbers in a slice correspond to `start:stop:step` with `stop` being exclusive. `obj[0:3:2, 1:3]` means to take every second line from the lines 0, 1 and 2, and to then take from this the elements in collumns 1 and 2.

[13]The C2PAP computing cluster consists of 128 nodes, each possessing two Intel Xeon CPU E5-2680 (8 cores 2.70 GHz + hyper-threading, 64 KiB L1 per core, 256 KiB L2 cache per core, 20 MiB L3 cache shared for all 8

Please note that D2O comes with an extensive suite of unit tests featuring a high code-coverage rate. By this we assure D2O's continuous code development to be very robust and provide the users with a reliable interface definition.
Regarding D2O's performance there are two important scaling directions: the size of the array data and the number of MPI processes. One may distinguish three different contributions to the overall computational cost. First, there is data management effort that every process has to cover itself. Second, there are the costs for inter-MPI-process communication. And third, there are the actual numerical costs.
D2O has size-independent management overhead compared to numpy. Hence, the larger the arrays are for which D2O is used, the more efficient the computations become. We will see below that there is a certain array size per node – roughly 2^{16} elements – from which on d2o's management overhead becomes negligible compared to the purely numerical costs. This size corresponds to a two-dimensional grid with a resolution of 256×256 or equivalently 0.5 MiB of 64-bit doubles. In Sec. 3.4.1 we focus on this very ratio of management overhead to numerical costs.
D2O raises the claim to be able to operate well running with a single process as well as in a highly parallelized regime. In Sec. 3.4.2, the scaling analysis regarding the MPI process count is done with a fixed local array size for which the process overhead is negligible compared to the numerical costs. For this weak scaling analysis we are interested in the costs arising from inter-process communication compared to those of actual numerics.
In the following three sections, we study the strong scaling of D2O, where the performance is a result of the combination of all three cost contributions. Sec. 3.4.3 covers the case in which the number of MPI processes is increased while the array size is left constant. In Sec. 3.4.4 we compare D2O's to *DistArray*'s (Enthought, 2016) performance and finally, in Sec. 3.4.5 we benchmark D2O's strong-scaling behavior when applied to a real-world application: a Wiener filter signal reconstruction.
A discussion on D2O's efficient Python iterators can be found in Sec. A.2.

3.4.1 Scaling the Array Size

One may think of D2O as a layer of abstraction that is added to numpy arrays in order to take care of data distribution and collection among multiple MPI processes. This abstraction comes with inherent Python overhead, separately for each MPI process. Therefore, if one wants to analyze how the ratio of *management overhead* to *actual numerical effort* varies with the data size, only the individual process' data size is important. Because of this, all timing tests for this subsection were carried out with one MPI process only.
A common task during almost all numerical operations is to create a new array object for storing its results[14]. Hence, the speed of object creation can be crucial for overall performance there. Note that in contrast to a numpy array which basically just allocates RAM, several things must be done during the initialization of a *distributed_data_object*. The Python object instance itself must

cores) and 64 GiB RAM each. The nodes are connected via Mellanox Infiniband 40 Gbits/s

[14]Exceptions to this are inplace operations which reuse the input array for the output data.

be created, a distributor must be initialized which involves parsing of user input, RAM must be allocated, the *distributed_data_object* must be registered with the d2o_librarian (cf. Sec. A.1.5), and, if activated, inter-MPI-process communication must be done for performing sanity checks on the user input.
By default the initialization requires 60 µs to complete for a *distributed_data_object* with a shape of `(1,)` when run within one single MPI process. Using this trivial shape makes the costs for memory allocation negligible compared to the others tasks. Hence, those 60 µs represent D2O's constant overhead compared to numpy, since a comparable numpy array requires $\approx$ 0.4 µs for initialization.
In order to speed up the initialization process one may disable all sanity checks on the user input that require MPI communication, e.g. if the same datatype was specified in all MPI processes. Even when run with one single process, skipping those checks reduces the costs by 27 µs from 60 µs to 33 µs.
Because of the high costs, it is important to avoid building *distributed_data_objects* from scratch over and over again. A simple measure against this is to use inplace operations like `obj += 1` instead of `obj = obj + 1` whenever possible. This is generally a favorable thing to do – also for numpy arrays – as this saves the costs for repeated memory allocation. Nonetheless, also non-inplace operations can be improved in many cases, as often the produced and the initial *distributed_data_object* have all of their attributes in common, except for their data: they are of the same shape and datatype, and use the same distribution strategy and MPI communicator; cf. `p = obj + 1`. With `obj.copy()` and `obj.copy_empty()` there exist two cloning methods that we implemented to be as fast as allowed by pure Python. Those methods reuse as much already initialized components as possible and are therefore faster than a fresh initialization: for the *distributed_data_object* from above `obj.copy()` and `obj.copy_empty()` consume 7.9 µs and 4.3 µs, respectively.
Tab. 3.2 shows the performance ratio in percent between serial D2O and numpy. The array sizes range from $2^0 = 1$ to $2^{25} \approx 3.3 \cdot 10^7$ elements. In the table, 100% would mean that D2O is as fast as numpy.
The previous section already suggested that for tasks that primarily consist of initialization work – like *array creation* or `copy_empty` – D2O will clearly follow behind numpy. However, increasing the array size from 2^{20} to 2^{22} elements implies a considerable performance drop for numpy's memory allocation. This in turn means that for arrays with more than 2^{22} elements D2O's relative overhead becomes less significant: e.g. `np.copy_empty` is then only a factor of four faster than `obj.copy_empty()`.
Functions like `max` and `sum` return a scalar number; no expensive return-array must be created. Hence, D2O's overhead is quite modest: even for size 1 arrays, D2O's relative performance lies above 50%. Once the size is greater than 2^{18} elements the performance is higher than 95%.
`obj[::-2]` is *slicing syntax* for *"take every second element from the array in reverse order"*. It illustrates the costs of the data-distribution and collection logic that even plays a significant role if there is no inter-process communication involved. Again, with a large-enough array size, D2O's efficiency becomes comparable to that of numpy.
Similarly, to `obj[::-2]`, the remaining functions in the table return a *distributed_data_object* as their result and therefore suffer from its initialization costs. However, with an array size of 2^{16}

elements and larger D2O's relative performance is at least greater than approximately 65%.
An interesting phenomenon can be observed for `obj + 0` and `obj + obj`: As for the other functions, their relative performance starts to increase significantly when an array size of 2^{16} is reached. However, in contrast to `obj += obj` which then immediately scales up above 95%, the relative performance of the non-inplace additions temporarily *decreases* with increasing array size. This may be due to the fact that given our test scenario 2^{18} elements almost take up half of the cache of C2PAP's Intel E5-2680 CPUs. D2O's memory overhead is now responsible for the fact, that its non-inplace operations – which need twice the initial memory – cannot profit that strongly from the cache anymore, whereas the numpy array still operates fast. Once the array size is above 2^{22} elements numpy's just as D2O's array-object is too large for profiting from the cache and therefore become comparably fast again: the relative performance is then greater than 98%.
Thus, when run within a single process, D2O is ideally used for arrays larger than $2^{16} = 65536$ elements which corresponds to 512 KiB. From there the management overhead becomes less significant than the actual numerical costs.

3.4.2 Weak Scaling: Proportional Number of Processes and Size of Data

Now we analyze the scaling behavior of D2O when run with several MPI processes. Repeating the beginning of Sec. 3.4, there are three contributions to the execution time. First, the fixed management overhead that every process has to cover itself, second, the communication overhead and third, the actual numerical costs. In order to filter out the effect of a changing contribution of management overhead, in this section we fix the MPI processes' local array size to a fixed value. Hence, now the global data size is proportional to the number of MPI processes.
Tab. 3.3 shows the performance of various array operations normalized to the time D2O needs when running with one process only. Assuming that D2O had no communication overhead and an operation scaled perfectly linearly with array size, the performance would be rated at 100%.
In theory, operations that do not inherently require inter-process communication like point-wise array addition or subtraction ideally scale linearly. And in fact, D2O scales excellently with the number of processes involved for those functions: here we tested `copy`, `copy_empty`, `sum(axis=1)`, `obj + 0`, `obj + obj`, `obj += obj` and `sqrt`.
Comparing `sum(axis=0)` with `sum(axis=1)` illustrates the performance difference between those operations that involve inter-process communication and those that don't: the *equal* distribution strategy slices the global array along its first axis in order to distribute the data among the individual MPI processes. Hence, `sum(axis=0)` – which means to take the sum along the first axis – does intrinsically involve inter-process communication whereas `sum(axis=1)` does not. Similarly to `sum(axis=0)`, also the remaining functions in Tab. 3.3 are affected by an increasing number of processes as they involve inter-process communication.
But still, even if – for example in case of `sum(axis=0)` – the relative performance may drop to 28.2% when using 256 processes, this means that the operation just took 3.5 times longer than the single-process run, whereat the array size has been increased by a factor of 256. This corresponds to a speedup factor of 72.2.

3.4.3 Strong Scaling: Varying Number of Processes with a Fixed Size of Data

Similarly to the previous Sec. 3.4.2, we vary the number of processes but now fix the data size *globally* instead of locally. This corresponds to the real-life scenario in which the problem size and resolution are already set – maybe by environmental conditions – and now one tries to reduce the run time by using more cores. Since the size of the local data varies with the number of processes, the overall scaling behavior is now a mixture of the varying ratio between management overhead and process-individual numerical costs, and the fact that an increasing amount of CPU cache becomes available at the expense of increased communication effort. Tab. 3.4 shows the benchmarking results for the same set of operations as used in the previous section on weak scaling and the results are reasonable. Those operations in the list that inherently cannot scale strongly as they consist of purely node-individual work, namely the `initialization` and `copy_empty`, show that their performance just does not increase with the number of processes. In contrast, operations without communication effort benefit strongly from the increasing total amount of CPU cache combined with smaller local arrays; above all `copy` which is about 3 times faster than what one would expect from linear scaling to 256 processes[15].
In theory, the strong-scaling behavior is the combination of the size- and weak-scaling we discussed in Sec. 3.4.1 and Sec. 3.4.2. In order to verify whether the strong-scaling behavior makes sense, we estimate the strong-scaling performance using the information from size- and weak-scaling.
We choose the `sum()` method as our test case. During the reduction phase, the n MPI-processes exchange their local results with each other. This corresponds to adding n times a fixed amount of communication time to the total computing time needed. Hence, we perform a linear fit to the weak-scaling data; cf. Tab. 3.3. Furthermore, we assume that the local computation time is roughly proportional to the local-array size. This is true for sufficiently large array sizes, since then numpy scales linearly and D2O is in a good efficiency regime, cf. Tab. 3.2. Again we performed a linear fit but now on the size-scaling timing data. Combining those two linear fits leads to the following run-time formula for applying `sum()` to an array with shape $(4096, 4096)$:

$$t(n) = (0.0065n + 1.57/n)\ \mathrm{s} \tag{3.1}$$

In the case of linear scaling, $t(n)$ is expected to be equal to $t(1)/n$. Hence, the relative performance $p(n)$ is the ratio between the two:

$$p(n)_{\mathrm{estimated}} = \frac{t(1)/n}{t(n)} = \frac{241.8}{240.8 + n^2} \tag{3.2}$$

Comparing the estimate with the actually measured relative performance – cf. Tab. 3.4 – Tab. 3.1 shows that even under those rough assumptions the strong scaling behavior of `sum()` can be explained as the combination of size- and weak scaling within about 20 % accuracy.

[15] This very strong scaling is indeed realistic: when analyzing pure numpy arrays one gets speedups in the order of magnitude of even 800 %.

Table 3.1: Strong Scaling Behavior: Estimate vs. Measurement

#processes : n	1	4	8	16	32	64	128	256
$p(n)_{\text{estimated}}$	100 %	94.2 %	79.3 %	48.7 %	19.1 %	5.58 %	1.45 %	0.37 %
$p(n)_{\text{measured}}$	100 %	91.7 %	74.1 %	60.9 %	24.4 %	7.05 %	1.82 %	0.45 %

3.4.4 Strong Scaling: Comparison with DistArray

In Sec. 3.1.3 we discussed several competitors to D2O. Because of their similarities, we conducted the strong scaling tests – as far as possible[16] – also with DistArray and compare the performance. In Tab. 3.5 the results are shown for the subset of all operations from the previous sections that where available for DistArray, too.

While being at least on a par for numerical operations when being run single-threaded, D2O outperforms DistArray more and more with an increasing number of processes. Furthermore, it is conspicuous that DistArray does not seem to support inplace array operations. Because of this, the inplace addition `obj+=obj` is way slower with DistArray than with D2O which is on a par with numpy in most cases, cf. Tab. 3.2, Tab. 3.3 and Tab. 3.4.

The fact that D2O is way more efficient when doing numerics on very small arrays – like `obj+0` using 256 processes – indicates that D2O's organizatorial overhead is much smaller than that of DistArray. Supporting evidence for this is that the initialization of an empty DistArray (`copy_empty`) becomes disproportionately costly when increasing the number of processes used.

[16] DistArray does, for example, not support negative step-sizes for slicing (`[::-2]`) and also the special method `bincount` is not available.

array size	2^0	2^2	2^4	2^6	2^8 (2 KiB)	2^{10}	2^{12}	2^{14} (128 KiB)	2^{16}	2^{18}	2^{20} (8 MiB)	2^{22}	2^{23}	2^{24} (128 MiB)	2^{25}
initialization	*0.65%*	*0.64%*	*0.69%*	*0.69%*	*0.71%*	*0.71%*	*0.74%*	*0.72%*	*0.75%*	*0.74%*	*0.75%*	*5.41%*	*5.58%*	*5.83%*	*6.00%*
copy_empty	*3.77%*	*3.86%*	*4.00%*	*4.12%*	*4.57%*	*3.92%*	*3.95%*	*3.98%*	*4.01%*	*4.10%*	*4.15%*	*25.0%*	*24.2%*	*25.3%*	*26.0%*
max	56.2%	56.0%	54.4%	55.5%	56.0%	56.9%	62.6%	79.2%	**91.7%**	**97.5%**	**99.4%**	**99.4%**	**99.9%**	**99.8%**	**99.9%**
sum	59.7%	59.0%	57.5%	60.1%	58.6%	59.8%	62.5%	74.8%	88.4%	**95.9%**	**99.0%**	**99.3%**	**99.7%**	**99.7%**	**100.0%**
obj[::-2]	*1.18%*	*1.17%*	*1.20%*	*1.24%*	*1.34%*	*1.50%*	*2.14%*	*4.77%*	*15.0%*	*22.2%*	*28.7%*	*24.1%*	45.8%	47.3%	47.7%
copy	*8.16%*	*8.86%*	*9.30%*	*9.72%*	*9.66%*	*10.4%*	*14.8%*	*26.7%*	65.8%	**95.5%**	**98.7%**	**99.1%**	**99.9%**	**99.4%**	**98.4%**
obj + 0	*6.58%*	*6.48%*	*6.67%*	*7.08%*	*7.37%*	*9.51%*	*16.2%*	35.0%	65.2%	34.7%	44.7%	**98.7%**	**99.7%**	**96.5%**	**100.0%**
obj + obj	*3.07%*	*3.10%*	*3.24%*	*3.59%*	*3.98%*	*5.70%*	*13.3%*	31.9%	64.0%	34.6%	45.0%	**98.7%**	**99.9%**	**99.5%**	**99.8%**
obj += obj	*5.17%*	*5.35%*	*5.41%*	*6.02%*	*6.66%*	*11.3%*	*22.8%*	46.4%	75.3%	**91.9%**	**97.8%**	**98.0%**	**99.8%**	**99.3%**	**99.7%**
sqrt	*3.25%*	*3.17%*	*3.26%*	*3.50%*	*4.42%*	*7.49%*	*18.7%*	45.6%	75.8%	65.2%	83.2%	**98.8%**	**99.4%**	**99.6%**	**99.8%**
bincount	*3.57%*	*3.35%*	*3.76%*	*4.04%*	*4.97%*	*7.54%*	*16.5%*	35.4%	58.0%	75.1%	78.0%	76.8%	78.9%	82.1%	83.2%

Table 3.2: Overhead costs: D2O's relative performance to numpy when scaling the array size and being run with one MPI-process. "100%" corresponds to the case when D2O is as fast as numpy. In order to guide the eye, values < 30% are printed italic, values ≥ 90% are printed bold. Please see Sec. 3.4.1 for discussion.

process count	1	2	3	4	8	16	32	64	128	256
initialization	**100.0%**	**90.9%**	87.9%	87.8%	74.6%	67.6%	54.9%	45.7%	34.6%	*19.9%*
copy_empty	**100.0%**	**97.5%**	**96.2%**	**97.5%**	**97.6%**	**103.6%**	**97.8%**	**97.7%**	**97.6%**	**95.1%**
max	**100.0%**	**97.5%**	**96.6%**	**95.6%**	**90.9%**	84.0%	72.1%	56.2%	39.1%	*24.3%*
sum	**100.0%**	**98.0%**	**95.3%**	**93.5%**	87.3%	79.2%	65.1%	48.3%	32.2%	*19.2%*
sum(axis=0)	**100.0%**	**100.2%**	**96.7%**	**96.5%**	**90.9%**	78.1%	74.6%	58.0%	42.7%	*28.2%*
sum(axis=1)	**100.0%**	**105.2%**	**103.2%**	**102.2%**	**100.6%**	**100.0%**	**98.3%**	**95.8%**	**93.2%**	88.6%
obj[::-2]	**100.0%**	70.4%	65.9%	64.0%	46.2%	46.6%	42.8%	33.6%	31.1%	*25.3%*
copy	**100.0%**	**104.7%**	**103.1%**	**101.3%**	**101.3%**	**105.3%**	**101.4%**	**101.2%**	**101.3%**	**101.5%**
obj + 0	**100.0%**	**105.1%**	**102.6%**	**100.6%**	**99.9%**	**103.5%**	**100.2%**	**100.0%**	**99.7%**	**100.1%**
obj + obj	**100.0%**	**105.2%**	**102.5%**	**100.1%**	**100.0%**	**103.7%**	**100.1%**	**100.1%**	**99.8%**	**100.2%**
obj += obj	**100.0%**	**102.3%**	**99.3%**	**98.6%**	**98.2%**	**101.8%**	**98.2%**	**98.2%**	**98.2%**	**98.4%**
sqrt	**100.0%**	**102.0%**	**100.6%**	**100.1%**	**99.6%**	**99.1%**	**99.2%**	**99.2%**	**98.6%**	**98.0%**
bincount	**100.0%**	**103.0%**	**101.2%**	**99.9%**	**98.8%**	**97.6%**	**94.1%**	88.3%	79.4%	65.8%

Table 3.3: Weak scaling: D2O's relative performance to the single-process case when increasing both, the number of processes and the global array size proportionally. The arrays used for this tests had the global shape $(n * 2048, 2048)$ with n being the number of processes. By this the local data size was fixed to 2^{22} elements, which is equal to 32 MiB. "100%" in the table corresponds to the case were the speedup is equal to the number of processes. Example: the 95.1% for `copy_empty` on 256 processes correspond to a speedup-factor of 243.5. In order to guide the eye, values < 30% are printed italic, values ≥ 90% are printed bold. Please see Sec. 3.4.2 for discussion.

processes (local size)	1 (128 MiB)	2 (64 MiB)	3 (42.7 MiB)	4 (32 MiB)	8 (16 MiB)	16 (8 MiB)	32 (4 MiB)	64 (2 MiB)	128 (1 MiB)	256 (512 KiB)
initialization	**100.0%**	40.3%	*23.7%*	*17.1%*	*6.80%*	*2.55%*	*0.89%*	*0.29%*	*0.10%*	*0.03%*
copy_empty	**100.0%**	47.8%	33.5%	*26.3%*	*23.2%*	*12.1%*	*6.16%*	*3.09%*	*1.55%*	*0.79%*
max	**100.0%**	**99.0%**	**96.7%**	**94.0%**	80.1%	64.3%	*29.6%*	*9.10%*	*2.41%*	*0.60%*
sum	**100.0%**	**100.4%**	**95.4%**	**91.7%**	74.1%	60.9%	*24.4%*	*7.05%*	*1.82%*	*0.45%*
sum(axis=0)	**100.0%**	**98.0%**	**94.2%**	89.9%	62.2%	45.3%	*19.8%*	*6.34%*	*1.78%*	*0.45%*
sum(axis=1)	**100.0%**	**100.5%**	**92.3%**	**92.6%**	79.0%	77.7%	47.1%	*20.6%*	*4.27%*	*1.25%*
obj[::-2]	**100.0%**	65.4%	62.4%	58.5%	40.7%	33.1%	*26.6%*	*18.3%*	*8.78%*	*3.25%*
copy	**100.0%**	**103.6%**	**105.9%**	**98.0%**	**145.1%**	**156.4%**	**155.3%**	**152.3%**	**157.5%**	**306.7%**
obj + 0	**100.0%**	**105.9%**	**109.1%**	**100.4%**	59.0%	**97.3%**	75.7%	79.2%	79.4%	**149.8%**
obj + obj	**100.0%**	**106.2%**	**109.2%**	**100.3%**	59.0%	**97.6%**	74.9%	79.0%	80.1%	**150.2%**
obj += obj	**100.0%**	**103.1%**	**101.3%**	**98.0%**	**97.8%**	**124.0%**	**122.8%**	**117.9%**	**108.1%**	**94.4%**
sqrt	**100.0%**	**101.8%**	**99.4%**	**98.7%**	**95.7%**	88.8%	76.0%	56.0%	37.1%	*16.4%*
bincount	**100.0%**	**102.3%**	**99.5%**	**98.0%**	**111.1%**	**107.3%**	84.2%	40.5%	*13.2%*	*3.57%*

Table 3.4: Strong scaling: D2O's relative performance to a single process when increasing the number of processes while fixing the global array size to $(4096, 4096)$ = 128 MiB. "100%" corresponds to the case were the speedup is equal to the number of processes. Example: the 94.4% for `obj+=obj` on 256 processes correspond to a speedup-factor of 241.7. In order to guide the eye, values < 30% are printed italic, values ≥ 90% are printed bold. Please see Sec. 3.4.3 for discussion.

processes (local size)	1 (128 MiB)	2 (64 MiB)	3 (42.7 MiB)	4 (32 MiB)	8 (16 MiB)	16 (8 MiB)	32 (4 MiB)	64 (2 MiB)	128 (1 MiB)	256 (512 KiB)
copy_empty	**23.49**	**27.70**	**33.99**	**40.03**	$\mathbf{1.11 \cdot 10^2}$	$\mathbf{1.12 \cdot 10^3}$	$\mathbf{1.06 \cdot 10^3}$	$\mathbf{1.91 \cdot 10^3}$	$\mathbf{5.57 \cdot 10^3}$	$\mathbf{1.90 \cdot 10^4}$
max	**1.05**	**1.11**	**1.16**	**1.20**	**1.34**	**5.00**	**4.74**	**3.77**	**3.25**	**4.17**
sum	**1.07**	**1.20**	**1.25**	**1.33**	**1.48**	**6.57**	**5.57**	**4.20**	**3.55**	**4.61**
sum(axis=0)	**1.02**	**1.09**	**1.12**	**1.22**	**1.03**	**3.61**	**3.35**	**2.75**	**2.42**	**3.06**
sum(axis=1)	**1.03**	**1.15**	**1.15**	**1.28**	**1.63**	**11.35**	**12.56**	**19.29**	**22.55**	**42.48**
obj + 0	**1.02**	**1.09**	**1.20**	**1.15**	**0.44**	**2.88**	**4.24**	**7.81**	**33.94**	$\mathbf{3.78 \cdot 10^2}$
obj + obj	**1.02**	**1.09**	**1.20**	**1.16**	**0.44**	**2.90**	**4.23**	**8.08**	**34.40**	$\mathbf{3.81 \cdot 10^2}$
obj += obj	**2.27**	**2.38**	**2.51**	**2.53**	**1.60**	**8.23**	**14.89**	**26.24**	$\mathbf{1.04 \cdot 10^2}$	$\mathbf{5.35 \cdot 10^2}$
sqrt	**1.00**	**1.03**	**1.03**	**1.04**	**0.81**	**1.57**	**2.06**	**2.59**	**6.66**	**16.77**

Table 3.5: Strong scaling comparison: D2O's relative performance compared to DistArray (Enthought, 2016) when increasing the number of processes while fixing the global array size to $(4096, 4096)$ = 128 MiB. "2" corresponds to the case were D2O is twice as fast as DistArray. Please see Sec. 3.4.4 for discussion.

3.4.5 Strong Scaling: Real-World Application Speedup – the Wiener filter

D2O was initially developed for NIFTY(Selig et al., 2013), a library for building signal inference algorithms in the framework of *information field theory* (IFT) (Enßlin et al., 2009). Within NIFTY v2 all of the parallelization is done via D2O; the code of NIFTY itself is almost completely agnostic of the parallelization and completely agnostic of MPI.
A standard computational operation in IFT-based signal reconstruction is the *Wiener filter* (Wiener, 1949). For this performance test, we use a Wiener filter implemented in NIFTY to reconstruct the realization of a Gaussian random field – called the *signal*. Assume we performed a hypothetical measurement which produced some *data*. The data model is

$$\text{data} = R(\text{signal}) + \text{noise} \tag{3.3}$$

where R is a smoothing operator and *noise* is additive Gaussian noise. In Fig. 3.2 one sees the three steps:

- the true signal we try to reconstruct,
- the data one gets from the hypothetical measurement, and
- the reconstructed signal field that according to the Wiener filter has most likely produced the *data*.

Tab. 3.6 shows the scaling behavior of the reconstruction code, run with a resolution of 8192×8192. Here, n is the number of used processes and t_n the respective execution time. The relative speedup $s_n = 2t_2/t_n$[17] is the ratio of execution times: parallel versus serial. In the case of exactly linear scaling s_n is equal to n. Furthermore we define the scaling quality $q = 1/(1 + \log(n/s_n))$, which compares s_n with linear scaling in terms of orders of magnitude. A value $q = 1$ represents linear scaling and $q \geq 1$ super-linear scaling.

Table 3.6: Execution time scaling of a Wiener filter reconstruction on a grid of size 8192×8192.

#nodes	1	1	1	1	2	4	8	16	32
#processes : n	1	2	3	4	8	16	32	64	128
t[s]	1618	622.0	404.2	364.2	181.7	94.50	46.79	18.74	8.56
$s_n = 2t_2/t_n$	0.769	2.00	3.08	3.42	6.85	13.2	26.6	66.4	145
$q_n = 1/(1+\log(\frac{n}{s_n}))$	0.900	1.00	1.01	0.94	0.94	0.92	0.93	1.02	1.06

This benchmark illustrates that even in a real-life application super-linear scaling is possible to achieve for a sufficiently large number of processes. This is due to the operations that are needed in order to perform the Wiener filtering: basic point-wise arithmetics that do not involve any inter-process communication and Fourier transformations that are handled by the high-performance library *FFTW* (Frigo and Johnson, 2005). While the problem size remains constant, the amount of available CPU cache increases with the number of processes, which explains the super-linear scaling, cf. Sec. 3.4.3.

[17] Since the combination of NIFTy and pyfftw exhibits an unexpected speed malus for one process, we chose the

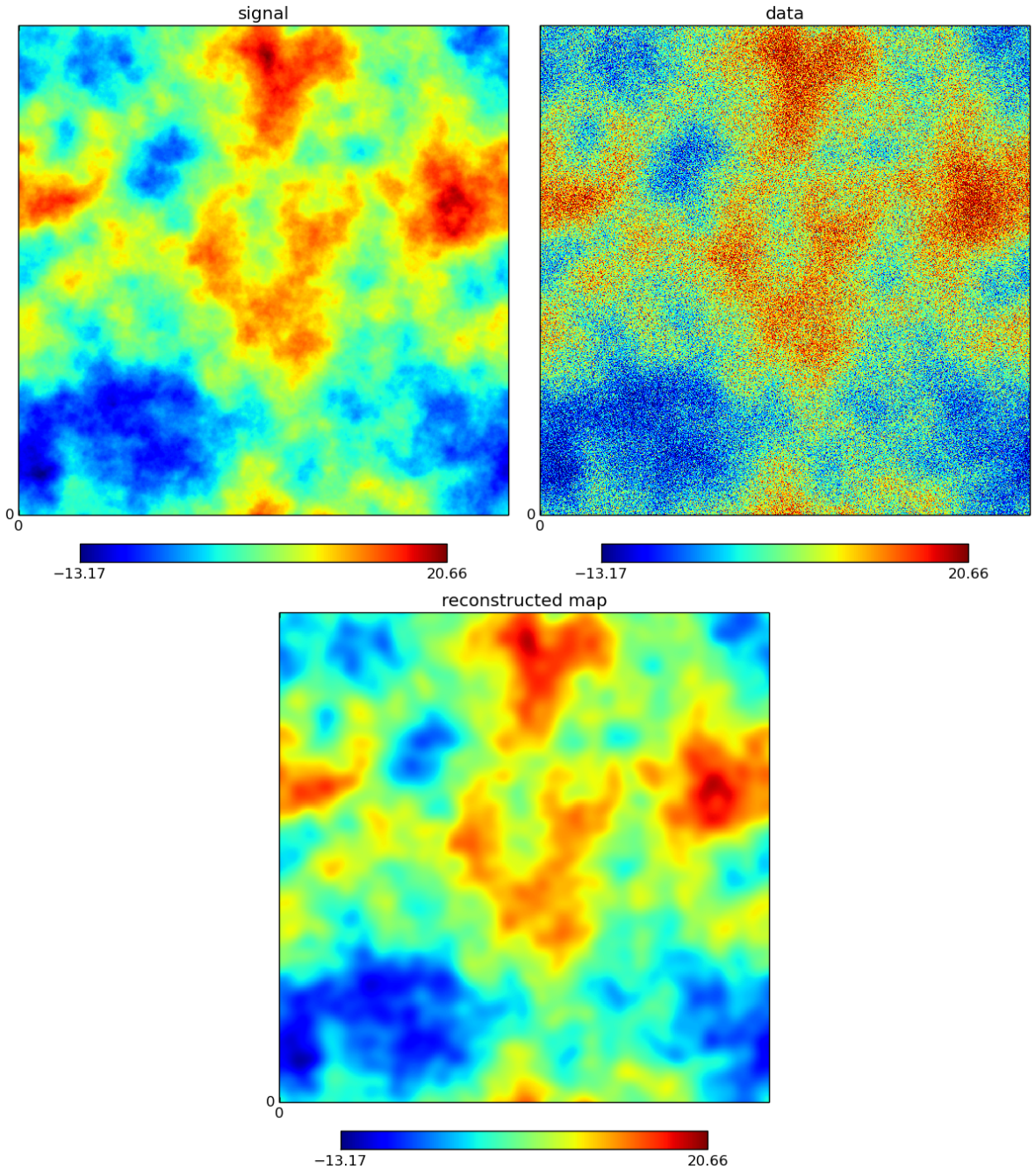


Figure 3.2: Wiener filter reconstruction. Top left: True signal to be reconstructed. Top right: Data which is the result of smoothing the signal and adding Gaussian noise. Bottom: reconstructed maximum of the posterior (conditional) probability density function for the signal given the data $\mathcal{P}$(signal|data).

3.5 Summary & Outlook

We introduced D2O, a Python module for cluster-distributed multi-dimensional numerical arrays. It can be understood as a layer of abstraction between abstract algorithm code and actual data-distribution logic. We argued why we developed D2O as a package following a low-level parallelization ansatz and why we built it on MPI. Compared to other packages available for data parallelization, D2O has the advantage of being ready for action on one as well as several hundreds of CPUs, of being highly portable and customizable as it is built with Python, that it is faster in many circumstances, and that it is able to treat arrays of arbitrary dimension.
For the future, we plan to cover more of *numpy*'s interface such that working with D2O becomes even more convenient. Furthermore we evaluate the option to build a D2O distributor in order to support *scalapy*'s block-cyclic distribution strategy directly. This will open up a whole new class of applications D2O then can be used for.
D2O is open source software licensed under the GNU General Public License v3 (GPL-3) and is available by `https://gitlab.mpcdf.mpg.de/ift/D2O`.

Competing interests

The authors declare that they have no competing interests.

Author's contributions

TS is the principal researcher for the work proposed in this article. His contributions include the primal idea, the implementation of presented software package, the conduction of the performance tests and the writing of the article. MG, FB and TE helped working out the conceptual structure of the software package and drafting the manuscript. FB also played a pivotal role for executing the performance tests. Furthermore, TE also fulfilled the role of the principal investigator. All authors read and approved the final manuscript.

Acknowledgments

We want to thank Jait Dixit, Philipp Franck, Reimar Leike, Fotis Megas, Martin Reinecke and Csongor Varady for useful discussions and support. We acknowledge the support by the DFG Cluster of Excellence "Origin and Structure of the Universe" and the Studienstiftung des deutschen Volkes. The performance tests have been carried out on the computing facilities of the Computational Center for Particle and Astrophysics (C2PAP). We are grateful for the support by Dr. Alexey Krukau through the Computational Center for Particle and Astrophysics (C2PAP).

two-process timing as the benchmark's baseline.

Chapter 4

NIFTy 3 - Numerical Information Field Theory - A Python framework for multicomponent signal inference on HPC clusters

This chapter is additionally used as a journal publication submitted to PLOS ONE *(Steininger et al., 2017).*
I am the principal researcher of the research described in this chapter. My contributions include the primal idea for the design and the implementation of presented software package, the conduction of the performance tests and the writing of the chapter. Jait Dixit (JD), Maksim Greiner (MG), Daniel Pumpe (DP), Martin Reinecke (MR), and Csongor Varady (CV) helped working out the conceptual structure of the software package. JD, Philipp Frank (PF), Sebastian Hutschenreuter (SH), Jakob Knollmüller (JK), Reimar Leike (RL), Natalia Porqueres (NP), DP, MR, Matevž Šraml (MS), and CV helped implementing and especially documenting the code, and, together with Torsten Enßlin (TE) helped to focus the work such that it has a high added-value in real-world applications. JD, MG, JK, RL, NP, DP, MR, CV and TE helped drafting the article: Specifically, MG contributed approximately 50% to Sec. 4.1. TE contributed approximately 90% to Sec. 4.2.1, Sec. 4.2.2, and Sec. 4.2.3. JK contributed approximately 50% to Sec. 4.3.3.1. RL, NP, and MR contributed approximately 5% to Sec. 4.4.1, each. JD, DP, and CV contributed approximately 5% to Sec. 4.4.2, each. Furthermore, TE also fulfilled the role of a principal investigator as he is my PhD supervisor. All authors read, commented, and approved the final manuscript.

Abstract

NIFTy, "Numerical Information Field Theory", is a software framework designed to ease the development and implementation of field inference algorithms especially – but not solely – used in astrophysics. Field equations of the quantities of interest are formulated independently of the

underlying spatial geometry allowing the user to focus on the algorithmic design. Under the hood, NIFTy ensures that the discretization of the implemented equations is consistent. This enables the user to prototype an algorithm rapidly in 1D and then apply it to high-dimensional real-world problems. This paper introduces NIFTy 3, a major upgrade to the original NIFTy framework. NIFTy 3 allows the user to run inference algorithms on massively parallel high performance computing clusters without changing the implementation of the field equations. It supports n-dimensional Cartesian spaces, spherical spaces, power spaces, and product spaces as well as transforms to their harmonic counterparts. Furthermore, NIFTy 3 is able to treat non-scalar fields. The functionality and performance of the software package is demonstrated with example code, which implements a real inference algorithm from the realm of information field theory. NIFTy 3 is open-source software available under the GNU General Public License v3 (GPL-3) at `https://gitlab.mpcdf.mpg.de/ift/NIFTy/`.

4.1 Introduction

Physical quantities are only accessible to the observer via measurements. Those measurements are never a perfect mapping of the quantity of interest, the so-called signal. They are virtually always subject to noise, which cannot be distinguished from the signal. Usually measurements capture the quantity of interest only indirectly, be it by event rates, receiver voltages, absorption or reflection patterns, or photon counts. Such indirect measurements cannot capture all properties of the signal.

There are two ends from which this shortcoming can be approached. The first one is to improve the measurement itself. This includes the development of more precise instruments, better observation strategies, and completely new experimental designs. The second one is to improve the processing of the data, i.e. the way how the signal is estimated once the data have been taken. This is the domain of information theory.

Modern information theory is based on the work of Cox (1946), Shannon (1948) and Wiener (1949). It has seen tremendous advances since the development of computers, which allow the calculation of statistical estimates in the absence of closed form solutions through methods such as gradient descent (e.g. Fletcher and Powell, 1963) and sampling (e.g. Metropolis et al., 1953). In the last decades, data sets have become larger and larger, which makes their processing feasible only on a computer, even if a closed form solution is available.

Many physical signals are fields, quantities that are described as functions over a continuous domain (e.g. time or space). Applying the rules of information theory to fields means applying the calculus of probabilities to functional spaces. This leads to a statistical field theory, which we call information field theory (IFT) (Enßlin and Weig, 2010, Enßlin et al., 2009). Since fields have an infinite amount of degrees of freedom by nature, the calculations cannot directly be implemented on a computer, which can only store and process a finite amount of bits. Careful considerations have to be made in discretizing the continuous equations to calculate statistical estimates that are independent of the chosen resolution, as long as all scales that are imprinted in the data are resolved. These considerations led to the development of NIFTy (Selig et al., 2013).

NIFTy – "Numerical Information Field Theory" – is a software framework in which algorithms

for information extraction from data and signal reconstruction are implemented. Its purpose is to ease the technical as well as conceptual difficulties which arise when working with IFT. To that end NIFTy was designed in a way that equations involving fields are formulated independently of the underlying spatial geometry and discretization. Thus, an algorithm implemented with NIFTy to infer a field living on a sphere, can be transformed to infer a field living in a three-dimensional Cartesian space by the change of a single line of code. NIFTy is a development framework written in Python[1]. The structure NIFTy brings in terms of a development framework is bundled with the power of external compiled-language modules for numerical efficiency.
As it greatly reduced the complexity of implementing algorithms for field inference, many applications of IFT have been implemented using NIFTy, most notably the reconstruction of the primordial scalar potential (Dorn et al., 2015b), the estimation of extragalactic Faraday rotation (Oppermann et al., 2015), and the denoised, deconvolved, and decomposed Fermi γ-ray sky (Selig et al., 2015). However, more and more complex and ambitious field inference projects brought the original NIFTy package to its limits. To advance further, the capability to make use of massively parallel high performance computing clusters, treat products of spaces, and work with non-scalar fields is needed. These requirements have led to a complete redesign of NIFTy. This new software framework, NIFTy 3, which additionally includes also numerous minor advances, is presented here and is available at `https://gitlab.mpcdf.mpg.de/ift/NIFTy`.

4.2 Problem Description

4.2.1 Information Field Theory

In general, a signal inference problem tries to invert a measurement equation

$$d = f(s, n), \tag{4.1}$$

that describes how the obtained data d depends on the unknown signal s and further nuisance parameters n. Actually, we want to know s and are not interested in n, but the latter also influences the measurement outcome. The problem is that the function f is not necessarily invertible in s, and that the nuisance parameters may not be known.
This problem is particularly severe when the signal is a field, viz. a continuous function $s(x)$ over a manifold Ω. In this case the number of unknowns is infinite, whereas the number of knowns, the components of the data vector, is finite. The reconstruction of a field from finite data therefore always requires the usage of additional information. Optimally this is given in the form of a joint probability distribution $\mathcal{P}(s, n)$ for all the unknowns. Hence, this induces a joint probability for data and signal,

$$\begin{aligned} \mathcal{P}(d, s) &= \int \mathcal{D}n\, \mathcal{P}(d|s, n)\, \mathcal{P}(s, n) \\ &= \int \mathcal{D}n\, \delta(d - f(s, n))\, \mathcal{P}(s, n), \end{aligned} \tag{4.2}$$

[1] Python homepage `http://www.python.org/`

where $\int \mathcal{D}n$ denotes the integration over all degrees of freedom of n. The posterior distribution for the unknown signal is then given by Bayes' theorem:

$$\mathcal{P}(s|d) = \frac{\mathcal{P}(d,s)}{\mathcal{P}(d)} = \frac{e^{-\mathcal{H}(d,s)}}{\mathcal{Z}(d)}, \tag{4.3}$$

where we introduced the language of statistical field theory by defining

$$\mathcal{H}(d,s) := -\ln \mathcal{P}(d,s) \quad \text{and} \tag{4.4}$$

$$\mathcal{Z}(d) := \mathcal{P}(d) = \int \mathcal{D}s\, e^{-\mathcal{H}(d,s)}. \tag{4.5}$$

Hereafter we focus on a special case that leads to the well-known Wiener filter theory.

4.2.2 Wiener Filter Theory

A simple example for an inference problem is given by a linear measurement of a signal s with additive and signal independent noise n

$$d = Rs + n \tag{4.6}$$

of a Gaussian signal

$$s \hookleftarrow \mathcal{P}(s) = \mathcal{G}(s,S) = \frac{1}{|2\pi S|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}\, s^\dagger S^{-1} s\right) \tag{4.7}$$

with additive and signal independent Gaussian noise $n \hookleftarrow \mathcal{P}(n|s) = \mathcal{G}(n,N)$ and known signal and noise covariances $S = \langle s\, s^\dagger \rangle_{(s)}$ and $N = \langle n\, n^\dagger \rangle_{(n)}$. $\dagger$ denotes transposition and complex conjugation so that $s^\dagger t = \int_\Omega dx\, s^*(x)\, t(x)$ is a scalar and $\langle f(x) \rangle_{(x|y)} = \int \mathcal{D}x \mathcal{P}(x|y)\, f(x)$ is the expectation of $f(x)$ over $\mathcal{P}(x|y)$.
The response R maps the continuous field s onto a discrete data vector d. For individual entries of this data vector Eq. (4.6) reads

$$d_i = \int_\Omega \mathrm{d}x\; R_i(x) s(x) + n_i \tag{4.8}$$

Here, $i \in \{1, \ldots, N_{\text{data}}\}$ with $N_{\text{data}} \in \mathbb{N}$ is the index of the data vector's entries. Ω is the physical manifold the signal is defined on.
Since Eq. (4.6) maps a continuous space with infinitely many degrees of freedom onto a finite-dimensional data vector and involves an additive stochastic noise term, the problem is not uniquely invertible. This means that many combinations of s- and n-realizations can result in the same data vector. One can now apply an inference algorithm to find the 'best'[2] estimate for s. One possibility here is the Wiener filter, which provides the posterior mean signal as the field estimate,

$$m = \left(S^{-1} + R^\dagger N^{-1} R\right)^{-1} R^\dagger N^{-1} d, \tag{4.9}$$

[2]The exact meaning of 'best' depends on the algorithm.

where S is the covariance of the signal and N the covariance of the noise. The superscript $\dagger$ denotes complex conjugation and transposition. Spelled out, this means the equation

$$\int_\Omega \mathrm{d}y \left(S^{-1}(x,y) + \sum_{i,j=1}^{N_{\mathrm{data}}} R_i(x) N_{ij}^{-1} R_j(y) \right) m(y) = \sum_{i,j=1}^{N} R_i(x) N_{ij}^{-1} d_j \tag{4.10}$$

is solved for the estimate $m(x)$.
Eq. (4.8) already shows the three main components of data equations in the area of signal reconstruction: *operators* (R) that act on *fields* (s) which are defined on a *space* (Ω). The signal posterior is also a Gaussian,

$$\begin{aligned} \mathcal{P}(s|d) &= \mathcal{G}(s-m, D) \text{ with} && (4.11)\\ D &= \left(S^{-1} + R^\dagger N^{-1} R \right)^{-1}, && (4.12)\\ m &= D\, j, \text{ and} && (4.13)\\ j &= R^\dagger N^{-1} d. && (4.14) \end{aligned}$$

The signal posterior mean $m = \langle s \rangle_{(s|d)}$ is obtained from the data by projecting it into the data space as $j = R^\dagger N^{-1} d$ after inverse noise weighting and then by operating on it with the so called information propagator D. The operation that turns the data into the signal estimate $m = F_{\mathrm{W}} d$,

$$F_{\mathrm{W}} = \left(S^{-1} + R^\dagger N^{-1} R \right)^{-1} R^\dagger N^{-1}, \tag{4.15}$$

is called the Wiener filter (Wiener, 1949). The posterior signal uncertainty $\langle (s-m)\,(s-m)^\dagger \rangle_{(s|d)}$ is also given by D, which is also called Wiener variance.
Eq. (4.15) illustrates the complexity operators can have. They are composed of a number of operators that act on different spaces that often need to be inverted as well. Since the number of pixels used to represent fields can be large, an explicit storage of such operators is prohibitive. They have to be represented by computer routines that perform their actions. This, however, makes it difficult to access the entries of an operator, for example to examine the uncertainty of the Wiener filter reconstruction, which is encoded in the entries of D.
The building blocks of D, the signal and noise covariances S and N, might be diagonal in harmonic spaces. In contrast, the response is usually best described in position space. Thus, the representation of $D^{-1} = S^{-1} + R^\dagger N^{-1} R$ requires harmonic transforms, e.g. Fourier transforms, and the application of D to j requires the solving of the linear system $D^{-1} m = j$, which asks for many evaluations of such transforms.
In Fig. 4.1 a minimal Wiener filter implementation in NIFTy 3 is shown. Thereby, an exemplary geometry is set up in the beginning and in the end the Wiener filter is applied explicitly. In the middle part the definition and initialization of the data (`d`), the signal- and noise-covariances (`S` and `N`), and the response operator (`R`) are left out, since this step is highly problem specific. Please refer to Sec. 4.5 for an exhaustive discussion of a Wiener filter implementation.

4.2.3 Interacting Information Field Theory

If any of the simplifying assumptions of the Wiener filter problem is violated, e.g. if signal or noise are not Gaussian, the noise depends on the signal, or one of the operators R, S, and N is

```python
# Import NIFTy package
import nifty as ift

# Import Wiener filter from nifty.library
from ift.library import WienerFilterCurvature

# Setting up...
#   ...position space (RGSpace as an example)
s_space = RGSpace([512, 512])
#   ...an operator for harmonic transforms
fft = FFTOperator(s_space)
#   ...harmonic space
h_space = fft.target[0]
#   ...power space
p_space = PowerSpace(h_space)

# Define specific components here
d, S, N, R = ...

# Compute the "information source"
j = R.adjoint_times(N.inverse_times(d))

# Setting up (the inverse of) the propagator
C = WienerFilterCurvature(S=S, N=N, R=R)

# Apply the Wiener filter
m = C.inverse_times(j)
```

Figure 4.1: Minimal example for a Wiener filter reconstruction in two dimensions, whereby the instantiation of problem specific components is omitted.

not known a priori, the inference problem becomes much more difficult and the posterior mean is not obtained by a linear transformation of the data.
However, even then an information Hamiltonian $\mathcal{H}(d, s) = -\ln \mathcal{P}(d, s)$ can be defined, which encodes all the information on the field. One might want to minimize this Hamiltonian with respect to the field s to obtain the maximum a posteriori estimate for it. Or one uses this Hamiltonian within a variational Bayes scheme, in which an approximate posterior $\widetilde{\mathcal{P}}(s|d)$ (usually a Gaussian) is matched to the full posterior via their Kullback-Leibler distance

$$\begin{aligned} \mathrm{KL}(\widetilde{\mathcal{P}}, \mathcal{P}) &= \int \mathcal{D}s\, \widetilde{\mathcal{P}}(s|d)\, \ln\left(\frac{\widetilde{\mathcal{P}}(s|d)}{\mathcal{P}(s|d)}\right) \\ &= \left\langle \mathcal{H}(s|d) - \widetilde{\mathcal{H}}(s|d) \right\rangle_{\widetilde{\mathcal{P}}(s|d)}. \end{aligned} \tag{4.16}$$

Therefore, dedicated numerical infrastructure is required to minimize such functionals efficiently.

4.2.4 Manifold Independence & Discretized Continuum

It is remarkable that the abstract algorithm for the inference of s (e.g. Eq. (4.9)) is independent of the choice of Ω. In a concrete implementation, however, a discretization of Ω must be provided, since classical computers perform only discrete arithmetic. Still, because of this independence of Ω, the implementation should be modular with respect to it. NIFTy 1 was specifically designed for separating the inference algorithm from the problem's manifold. For this the concept of space- and field-objects was introduced: spaces describe the geometrical properties of physical space, whereas fields are defined on spaces and carry actual data. As a consequence, algorithms implemented within the NIFTy framework are by design manifold and resolution independent. Particularly, NIFTy takes care of correctly including *volume factors* (V) whenever scalar products or, generally speaking, index contractions between tensors are performed. Volume factors are a direct consequence of the discretization of Ω. Giving an example, the scalar product of two signal fields s and u on Ω behaves like

$$s^\dagger u = \int_\Omega \mathrm{d}x\; s^*(x) u(x) \approx \sum_{q=1}^{Q} V_q\, s_q^* u_q. \tag{4.17}$$

Compare Selig et al. (2013, chapter 2.2) for more details.

For completeness, two further key concepts of NIFTy are discussed.

4.2.5 Data Representation

For many signal reconstruction problems one can assume that the signal s is statistically homogeneous and isotropic. This means that the statistical properties of $s(x)$ are independent of *position* in Ω for the former, and from *direction* for the latter. Such fields have the remarkable property that their correlation structure can be described by a one-dimensional power spectrum which is

diagonal in a harmonic basis of Ω. Hence, it is very beneficial to perform covariance-related calculations in the harmonic basis, since the involved Fourier transforms scale[3] like $\mathcal{O}(N_{\mathrm{pix}} \log(N_{\mathrm{pix}}))$ and therefore are cheap compared to full rank matrix multiplications scaling like $\mathcal{O}(N_{\mathrm{pix}}^2)$, where N_{pix} is the cardinality of Ω's pixelization. NIFTy has representations for its supported manifolds in signal as well as harmonic bases, and transforming fields from one representation to another is done via efficient external libraries (Frigo and Johnson, 2005, Gorski et al., 2005).

4.2.6 Implicit vs. Explicit Operators

In the previous section we have seen that in many cases one can choose a basis for which the covariance operators are diagonal. However, in general an arbitrary measurement device's response is not a square operator and therefore is not diagonalizable. Expressing the operator explicitly as a matrix of shape $(N_{\mathrm{pix}}, N_{\mathrm{data}})$ leads to problems if N_{pix} is large: first, matrix multiplications will slow down the reconstruction algorithm as they scale like $\mathcal{O}(N_{\mathrm{pix}}^2)$ and second, the operator may simply not fit into main memory. Because of this, operators in NIFTy are almost always stored as implicit objects. This means that instead of storing every entry of an operator's matrix representation explicitly, only its *action* is implemented. However, expressing operators implicitly has the drawback that properties like the operator's diagonal, trace or determinant can – within reasonable computing time – only be determined approximately by probing (Hutchinson, 1989) that might be combined with an inference step (Dorn and Enßlin, 2015, Selig et al., 2012). Also, the inverse of an implicitly defined operator must be computed using numerical inversion algorithms like the conjugate gradient method (Hestenes and Stiefel, 1952), whenever it is applied to a vector. To overcome those inconveniences, NIFTy has the built-in functionality to compute those properties and actions transparently for the user.

4.2.7 Reference Projects

The NIFTy framework provides rich structure in terms of a class hierarchy and built-in functionality, while at the same time custom objects – first and foremost operators – can be implemented easily. Therefore, it is well suited for rapidly prototyping new inference algorithms, as well as for building full-grown signal reconstruction applications. Because of this, since its release in 2013, NIFTy 1 has been used in various published scientific codes and papers. Some noteworthy examples are:

- Estimating galactic and extragalactic Faraday rotation (Oppermann et al., 2012, 2015, Vacca et al., 2016)
- D^{3}PO for photo count imaging (Selig and Enßlin, 2015)
- Signal inference with unknown response (Dorn et al., 2015a)
- Cosmic expansion history reconstruction from SNe Ia data (Porqueres et al., 2017)

[3] Note that for regular grids *fast* Fourier transforms exist that scale with the aforementioned $\mathcal{O}(N_{\mathrm{pix}} \log(N_{\mathrm{pix}}))$. On curved manifolds like the sphere and for non-regular spaced discretizations the costs may be higher.

- Dynamic system classifier (Pumpe et al., 2016)
- PySESA for geospatial data analysis (Buscombe, 2016)
- RESOLVE for radio interferometry (Junklewitz et al., 2016)
- Tomography of the Galactic free electron density (Greiner et al., 2016)
- Bayesian weak lensing tomography (Böhm et al., 2017)
- Noisy independent component analysis of auto-correlated components (Knollmüller and Enßlin, 2017)

4.3 Limitations of NIFTy 1

Despite all its strengths, NIFTy 1 has some considerable limitations which were the cause for developing NIFTy 3. Those limitations are discussed in the following.

4.3.1 Combined Manifolds & Field Types

As discussed in Sec. 4.2.4, NIFTy separates the domain on which a field lives from the field itself. The code architecture was designed for natively supporting only scalar fields on domains consisting of only one space. However, there are many physical applications where this is too limiting.
For example, the D^{3}PO code (Selig and Enßlin, 2015) performs a reconstruction of the diffuse and point-like γ-ray sky emissivity for a given set of photon counts. This can be done for photon counts at arbitrary energies permitting the construction of sky images in distinct energy bands. However, the individual energy bands are reconstructed separately so far; i.e. for a fixed pixel and fixed energy band the relevant information from nearby energy bands is not exploited. The next logical step is to reconstruct and decompose the true sky emissivity for all energy bands jointly within one single reconstruction. In this scenario information about the energy correlation function should be used for linking and letting the information cross-talk between the energy bands. This involves fields living on a domain which is the combination, viz. the Cartesian product, of the geometrical space (the celestial sphere) and the energy dimension (a one-dimensional Cartesian space).
Another example is the reconstruction of the Galactic free electron density by Greiner et al. (2016). The algorithm was initially built for reconstructing scalar quantities. But in principle the algorithm can be used for reconstructing vector fields, e.g. magnetic fields, or even tensor fields as well. This makes it necessary to be able to specify fields of arbitrary type; e.g. scalar, vector or tensor type.
Independently of its technical challenges, dealing with power spectra on domains being the Euclidean product of single spaces is a conceptually non-trivial task. Analyzing power spectra, as well as drawing random samples from given spectra for such domains, requires high diligence, as the statistical normalization and the field's Hermitianity are broken when using naive standard approaches. In Sec. 4.4.1, and especially in Sec. 4.4.1.7 and Sec. 4.4.1.8 it is discussed how NIFTy 3 tackles those challenges.

4.3.2 Scalability & Parallelizability

Signal reconstruction problems easily reach sizes where one mainly runs into two problems. First, given a reconstruction algorithm, the individual steps of this algorithm scale at best linearly ($\mathcal{O}(N_{\text{data}})$) with the size N_{data} of the data set. For example, adding 1024 MiB of array data takes twice as long as adding 512 MiB[4]. Fourier transforms scale with $\mathcal{O}(N_{\text{data}}\log(N_{\text{data}}))$ at best and explicit matrix-vector multiplications even only with $\mathcal{O}(N_{\text{data}}^2)$. Hence, with large data sets one easily reaches unacceptable computational run-times.
Second, for large problems the quantities of interest may not fit into a single computer's memory anymore. For example, for the reconstruction of the Milky Way's dust density in terms of a cubic volume, a side resolution of 2048 is a reasonable degree of refinement. This cube contains $2048^3 \approx 8.6 \cdot 10^9$ voxels. Hence, a scalar field on this cube containing 64-bit floating point numbers consumes 64 GiB, and in practice one needs to handle several field instances at the same time. Thus, even if runtime was not a problem, the reconstruction for such a field simply does not fit onto an affordable shared memory machine.
Hence, NIFTy 1, which does not provide parallelization to the core, reaches its limits for large reconstruction problems. In Sec. 4.4.6 it is described how NIFTy 3 makes use of the software packages D2O (Steininger et al., 2016) to achieve scalability and parallelizability, and *keepers*[5] for cluster compatibility.

4.3.3 Refactoring the Code Structure

In addition to the issues addressed in the previous Sec. 4.3.1 and Sec. 4.3.2, a whole set of further modifications was due, to ensure that the NIFTy framework and the codes being built with it remain robust and modular in the future.

4.3.3.1 Energy Functionals

In almost all realistic applications an energy functional, for example the information Hamiltonian,

$$\mathcal{H}(d, s) = -\ln \mathcal{P}(d, s) \tag{4.18}$$

must be minimized to reconstruct a sought-after signal. See Oppermann et al. (2015), Selig and Enßlin (2015), or Greiner et al. (2016) as examples. When minimizing the information Hamiltonian in the context of IFT, one has the advantage that its analytic form is known. Hence, in contrast to most common minimization settings, the gradient, curvature and any other derivative of the energy functional are directly accessible and can be used by appropriate minimizers[6]. Because of this structure, evaluating the energy functional at a certain location naturally means to consecutively calculate the functional's derivatives with decreasing degree and reusing the pre-

[4] For very small data arrays, a doubled array size can result in less than the doubled time because of constant-time overheads.

[5] `https://gitlab.mpcdf.mpg.de/ift/keepers`

[6] The gradient can be used directly in gradient-based methods like *steepest-descent* or *LBFGS* (Byrd et al., 1995, Liu and Nocedal, 1989); additionally, the true curvature can be used in Newton minimization schemes.

vious information.
As an example, for a simple Wiener filter (Wiener, 1949) this looks as follows. Given a linear measurement of a signal $s \hookleftarrow \mathcal{G}(s, S)$ with an instrument having a response R and additive noise $n \hookleftarrow \mathcal{G}(n, N)$, the data equation reads

$$d = R(s) + n \tag{4.19}$$

(cf. Eq. (4.6)). The maximum-a-posteriori solution given by a Wiener filter is represented by the minimum of the following information Hamiltonian

$$\mathcal{H}(d, s) = \frac{1}{2} s^\dagger D^{-1} s - j^\dagger s + \text{const}, \tag{4.20}$$

where

$$D = (S^{-1} + R^\dagger N^{-1} R)^{-1} \qquad \text{and} \qquad j = R^\dagger N^{-1} d. \tag{4.21}$$

When evaluating $\mathcal{H}(x)$, one computes the components of its derivatives along the way, too:

$$\partial_{s_x} \partial_{s_y} \mathcal{H}(d, s) = (D^{-1})_{xy} \qquad \text{and} \qquad \partial_{s_x} \mathcal{H} = D^{-1} s - j \tag{4.22}$$

Hence, for computation efficiency it is crucial to reuse those very building blocks to avoid duplicate calculations. A minimizer, for example, will start with evaluating the functional's value at the given current position in parameter space. *Afterwards*, it will need the gradient and (depending on the algorithm) maybe the curvature. When storing the right portions of information, those queries can be answered immediately, speeding up the minimization process as a whole.
So far NIFTy was missing structure which supported the efficient implementation of information Hamiltonians. This changed with NIFTy 3, cf. Sec. 4.4.5.

4.3.3.2 Modularity of Minimizers and Probers

Real-life information Hamiltonians are hard to minimize since they may be ill-conditioned, multi-modal, non-convex and exhibit plateaus. This makes it all the more important to freely choose a minimization algorithm and maybe even sequentially combine several of them. Sec. 4.4.5 shows how generality and modularity have been improved for NIFTy 3.
As described in Sec. 4.2.6, probing is necessary to infer quantities like the trace or diagonal of an arbitrary implicitly represented operator. Since probing an operator can become the most costly part of an inference algorithm, it is crucial that this task is carried out wisely. In Sec. 4.2.6 an overview is given how NIFTy 3 supports the user in this respect.

4.3.3.3 Meta-Information on Power Spectra

Power spectra are crucial quantities in most inferences that are built with NIFTy, as they encode the statistical information which is used to overcome the limited and noisy nature of measurement data. However, in NIFTy 1 power spectra are simply stored as un-augmented arrays, putting the burden on the users to manage all the meta-information like band structures, links to the harmonic partner space, or binning information. In Sec. 4.4.1.6 it is described how NIFTy 3 uses the concept of PowerSpaces to confine all of this interrelated information in one place.

4.4 The Structure of NIFTy 3

In this section we discuss how NIFTy 3 meets the requirements stated in Sec. 4.2.

4.4.1 Domain Objects and Fields

Fields are the main data carriers in NIFTy. Beside its data array, a field instance also stores information about its data's domain. A domain object can either be a `Space` or `FieldType` instance and it contains the information and functionality a field needs to know about the data it holds. A space in NIFTy 3 defines a spatial geometry, or to be precise, it represents the finite resolution discretization of a certain continuous and compact manifold. Additionally, NIFTy 3 also provides a field-type class, which is the base class for all bundles that may be put on top of the manifold: vectors, tensors, or just lists of numbers[7]. This illustrates how the concept of spaces from NIFTy 1 has been extended to a new base class called `DomainObject`. The class tree is shown in Fig. 4.2.

A crucial feature of NIFTy 3 is that the domain of a field is a *tuple* of domain objects instead of a single space, i.e. in general the domain is the Cartesian product of spaces and field-types. Actually, the `domain` attribute of a field can contain an arbitrary amount of spaces and field types, including zero. This makes it a very general concept which naturally covers various use cases.
The conventional use case is a field which is defined over one single space, for example a two-dimensional regular grid space.

```
In [1]: f = ift.Field(domain=ift.RGSpace(shape=(4, 4)),
                      val=1))
In [2]: f.domain
Out[2]: (RGSpace(shape=(4, 4),
                 zerocenter=(False, False),
                 distances=(0.25, 0.25),
                 harmonic=False),)
In [3]: f.val
Out[3]: <distributed_data_object>
        array([[1, 1, 1, 1],
               [1, 1, 1, 1],
               [1, 1, 1, 1],
               [1, 1, 1, 1]])
```

Note that, as described above, for consistency with multi-domain fields `f.domain` returns a *tuple* of space(s) even though `f` was defined on a single space.
The second example shows a field that is defined over the Cartesian product of a HEALPix sphere (Gorski et al., 2005) and a one-dimensional regular grid space.

```
In [4]: f = ift.Field(domain=(ift.HPSpace(nside=64),
                      ift.RGSpace(shape=(128,))))
In [5]: f.shape
```

[7] The `FieldArray` class represents a collection of numbers with a certain shape, but without any further structure.

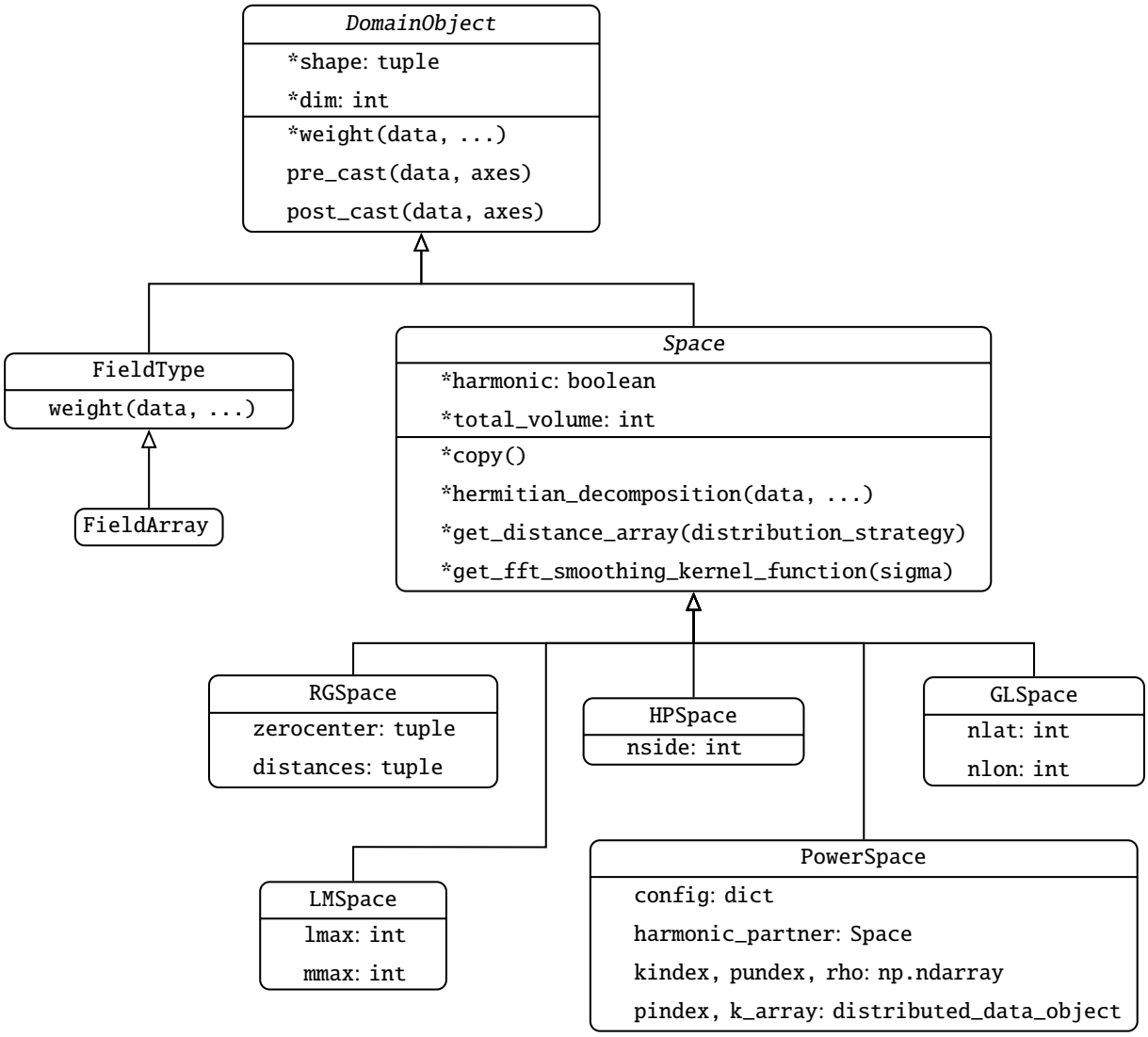


Figure 4.2: UML diagram that shows `DomainObject` and its descendants. For the derived classes only added attributes and methods are given. All superordinate attributes and methods apply implicitly to the child classes, too, since NIFTy 3 strictly obeys the Liskov substitution principle. Abstract attributes and methods are denoted with a *; Unless specified explicitly, they are implemented only by the leafs of the inheritance tree.

```
Out[5]: (49152, 128)

In [6]: f.dof
Out[6]: 6291456
```

This can be used for a combined analysis of 128 energy bands for each pixel on a sphere. Note that the shape of `f`'s data array is composed of the shape of a single HEALPix sphere (49152) and the regular grid space. Hence, the field has 6291456 degrees of freedom in total.

The way to store data[8] in NIFTy 3 is to define a field which lives on *no* geometric space at all.

```
In [7]: f = ift.Field(domain=ift.FieldArray((128,)))

In [8]: f.shape
Out[8]: (128,)
```

The domain concept generalizes so far that one can even define a field whose domain is empty:

```
In [9]: f = ift.Field(domain=(), val=123.)

In [10]: f.shape
Out[10]: ()

In [11]: f.val
Out[11]: <distributed_data_object>
         array(123.)
```

This represents a single scalar, without any further information; neither geometrical nor structural.

4.4.1.1 Separation of Responsibilities

In contrast to NIFTy 1, domain objects in NIFTy 3 only contain information and functionality regarding the geometry they define, e.g., grid distances and pixel volumes. Actually, in NIFTy 1 the field class was conceptually agnostic regarding the data object it possessed, as all numerical work was done by the space instances. In NIFTy 3 fields are defined on a tuple of domain-objects and therefore numerical operations affect the structure as a whole. Hence, the responsibility for numerical operations had to be elevated from the space objects. To be precise, this means that the responsibility for, e.g., arithmetics, power spectrum analysis, and random sample generation was moved into the field class. Functionality for, e.g., harmonic transforms, smoothing, and plotting was realized in terms of new dedicated operators. This change was an important step to better obey the *single responsibility principle* (Martin, 2003), which is why the code is now much more modular, stable and extendable.

The NIFTy standard-library contains the following spaces.

4.4.1.2 RGSpace

The `RGSpace` (**R**egular **G**rid **Space**) represents an n-dimensional toroidal manifold, i.e. the Cartesian product of n circles ($S^1 \times ... \times S^1$). Since the grid is regular, each pixel has the same volume weight, which in turn is given by the pixels' edge lengths, i.e. the `distances` attribute. The harmonic partner space of an RGSpace is again an RGSpace with identical shape but different grid lengths. Actually, the harmonic space's grid is the reciprocal lattice of the signal space's

[8] Here, *data* stands for the manifold-free result of a measurement in the sense of Eq. (4.1): $d = f(s, n)$.

grid. Hence, the pixels' edge lengths in the harmonic grid are given by the inverse of the longest distances in the signal space grid[9].

```
In [12]: signal_space = ift.RGSpace(shape=(4, 5), distances=(0.8,
    0.4))
In [13]: fft = ift.FFTOperator(signal_space)
In [14]: f = ift.Field(signal_space, val=np.arange(20.))
In [15]: g = fft(f)
In [16]: g.domain
Out[16]: (RGSpace(shape=(4, 5),
                  zerocenter=(False, False),
                  distances=(0.3125, 0.5),
                  harmonic=True),)
In [17]: f.norm()
Out[17]: 28.114053425288926

In [18]: g.norm()
Out[18]: 28.114053425288926
```

Note that the scalar product and therefore the norm remains invariant, irrespective of whether the field was represented in the signal or the harmonic basis.
Additionally, the `RGSpace` class has the degree of freedom, whether the data should be stored *zerocentered* or not. If the `zerocenter` attribute is set to true for a certain axis, the origin is put at the center of the data array. This means that in case of a one-dimensional zero-centered harmonic space with 6 pixels the values for k are placed like:

$$\{k_{-3}, k_{-2}, k_{-1}, k_0, k_1, k_2\} \quad \text{for} \quad k_i \quad \text{with} \quad i \in [-3, ..., 2] \tag{4.23}$$

In contrast, non-zero-centered ordering yields

$$\{k_0, k_1, k_2, k_{-3}, k_{-2}, k_{-1}\} \quad \text{for} \quad k_i \quad \text{with} \quad i \in [-3, ..., 2] \tag{4.24}$$

Albeit being less convenient when looking at a spectrum by eye, fft libraries like *FFTW* (Frigo and Johnson, 2005) usually follow the non-zero-centered convention. NIFTy 3 can handle both conventions and performs the mediation with respect to those external libraries.

4.4.1.3 HPSpace

The `HPSpace` (**H**EAL**P**ix **Space** (Gorski et al., 2005)) represents a unit 2-sphere. The HEALPix pixelation subdivides the surface of a sphere into pixels of equal area. Those pixels reside on iso-latitude circles, with equal spacing on each circle. In NIFTy the represented sphere has a fixed radius of 1. Hence, the only remaining parameter for `HPSpace` is the HEALPix resolution parameter `nside`. The harmonic partner space for `HPSpace` is the `LMSpace`.

[9] Since the `RGSpace` represents an n-dimensional torus, the longest distances are given by the circumferences of the individual circles the torus is made of.

4.4.1.4 GLSpace

Similar to the `HPSpace`, the GLSpace (**G**auss-**L**egendre **Space**) is a discretization of the unit 2-sphere. It consists of n_{lat} iso-latitude rings containing n_{lon} pixels each. The pixels are equidistant in azimuth on every ring, and the ring latitudes coincide with the roots of the Legendre polynomial of degree n_{lat}. Within each ring the pixel weights are identical. However, the individual weights for the different rings are chosen such that a numerical integration is a Gauss quadrature that is exact for all spherical harmonics Y_{lm} with $l < n_{\text{lat}}$ and $|m| \leq (n_{\text{lon}} - 1)/2$. The harmonic partner of a `GLSpace` is the `LMSpace`.

4.4.1.5 LMSpace

The `LMSpace` is the harmonic partner domain for a pixelization of the unit 2-sphere. Its name is derived from the typically employed indices of the spherical harmonics Y_{lm}. A `LMSpace` instance holds a value for every Y_{lm} with $-l \leq m \leq l$ and $0 \leq l \leq l_{\text{max}}$ where l_{max} is the cutoff frequency. It can be a harmonic partner to either the HPSpace or the GLSpace. There is a unique projection from LMSpace to HPSpace and to GLSpace, respectively. However, this projection is not necessarily invertible; see Sec. 4.4.2.1 for details. The volume weight is 1 for each of the pixels in `LMSpace`.

4.4.1.6 PowerSpace

The `PowerSpace` is used to condense the k-modes of a harmonic space into a one-dimensional space, the discrete isotropic power spectrum. This is a natural step when analyzing statistically homogeneous and isotropic fields, and the `PowerSpace` class provides a rich functionality for this kind of analysis (cf. Sec. 4.4.1.7). The `PowerSpace` condenses the information of a harmonic space of arbitrary dimension into a one-dimensional space by defining bins into which the different k-modes are aggregated. By this, it enables data scientists to write algorithms that are independent of the underlying space. The `PowerSpace` keeps information about the relation to its harmonic partner space for both numerical efficiency and enabling field synthetization (cf. Sec. 4.4.1.8).
Note that depending on the application there is an ambiguity to what the correct volume weighting for this space should be, cf. Sec. 4.2.4. The first weighting is induced by the harmonic partner space and uses the multiplicity ρ of the modes in each bin as weights. This weighting is highly dependent on the structure of the harmonic partner space. The second weighting is induced by regarding the power space as a one-dimensional space on a (potentially) irregular grid, which induces a volume factor equal to the bin size. The third weighting is obtained analogously to the second but differs by using a logarithmic axis $\log(k)$. It is used when calculating smoothness on a log-log scale, which is required when a prior that favors power law spectra is wanted (cf. Weig and Enßlin 2010). Finally, there is a trivial fourth weighting where the volume factors are constantly set to one. This is, for example, needed in the critical filter (cf. Enßlin and Frommert 2011).

4.4.1.7 Power Spectrum Analysis

In this section we describe how the one-dimensional power spectrum for a given field is computed. As mentioned in Sec. 4.2.5, this one-dimensional quantity fully describes the correlation structure of statistically homogeneous and isotropic fields. The reason for this is that for such fields the statistical properties in the harmonic space solely depend on the distance to the origin, but not the direction.
The following example is in order: Let `s` be the discretization of a two-dimensional Cartesian space and `f` a field on `s`:

```
In [19]: s = ift.RGSpace(shape=(4,4), zerocenter=True)
In [20]: s
Out[20]: RGSpace(shape=(4, 4),
                 zerocenter=(True, True),
                 distances=(0.25, 0.25),
                 harmonic=False)
In [21]: f = ift.Field(domain=s, val=1.)
```

`s` has a total volume of 1 by default, which is the reason why the edge length of `s`'s pixels is equal to 0.25. To compute its power spectrum, `f` first must be transformed to the harmonic basis.

```
In [22]: fft = ift.FFTOperator(s)
In [23]: f_h = fft(f)
In [24]: s_h = f_h.domain[0]
In [25]: s_h
Out[25]: RGSpace(shape=(4, 4),
                 zerocenter=(True, True),
                 distances=(1.0, 1.0),
                 harmonic=True)
```

Now, the power spectrum can be computed by:

```
In [26]: f_p = f_h.power_analyze()
```

Note that the domain of `f` is a `PowerSpace` instance, viz. a one-dimensional irregularly gridded space with 6 pixels:

```
In [27]: s_p = f_p.domain[0]
In [28]: s_p
Out[28]: PowerSpace(
    harmonic_partner=RGSpace(shape=(4, 4),
                             zerocenter=(True, True),
                             distances=(1.0, 1.0),
                             harmonic=True),
    distribution_strategy='not',
    logarithmic=False,
    nbin=None,
    binbounds=None)
In [29]: f_p.shape
Out[29]: (6,)
```

Let's discuss what happened during the `f_h.power_analyze()` call. Recall that `f_h` is defined on a 4×4 grid. First of all, `s_h` is asked for the distances with respect to the center:

```
In [30]: s_h.get_distance_array('not')
Out[30]:
<distributed_data_object>
array([[ 2.82,  2.23,  2. ,  2.23],
       [ 2.23,  1.41,  1. ,  1.41],
       [ 2.  ,  1.  ,  0. ,  1.  ],
       [ 2.23,  1.41,  1. ,  1.41]])
```

There are several pixels which have the same distance to the center. Those pixels together constitute a bin; hence, there are six bins in total. A pixel's bin-affiliation is stored in the so-called `pindex` array:

```
In [31]: s_p.pindex
Out[31]: <distributed_data_object>
         array([[5, 4, 3, 4],
                [4, 2, 1, 2],
                [3, 1, 0, 1],
                [4, 2, 1, 2]])
```

The multiplicity of each bin is given by `rho`; its distance to the grid center by `kindex`:

```
In [32]: s_p.rho
Out[32]: array([1, 4, 4, 2, 4, 1])
In [33]: s_p.kindex
Out[33]:
array([0., 1., 1.41, 2., 2.24, 2.83])
```

Concretely, calculating the power spectrum of `f_h` now means accumulating the absolute squared of `f_h`'s pixel values in their respective bins and divide by `rho` to form the average. Assuming the artificial case that `f_h` has the following values

```
In [34]: f_h.val
Out[34]:
<distributed_data_object>
array([[-1.+3.j, 4.+5.j, 2.+4.j, 0.+2.j],
       [-3.+6.j,-3.+1.j,-4.+2.j, 3.+3.j],
       [ 1.+4.j, 2.+7.j,-4.+1.j, 4.+5.j],
       [ 3.+5.j,-2.+0.j,-3.+6.j,-3.+1.j]])
```

this results in

```
In [35]: f_p = f_h.power_analyze()
In [36]: f_p.val
Out[36]:
<distributed_data_object>
array([ 17., 39.75, 10.5, 18.5, 31., 10.])
```

Note that the power spectra we have discussed so far are always real. However, if a field's domain is the Cartesian product of multiple spaces, it can be necessary to keep track of the real and imaginary part of `f` in the position basis separately. In NIFTy 3, this can be controlled with the `keep_phase_information` keyword in `power_analyze`. If it is set to true, the resulting power spectrum will be complex. Its real (imaginary) part is constructed from the real (imaginary) part of `f`. To avoid harmonic transforms whenever possible, the decomposition is done using Hermitian symmetry constraints, see Sec. 4.4.1.9 for details.

4.4.1.8 Power Spectrum Synthesis

It is very important to be able to create random samples for a given power spectrum. For a field `f_p` in NIFTy 3, this can be done using the method `f_p.power_synthesize()`. This routine draws a Gaussian random field with zero mean and unit variance in the harmonic partner space of the power space. Afterwards, the modes are weighted according to the power spectrum that is stored in `f_p`.
Note that it is non-trivial to create the Gaussian random field sample in case of real-valued signal-space fields. This is because the result of Fourier-transforming a real-valued signal space field (`f`) has a point symmetry with respect to the origin in the harmonic basis. This becomes particularly exigent if one synthesizes a random field from a multi-dimensional power spectrum which is associated with the Cartesian product of two or more spaces. In this case one needs a well-engineered approach for handling the aforementioned field's symmetry. This so-called *Hermitian decomposition* is discussed in the next section.

4.4.1.9 Hermitian Decomposition

Hermitian decomposition is needed for both, power spectrum analysis and synthesis. Hence, in the following its rationale is discussed.
The result of a harmonic transform of a real-valued field is in general complex-valued.[10] However, the degrees of freedom are the same for both representations, since – as noticed in the previous section – in the harmonic basis the field exhibits a point symmetry modulo complex conjugation. Therefore, it is called *Hermitian* symmetry. Because of the complex conjugation, the imaginary part of the reflection's fixed points vanishes. For power spectrum analysis we need the functionality to decompose a field into its Hermitian and anti-Hermitian part. In addition, for power-spectrum synthesis of real fields, one needs normally distributed Hermitian random samples.
In the following example the harmonic representation of a real-valued random field is shown. The fixed points of the Hermitian symmetry, where only real numbers should appear, are marked red.

```
In [37]: s = ift.RGSpace((4, 4))
In [38]: f = ift.Field(s, val=np.random.random((4, 4)))
```

[10] This is true if the customary transformation kernels are used; e.g. for flat manifolds this is $e^{-2\pi ikx}$. However, in principle also sine and cosine kernels are applicable, where the real/imaginary part of the field in position space gets mapped onto the real/imaginary part of the field in harmonic space, respectively.

```
In [39]: f *= 10.
In [40]: f.val
Out[40]: <distributed_data_object>
         array([[0.7, 9.5, 9.7, 8.1],
                [0.3, 1.0, 6.8, 4.4],
                [1.2, 5.0, 0.3, 9.1],
                [2.6, 6.6, 3.1, 5.2]])
In [41]: fft = ift.FFTOperator(s)
In [42]: fft(f).val
<distributed_data_object>
array(
[[4.8+0.j  ,-0.8+0.3j,-1.3+0.j  ,-0.8-0.3j],
 [0.8+0.1j,-0.3-0.1j, 0.3-0.7j,-0.9+0.6j],
 [0.7+0.j  ,-0.2+0.j ,-1.1+0.j  ,-0.2-0.j ],
 [0.8-0.1j,-0.9-0.6j, 0.3+0.7j,-0.3+0.1j]])
```

Please note that, for example, the entry at position $(2, 4)$ with value $(-0.9 + 0.6i)$ is the complex conjugated partner of position $(4, 2)$ with value $(-0.9 - 0.6i)$.
As mentioned in Sec. 4.4.1.7, for power-spectrum analysis it can be necessary to decompose a complex-valued position-space field into its real and imaginary part. However, for signal inference often the native basis is a field's harmonic space. In this case, one certainly could transform the field from its harmonic into its position space basis, split real and imaginary parts there, and, finally, transform back. Nevertheless, one ought to avoid harmonic transforms whenever possible, since they are at least either computationally expensive, or even inherently inexact, cf. Sec. 4.4.2.1. Luckily, as seen in the example above, the real part of a position-space field corresponds to the Hermitian symmetric part of the harmonic-space partner field. Since harmonic transforms are linear operations, the imaginary part corresponds to the anti-symmetric part of the field, analogously. Hence, splitting a position-space field into real and imaginary part is equivalent to splitting the harmonic-space field into its Hermitian and anti-Hermitian part.
For synthetization of fields from power spectra, as discussed in Sec. 4.4.1.8, it is necessary to create a normal random field that lives in harmonic space and has zero mean and unit variance. If the position space field shall be real, there are two approaches that correspond to those of the former paragraph. First, a real-valued sample can be drawn in position space which is then transformed into the harmonic basis. For the reasons set out above, this approach is sub-optimal. Second, a random sample can be created in harmonic space directly, from which the anti-Hermitian part is removed. By this, again, any harmonic transforms are avoided. However, particular diligence is needed in order to preserve the correct variances.
In the following it is discussed how a field can be decomposed into its (anti-)Hermitian parts. A straightforward approach is to use a reflection or flipping operator γ which performs a point reflection with respect to a space's center. Therefore for a given field f, the Hermitian and anti-

Hermitian parts f_h and f_a, respectively, are given by the half-sum and half-difference

$$f_h = \frac{1}{2}\left(f + \gamma(f)^*\right) \tag{4.25}$$

$$f_a = \frac{1}{2}\left(f - \gamma(f)^*\right), \tag{4.26}$$

where

$$f = f_h + f_a. \tag{4.27}$$

This works perfectly fine for individual spaces and also generalizes for the Cartesian products of spaces. Assuming a field f whose domain is (s_1, s_2), the individual flips must be applied consecutively before applying the complex conjugation.

$$f_h = \frac{1}{2}\left(f + \gamma_1(\gamma_2(f))^*\right) \tag{4.28}$$

$$f_a = \frac{1}{2}\left(f - \gamma_1(\gamma_2(f))^*\right) \tag{4.29}$$

Correcting the variance When drawing samples for a given power spectrum it is necessary to create random samples with zero mean and unit variance. However, when forming the half-sums and -differences of a Gaussian random field with the mirrored version of itself, its variance must be corrected by a factor of $\sqrt{2}$[11]. However, a priori the correction only applies to those pixels that got mapped to different pixels via the flipping. The pixels which the flip maps onto themselves, i.e. the fixed points of the flip, are averaged with themselves and therefore do not need a variance correction factor. But once the sample is complex valued, the involved complex conjugation will eliminate the fixed point's imaginary part in case of the Hermitian portion. To compensate this loss of power, in this case the fixed points after all need a correction factor of $\sqrt{2}$, too.

4.4.2 Linear Operators

Up to now we have introduced fields as well as their domain-objects. In the following paragraph we will show how linear and implicitly defined operators act on fields and how they are implemented.

Every operator in NIFTy is obliged to be linear and therefore inherits from the abstract base class `LinearOperator`. This base class provides the user with a blueprint for any elaborated operator. A generic operator A is defined by its input and target domains, the boolean whether it is unitary or not, and its actual action. With these properties at hand, the application of A on a given field s, viz. $A(s)$, is split into a generic and a specific part. The generic part involves consistency checks, e.g. whether the domain of s and A match, while the specific part is the individual operator's action. More specifically, the application of $A(s)$ is implemented as `A(s)` or equivalently as `A.times(s)`. Here, `times` basically first calls the internal method `_check_input_compatibility`, which performs the domain compatibility check for s and A.

[11] Recap that the sum of two Gaussian distributions is again a Gaussian distribution.

NIFTy Operator	description
↪ `ComposedOperator`	represents a container of `LinearOperators` which are applied in sequence.
↪ `EndomorphicOperator`	represents all operators with equal domain and target.
↪ `DiagonalOperator`	represents a diagonal matrix in specified domain.
↪ `LaplaceOperator`	represents the discrete second derivative for 1D spaces.
↪ `ProjectionOperator`	projects the input onto a NIFTy field.
↪ `SmoothingOperator`	convolves the input with a Gaussian kernel.
↪ `SmoothnessOperator`	measures the smoothness of the input by applying the LaplaceOperator twice.
↪ `FFTOperator`	implementation of harmonic transforms of fields between different domains.
↪ `ResponseOperator`	represents an exemplary response including convolutional smoothing and exposure.

Table 4.1: Overview of provided Operators, inherited from `LinearOperator`

This is followed by the implementation of the operator's action in `_times`.
In case s is defined over multiple domain-objects but A is not, and A is supposed to act on a specific domain-object of s, one pins the action of A to a certain space of s's domain tuple. This is done by passing the information along with `times`, e.g. `A.times(s, spaces=0)`, or by using the `default_spaces` keyword argument during the operator's initialization. The output of $A(s)$ may live on a different domain, depending on the target of A.
As one often needs more than the pure forward application of an operator on a field, i.e. `.times`, NIFTy allows the user to further implement `_adjoint_times`, `_inverse_times`, and if needed also `_adjoint_inverse_times`. In case an operator is unitary, one only needs to implement `_inverse_times` or `_adjoint_times`, as NIFTy 3 does a mapping of methods internally.
NIFTy 3 provides the user with a set of often reappearing operators. An overview can be found in Tab. 4.1 and Fig. 4.3.
In the following, two especially mentionable operators will be described: the `FFTOperator` and `SmoothingOperator`.

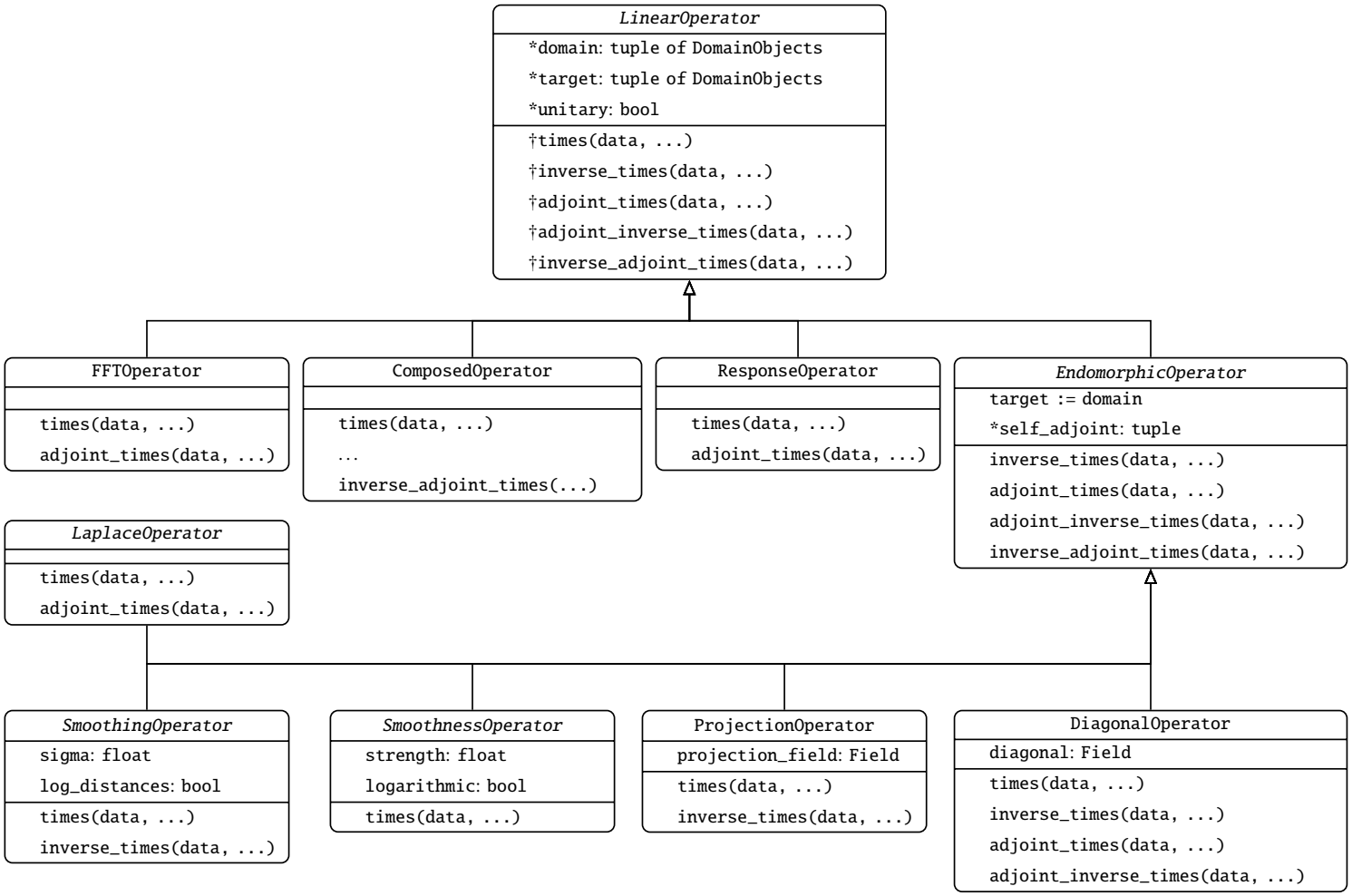


Figure 4.3: UML diagram that shows the inheritance structure for the NIFTy `Operator`s. For the derived classes only added attributes and methods are given. All superordinate attributes and methods apply implicitly to the child classes, too, since NIFTy 3 strictly obeys the Liskov substitution principle. Abstract attributes and methods are denoted with a *; Unless specified explicitly, they are implemented only by the leafs of the inheritance tree. For the `LinearOperator` class, the † signals, that the methods are defined, though not implemented.

4.4.2.1 FFTOperator

The purpose of this operator is the conversion of data from the spatial domain to the frequency domain and vice versa. For data living on equidistant Cartesian grids (i.e. RGSpaces) this is more or less straightforward to do: every configuration in position or frequency domain has a unique corresponding configuration in the other domain. The conversion between both sides is done efficiently – and without information loss – using an FFT algorithm.
For data living on the 2-sphere, things are more complicated: first of all, it is not possible to define pairs of equivalent spaces in the sense that data conversion is lossless in both directions for arbitrary maps. However, for any given LMSpace L a GLSpace G can be found, such that any data set living on L can be converted to G and back without losing information[12]. In contrast, it is not possible to find an LMSpace that can hold the harmonic equivalent of all possible data sets living on a GLSpace. Hence, the set of *band-limited* functions on the sphere, which can be transformed without loss in both directions, is only a tiny subset of all possible functions that can live on a GLSpace grid. When solving problems on the sphere, it is therefore advantageous to formulate them primarily in harmonic space.
For the HEALPix discretization of the sphere, things are somewhat worse still: since no analytic quadrature rule for this set of pixels exists, any transform from LMSpace to HPSpace and back will only produce an approximation of the input data. In many situations this is acceptable given the other advantages of this pixelization, like equal-area pixels. But this imperfection must be kept in mind when choosing pixelizations on which to solve a problem.
In NIFTy 1 harmonic transformations are performed using a method of the particular field instance `f`, viz. `f.transform()`. This is a source of confusion and suggests general invertibility of the harmonic transform, since one could naively assume that `f.transform().transform()` is equal to `f` – which is, incidentally, actually true for RGSpaces. In NIFTy 3 the `FFTOperator` solely implements the `times` and `adjoint_times` methods. The `inverse_times` is available if and only if a certain transform path is explicitly marked to be unitary. This structure makes it very apparent when there is the risk of information loss.

4.4.2.2 Smoothing Operator

This operator convolves a field with a Gaussian smoothing kernel. The preferred approach is to apply the convolution in position space as a point-wise multiplication in harmonic space if possible. The `PowerSpace` is in general non-regularly gridded, which makes harmonic transformations hard, albeit not impossible. But it is not periodic either, which is why convolutional smoothing cannot be applied to it. For this case, the `SmoothingOperator` will perform its smoothing explicitly in position space.

4.4.2.3 Harmonic Smoothing Operator

This operator exploits the fact that a convolution in the spatial domain is equivalent to multiplication in the frequency domain. Hence, the input field is first transformed using the `FFTOperator`.

[12] Numerical noise caused by the finite precision of floating-point algebra can slightly break invertibility, though.

The Gaussian kernel to be multiplied is generated with a standard deviation σ, given in the units of the domain's geometry. The result after multiplying the transformed input data is reverted to the position domain by calling `FFTOperator`'s `adjoint_times` method. Thus, the caveats pertaining to transformations mentioned in the Sec. 4.4.2.1 for `FFTOperator` apply here as well. Convolutional smoothing is available for all position spaces, i.e. `RGSpace`, `GLSpace` and `HPSpace`. Internally, a method called `get_fft_smoothing_kernel_function` is needed for the respective harmonic partner domain; it returns a function which is used to construct the kernel and is therefore defined on `RGSpace` and `LMSpace`. Note that the correct Gaussian kernel function for smoothing on the sphere is given by

$$K_{\text{sphere}}(l) = \exp(-\frac{1}{2}l(l+1)\sigma^2) \tag{4.30}$$

instead of the standard frequency response of a Gaussian filter

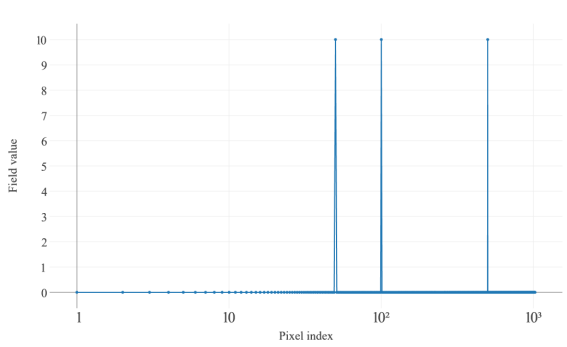
$$K(k) = \exp(-2(\pi k\sigma)^2), \tag{4.31}$$

since one has to take into account the sphere's curvature (cf. Challinor et al. 2000).

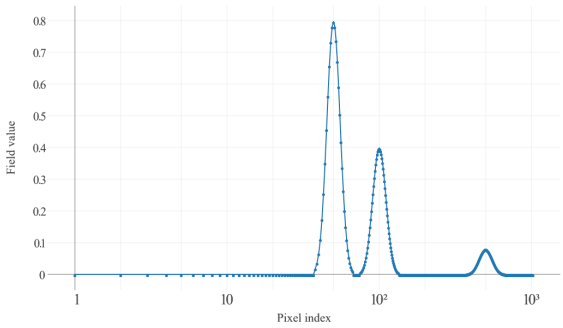
4.4.2.4 Direct Smoothing Operator

The main motivation behind this operator is the aforementioned `PowerSpace` from Sec. 4.4.1.6, which is based on a non-regular grid. Since it is not periodic, this approach does not do a transformation into harmonic space but applies smoothing directly. A point-by-point convolution is done on the space with a 1-dimensional Gaussian kernel with standard deviation σ.
In a normal case of convolution with a kernel of window length w (assume as *odd* for centered symmetric kernels) the result is shorter than the original data by $w-1$ points, losing $(w-1)/2$ points on each end of the data vector. Since this behavior is unsuitable as a solution for NIFTy, the convolution at the beginning and end of the array, is carried out with partial Gaussian kernels. Although this method is less accurate at the ends of the data, it does not sacrifice information resolution at each smoothing. Also note that, since the `SmoothingOperator` must conserve the sum of all field entries, on a non-regular grid the exact smoothing behavior depends on the location within the grid. The denser a certain pixel's neighborhood, the larger the number of weight receptors and therefore the lower the resulting Gaussian bell, when applying the `SmoothingOperator` on single peaks. An illustration for this behavior is given in Fig. 4.4. There, the result of smoothing a constant field with three dedicated peaks is shown. Since the smoothing happens on a logarithmic scale, the peak's weight is distributed to a different number of pixels for each peak. The code that was used to produce the plots in Fig. 4.4 is given by

```
import nifty as ift
power_space = ift.PowerSpace(ift.RGSpace((2048),
                                         harmonic=True))
f = ift.Field(power_space, val=1.)
f.val[[50, 100, 500]] = 10
sm = ift.SmoothingOperator(power_space,
```



(a) Unsmoothed field with three peaks.



(b) Smoothed field with different height for each peak.

Figure 4.4: Illustration of the behavior of smoothing an input field (a) on a non-regular grid or on a non-regular scale, respectively, which yields (b). Here, the field was smoothed on a logarithmic scale.

```
                              sigma=0.1,
                              log_distances=True)
plotter = ift.plotting.PowerPlotter()
plotter.figure.xaxis.label = 'Pixel index'
plotter.figure.yaxis.label = 'Field value'
plotter(f, path='unsmoothed.html')
plotter(sm(f), path='smoothed.html')
```

4.4.3 Operator Inversion

There are cases in which the inverse action of an operator is needed, but not analytically accessible and therefore not directly implementable. An example for this is the propagator operator in the Wiener filter (cf. Eq. (4.15)), where the quantity $D^{-1} = \left(S^{-1} + R^\dagger N^{-1} R\right)$ must be inverted. For such cases NIFTy 3 provides a robust implementation of the conjugate gradient method, following Nocedal and Wright (2006).
Additionally, NIFTy 3 supplies the user with the special class `InvertibleOperatorMixin`. In conjunction with multiple inheritance, this mixin-class (cf. Lutz 2010, p. 599), can be used to equip a custom operator with missing `*_times` methods. For example, if solely the `inverse_times` of a custom operator is implemented, the mixin-class will provide the `times` via conjugate gradient.

4.4.4 Probing

NIFTy 1 already provided a class for probing the trace and diagonal of an implicitly defined operator A by evaluating the following expressions:

$$\mathrm{tr}[A] \approx \left\langle \xi^\dagger A \xi \right\rangle_{\{\xi\}} = \sum_{pq} A_{pq} \left\langle \xi_p \xi_q \right\rangle_{\{\xi\}} \to \sum_p A_{pp}, \tag{4.32}$$

$$\Big(\mathrm{diag}[A]\Big)_p \approx \Big(\langle \xi * A\xi \rangle_{\{\xi\}}\Big)_p = \sum_q A_{pq} \left\langle \xi_p \xi_q \right\rangle_{\{\xi\}} \to A_{pp}, \tag{4.33}$$

where $\langle \cdot \rangle_{\{\xi\}}$ is the sample average of fields ξ which have the property $\left\langle \xi_p \xi_q \right\rangle_{\{\xi\}} \to \delta_{pq}$ for $\#\{\xi\} \to \infty$, and $*$ denotes component-wise multiplication (Selig et al., 2013, sec. 3.4). However, especially if the operator evaluation involves a conjugate gradient, the probing can account for the majority of an algorithm's computational costs, which is why several heuristics turned out to be useful in practice. Namely, in iterative schemes, it can be beneficial to reuse some or even all probes $\{\xi\}$ and/or utilize the former results as starting points for the conjugate gradient runs. To naturally allow for such heuristics, the prober class has been completely rewritten for NIFTy 3: now it exhibits special call-hooks for customization.

4.4.5 Energy Object & Minimization

Inferring posterior parameters typically requires the minimization of some energy functional, such as the Information Hamiltonian or a Kullback-Leibler divergence. The `Energy` class provides the structure required for efficient implementation of such functionals. Its interface provides the current value, its gradient and curvature at its position. To avoid multiple evaluations of the same quantity, intermediate results can be memorized and reused later. To ensure consistency when reusing intermediate results, an `Energy` class instance in NIFTy 3 is fixed to one certain position. Setting a new position therefore creates a new instance of the class. `Energy` instances are passed to minimizers, which perform the actual numerical minimization. The position of the passed instance is used as the starting point of the minimization.
NIFTy 3 provides implementations for steepest descent, VL-BFGS (Chen et al., 2014) and a relaxed Newton scheme. Those methods differ in how the descent direction is determined. Steepest descent just follows the downhill gradient. VL-BFGS incorporates prior steps to estimate the curvature of the energy landscape, suggesting an improved direction. The relaxed Newton minimizer makes use of the full local curvature, additionally providing an estimate of the step size which is optimal in a quadratic potential. After calculating the descent direction, a line search along this very direction is done to determine the – with respect to the the Wolfe conditions – optimal step size. This is iterated until the convergence criterion is satisfied or the maximum number of steps has been reached.

4.4.6 Parallelization & Cluster Compatibility

The software package D2O (Steininger et al., 2016) was originally developed for massively parallelizing NIFTy by distributing the fields' data arrays among multiple nodes in a MPI cluster. If

two fields are distributed with the same distribution strategy individual nodes hold the same part of each of the fields' arrays. With this ansatz, numerical operations like adding two fields involve no communication and hence exhibit excellent scaling behavior. Please refer to Steininger et al. (2016) for an exhaustive discussion. As a consequence, in contrast to NIFTy 1 the arrays in NIFTy 3 are `distributed_data_objects`, rather than `numpy.ndarrays`. Due to this strong encapsulation, most of its code base and the work with NIFTy 3 is agnostic of parallelization. By utilizing D2O, NIFTy can operate on high performance computing clusters and thus terabytes of RAM. This renders a large quantity of new applications possible, like high-resolution runs of 3D reconstructions, cf. (Böhm et al., 2017, Greiner et al., 2016), and Faraday synthesis algorithms (Bell and Enßlin, 2012).
Another side product of NIFTy 3 is the *keepers* package[13], which is aimed at making scientific research on clusters as convenient as possible. To be precise, this involves classes and functionality for cluster-compatible logging, convenient algorithm parametrization, storing NIFTy objects to disk in a versioned fashion and therefore allowing for restartable jobs.

4.5 Application: Wiener Filter Reconstructions

4.5.1 Case 1: Single Space Geometry

Fig. 4.5 shows an exemplary implementation of the Wiener filter with NIFTy 3. First, the parameters that characterize the mock signal's correlation structure are set up (lines 3ff.). Then, starting with line 12, the domain geometry is specified: here, a single regular gridded two-dimensional Cartesian space is used. Afterwards, in lines 19ff., a signal covariance is defined to create the mock signal that shall be reconstructed by the Wiener filter later. In line 27 an exemplary response operator is set up which acts via smoothing and masking on the input signal. After setting up a noise covariance and creating a noise sample, in line 34 the mock data is created. In line 39 the actual Wiener filter is performed by applying the propagator operator D, which coincides with the inverse of the information Hamiltonian's curvature. Once the reconstruction is done, an uncertainty map is computed (lines 40ff.). Therefore, a prober class with the desired probing-targets – here only the diagonal – is constructed via multiple inheritance. After this, a prober instance is created, which is then applied to the Wiener filter curvature operator. Because the `wiener_curvature` operates in harmonic space, but the probes must be evaluated in position space, Fourier transforms are wrapped around the operator in line 43. Please note that the reconstruction's variance is given by the bare diagonal entries, viz. no volume factors included, of the inverse curvature, which is the reason for the inverse weighting in line 45. Finally, the lines 47ff. produce the plots that are shown in Fig. 4.6.

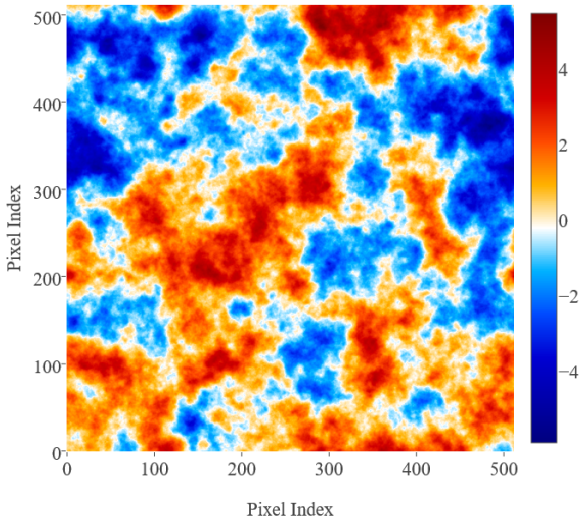
4.5.2 Case 2: Cartesian Product Space Geometry

As discussed in Sec. 4.4.1, one of the crucial features of NIFTy 3 is that fields can be defined on the Cartesian product of multiple individual spaces. This makes it possible to easily implement

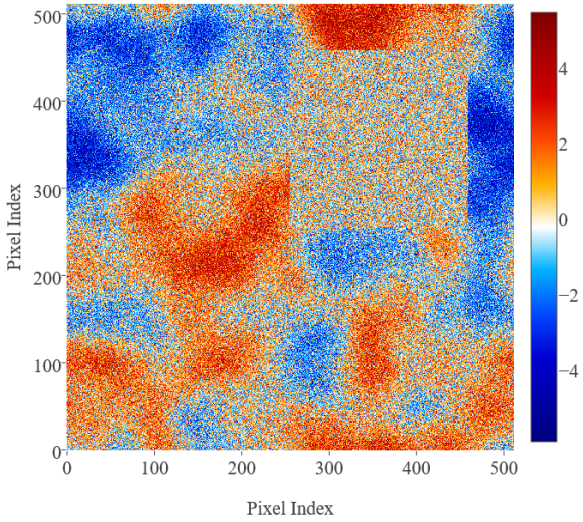
[13] `https://gitlab.mpcdf.mpg.de/ift/keepers`

```python
import nifty as ift; import numpy as np; from keepers import Repository

# Setting up parameters
correlation_length_scale = 1.  # Typical distance over which the field is correlated
fluctuation_scale = 2.         # Variance of field in position space
response_sigma = 0.05          # Smoothing length of response (in same unit as L)
signal_to_noise = 1.5          # The signal to noise ratio
np.random.seed(43)             # Fixing the random seed
def power_spectrum(k):         # Defining the power spectrum
    a = 4 * correlation_length_scale * fluctuation_scale**2
    return a / (1 + (k * correlation_length_scale)**2) ** 2
# Setting up the geometry
L = 2.  # Total side-length of the domain
N_pixels = 512  # Grid resolution (pixels per axis)
signal_space = ift.RGSpace([N_pixels, N_pixels], distances=L/N_pixels)
harmonic_space = ift.FFTOperator.get_default_codomain(signal_space)
fft = ift.FFTOperator(harmonic_space, target=signal_space, target_dtype=np.float)
power_space = ift.PowerSpace(harmonic_space)
# Creating the mock signal
S = ift.create_power_operator(harmonic_space, power_spectrum=power_spectrum)
mock_power = ift.Field(power_space, val=power_spectrum)
mock_signal = fft(mock_power.power_synthesize(real_signal=True))

# Setting up an exemplary response
mask = ift.Field(signal_space, val=1.); N10 = int(N_pixels/10)
mask.val[N10*5:N10*9, N10*5:N10*9] = 0.
R = ift.ResponseOperator(signal_space, sigma=(response_sigma,), exposure=(mask,))
data_domain = R.target[0]
R_harmonic = ift.ComposedOperator([fft, R], default_spaces=[0, 0])
# Setting up the noise covariance and drawing a random noise realization
N = ift.DiagonalOperator(data_domain, diagonal=mock_signal.var()/signal_to_noise, bare=True)
noise = ift.Field.from_random(domain=data_domain, random_type='normal',
                              std=mock_signal.std()/np.sqrt(signal_to_noise), mean=0)
data = R(mock_signal) + noise

# Wiener filter
j = R_harmonic.adjoint_times(N.inverse_times(data))
wiener_curvature = ift.library.WienerFilterCurvature(S=S, N=N, R=R_harmonic)
m_k = wiener_curvature.inverse_times(j); m = fft(m_k)
# Probing the uncertainty
class Proby(ift.DiagonalProberMixin, ift.Prober): pass
proby = Proby(signal_space, probe_count=800)
proby(lambda z: fft(wiener_curvature.inverse_times(fft.inverse_times(z))))
sm = ift.SmoothingOperator(signal_space, sigma=0.03)
variance = ift.sqrt(sm(proby.diagonal.weight(-1)))

# Plotting
plotter = ift.plotting.RG2DPlotter(color_map=plotting.colormaps.PlankCmap())
plotter.figure.xaxis = ift.plotting.Axis(label='Pixel Index')
plotter.figure.yaxis = ift.plotting.Axis(label='Pixel Index')
plotter.plot.zmax = variance.max(); plotter.plot.zmin = 0
plotter(variance, path = 'uncertainty.html')
plotter.plot.zmax = mock_signal.max(); plotter.plot.zmin = mock_signal.min()
plotter(mock_signal, path='mock_signal.html')
plotter(ift.Field(signal_space, val=data.val), path='data.html')
plotter(m, path='map.html')
```

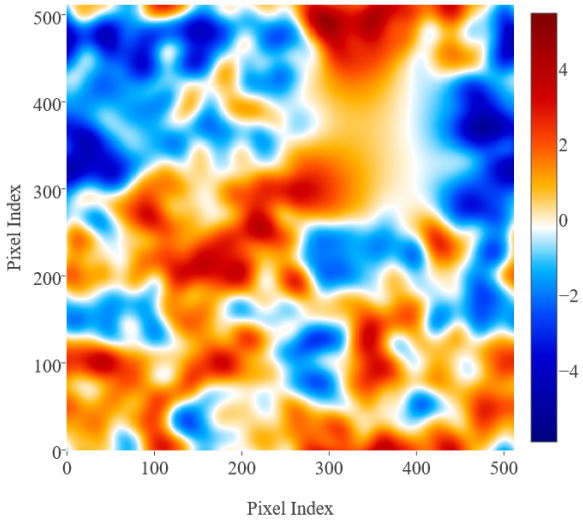
Figure 4.5: A full-feature Wiener filter implementation in NIFTy 3, including mock-data creation, signal reconstruction, uncertainty estimation, and plotting of the results.



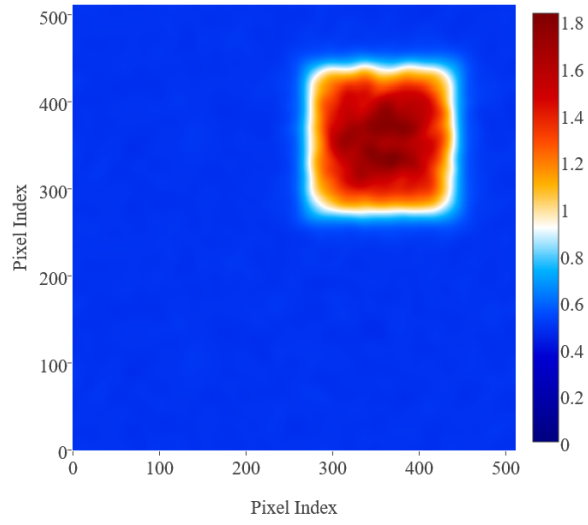
(a) Mock signal.



(b) Data, which is the result of smoothing, masking and additive noise.



(c) Maximum a-posteriori reconstruction.



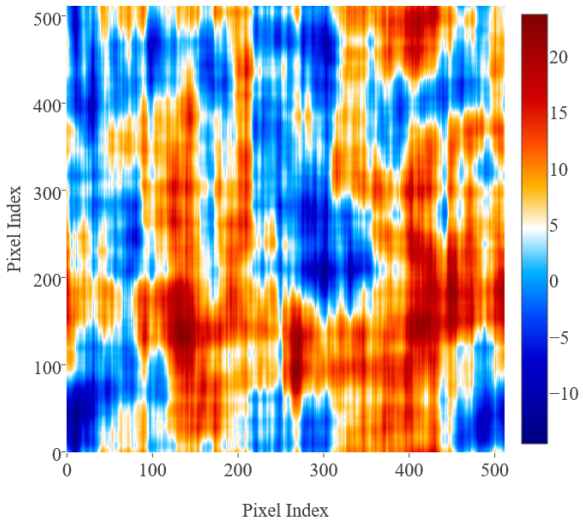
(d) Uncertainty map.

Figure 4.6: Illustration of a Wiener filter reconstruction. The labeling of the axes shows pixel numbers.

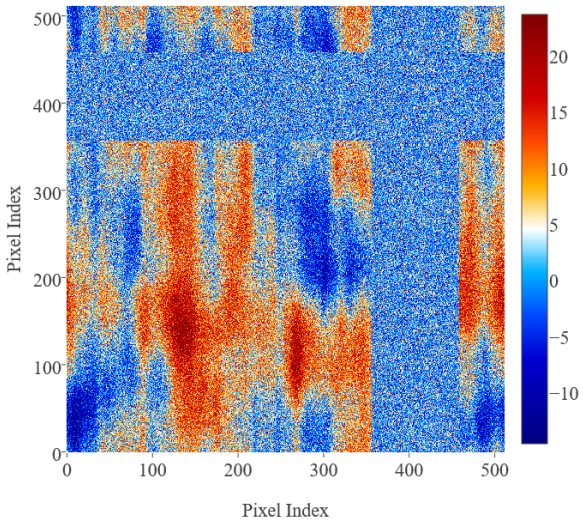
inference algorithms that reconstruct signals which have a mixed correlation structure. In the above case, a Wiener filter was applied in the context of a two-dimensional regular grid geometry. With NIFTy 3 this can easily be extended to, for example, the case of the Cartesian product of two one-dimensional regular grid spaces, shown in Fig. 4.7. Now the power spectrum for each space-component of the signal field differs. Additionally, the response operator has individual smoothing lengths and masks for each of the spaces. In this example, the power spectrum of the space drawn on the vertical axis is steeper than the one of the horizontal space. As a consequence, the field possesses small scale structure primarily in the horizontal direction. A remarkable consequence of this is reflected in the uncertainty map. Although the occlusion mask is equally broad for each space, the uncertainty is much higher in the direction of the horizontal space. There, the Wiener filter is not able to interpolate as well as for the large scale dominated vertical space.
Since the structure of the code for the case of a Cartesian product of spaces is very similar to the one in Fig. 4.5 – actually, one mainly has to define the two individual geometries respectively – the code is not shown explicitly here. For interested readers, the code that was used to produce the plots given in Fig. 4.7 is available as a demo in the NIFTy 3 code release, which is available here: `https://gitlab.mpcdf.mpg.de/ift/NIFTy`.

4.6 Conclusion

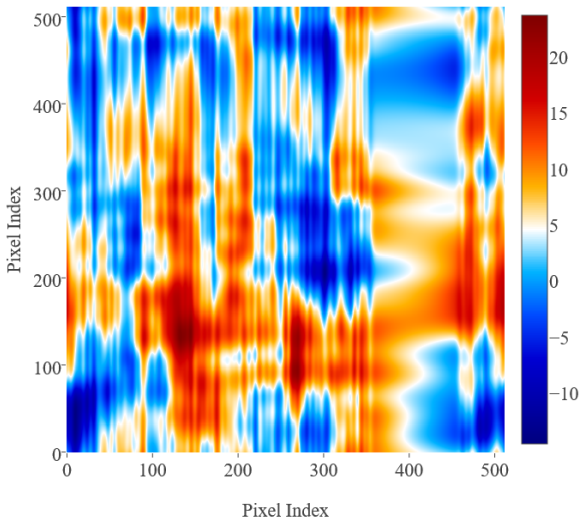
NIFTy 3 allows the programming of field equations independent of the underlying geometry or resolution. This freedom is particularly desirable in the implementation of inference algorithms of continuous quantities. This behavior is achieved by an object-oriented structure which cleanly separates the abstract mathematical operations from the underlying numerical calculations. By using layers of abstraction the operations are kept general and simple while preserving the continuum limit. Normalizations are applied automatically without the need of specification by the user. NIFTy 3 comes with full support of n-dimensional Cartesian spaces, the surface of the sphere and product spaces generated from them. Other geometries can be included in a straightforward fashion due to the layers of abstraction in NIFTy 3.
Algorithms implemented with NIFTy 3 are suitable to be run on HPC clusters with (almost) no MPI awareness needed from the user. With NIFTy 3 formulas can be transformed easily into code and, conversely, code can be easily read as formulas. This allows both, rapid prototyping and the implementation of large-scale algorithms.
NIFTy 3 has been developed to ease the implementation of inference algorithms on continuous quantities. Therefore, it is applicable in various areas. The first version of NIFTy already enabled numerous inference projects including many imaging algorithms performing interferometry, tomography (both astronomical and medical) or deconvolution, but also more abstract inference algorithms such as the estimation of cosmological parameters or instrument calibration. With the additional power of NIFTy 3 it can already be observed that this variety now grows even further.
NIFTy 3 is open-source software available under the GNU General Public License v3 (GPL-3) at `https://gitlab.mpcdf.mpg.de/ift/NIFTy/`.



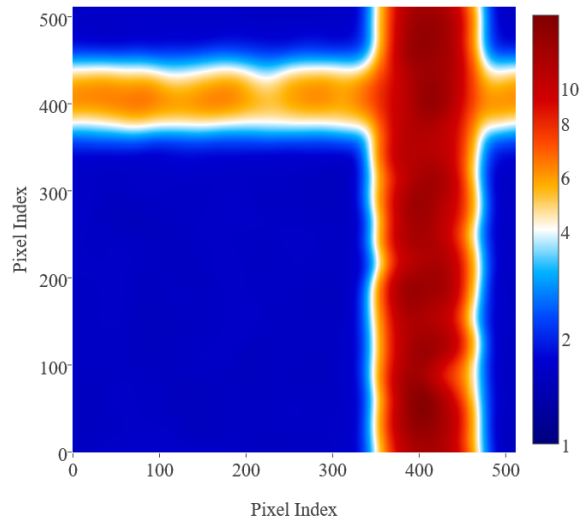
(a) Mock signal with two individual power spectra for each space.



(b) Data, which is the result of smoothing, masking and additive noise.



(c) Maximum a-posteriori reconstruction.



(d) Uncertainty map shown on a logarithmic scale.

Figure 4.7: Illustration of a Wiener filter reconstruction in the context of the Cartesian product of two one-dimensional regular grid spaces. The labeling of the axes shows pixel numbers.

Acknowledgements

Part of this work was supported by the *Studienstiftung des deutschen Volkes*.

Chapter 5

Further Work

This chapter lists all further publications I have been part of. As my contributions to those are marginal these publications are not shown in their full extend, only their abstracts.

5.1 Field dynamics inference via spectral density estimation

This section is used additionally as a journal publication in Physical Review E *(Frank et al., 2017)*

Stochastic differential equations (SDEs) are of utmost importance in various scientific and industrial areas. They are the natural description of dynamical processes whose precise equations of motion are either not known or too expensive to solve, e.g., when modeling Brownian motion. In some cases, the equations governing the dynamics of a physical system on macroscopic scales occur to be unknown since they typically cannot be deduced from general principles. In this work, we describe how the underlying laws of a stochastic process can be approximated by the spectral density of the corresponding process. Furthermore, we show how the density can be inferred from possibly very noisy and incomplete measurements of the dynamical field. Generally, inverse problems like these can be tackled with the help of *Information Field Theory* (IFT). For now, we restrict to *linear* and *autonomous* processes. Though, this is a non-conceptual limitation that may be omitted in future work. To demonstrate its applicability we employ our reconstruction algorithm on a time-series and spatio-temporal processes.

5.2 Search for quasi-periodic signals in magnetar giant flares

This section is used additionally as a journal publication in Astronomy & Astrophysics *(Pumpe et al., 2017)*

Quasi-periodic oscillations (QPOs) discovered in the decaying tails of giant flares of magnetars are believed to be torsional oscillations of neutron stars. These QPOs have a high potential to constrain properties of high-density matter. In search for quasi-periodic signals, we study

the light curves of the giant flares of SGR 1806-20 and SGR 1900+14, with a non-parametric Bayesian signal inference method called D^{3}PO. The D^{3}PO algorithm models the raw photon counts as a continuous flux and takes the Poissonian shot noise as well as all instrument effects into account. It reconstructs the logarithmic flux and its power spectrum from the data. Using this fully noise-aware method, we do not confirm previously reported frequency lines at $\nu \gtrsim 17\,\mathrm{Hz}$ because they fall into the noise-dominated regime. However, we find two new potential candidates for oscillations at 9.2 Hz (SGR 1806-20) and 7.7 Hz (SGR 1900+14). If these are real and the fundamental magneto-elastic oscillations of the magnetars, current theoretical models would favour relatively weak magnetic fields $\bar{B} \sim 6 \times 10^{13} - 3 \times 10^{14}\,\mathrm{G}$ (SGR 1806-20) and a relatively low shear velocity inside the crust compared to previous findings.

5.3 Inference of signals with unknown correlation structure from nonlinear measurements

This section is used additionally as a publication submitted to the Journal of Machine Learning Research *(Knollmüller et al., 2017)*

We present a method to reconstruct autocorrelated signals together with their autocorrelation structure from nonlinear, noisy measurements for arbitrary monotonous nonlinear instrument response. In the presented formulation the algorithm provides a significant speedup compared to prior implementations, allowing for a wider range of application. The nonlinearity can be used to model instrument characteristics or to enforce properties on the underlying signal, such as positivity. Uncertainties on any posterior quantities can be provided due to independent samples from an approximate posterior distribution. We demonstrate the methods applicability via simulated and real measurements, using different measurement instruments, nonlinearities and dimensionality.

Conclusion and Outlook

We have paved the way for future parametric as well as non-parametric magnetic field inference. With IMAGINE, there is now a research platform that allows to sustainably build parametric models of the Milky Way by allowing to easily keep their parameter fits up to date in the future. The magnetic field models that were analyzed with IMAGINE can easily be compared via Bayesian model comparison. Furthermore, IMAGINE provides a generic ansatz to treat Galactic variance correctly, regardless of the individual peculiarities of the specific GMF models. The results are honest statistical statements, the value of which we have shown on the basis of mock data as well as real data. Ignoring the galactic variance leads to considerable inaccuracies and misinterpretations, which can be seen in the fact that with new data becoming available in the past, support for certain properties of GMF models have changed dramatically. With IMAGINE it is now possible to produce much more steady inference results.

For sophisticated non-parametric inference of large 3D fields, there has been a need to catch up with respect to the available methods as well as the available data. With the development of D2O and the subsequent advances in NIFTY, the basis for high-resolution and entangled inference, in which several fields such as dust density, thermal electron density, and the GMF are inferred at the same time, is now available. Based on recent starlight extinction data like Pan-STARRS 1, or the soon to be expected Gaia data release 2, as a next step the creation of a 3D dust map should be conducted. Further, parametric just as well as non-parametric reconstructions should be set up as joint analyses of the thermal electron density and the GMF to resolve the degeneracies between the fields. NIFTY now offers structure that allows scaling existing 3D tomography algorithms to joint scalar/vector field reconstructions, that were previously used for reconstructing the thermal electron density only. With respect to parametric inference, IMAGINE allows for carrying out joint-reconstructions right away. Now the IMAGINE data library must be continuously extended. Due to the high complexity and interdependence of the physical quantities to be reconstructed, high data quality is all the more important. First a combination of the upcoming next-generation data releases, such as the Gaia DR 2 or the SKA data release mentioned above, will make it possible to fully exploit the potential of the methods created.

In addition to the challenge of reconstructing the GMF, the research and developments carried out in this work, especially in terms of NIFTY, also add value to many other applications: Numerous projects in the fields of interferometry, tomography (both astronomical and medical) and deconvolution, but also more abstract inference algorithms such as the estimation of cosmological parameters have only been made possible by NIFTY 3, cf. e.g., Frank et al. (2017), Knollmüller et al. (2017), Pumpe et al. (2017, 2018). Thus, the surplus value of this work does

not only extend to the question of GMF inference, but goes far beyond that.

Appendix A

D2O Appendix

A.1 Advanced Usage and Functional Behavior

Here we discuss specifics regarding the design and functional behavior of D2O. We set things up by importing `numpy` and `d2o.distributed_data_object`:

```
In [1]: import numpy as np
In [2]: from d2o import distributed_data_object
```

A.1.1 Distribution Strategies

In order to see the effect of different distribution strategies one may run the following script using three MPI processes. In lines 13 and 19, the `distribution_strategy` keyword is used for explicit specification of the strategy.

```
mpirun -n 3 python distribution_schemes.py
```

```
# distribution_schemes.py
from mpi4py import MPI
import numpy as np
from d2o import distributed_data_object
rank = MPI.COMM_WORLD.rank

a = np.arange(16).reshape((4, 4))
if rank == 0: print((rank, a))

# use 'not', 'equal' and 'fftw'
for strategy in ['not', 'equal', 'fftw']:
    obj = distributed_data_object(
                a, distribution_strategy=strategy)
    print (rank, strategy, obj.get_local_data())
```

```
15
16 # use the 'freeform' slicer
17 a += rank
18 obj = distributed_data_object(
19           local_data=a, distribution_strategy='freeform')
20 print (rank, 'freeform', obj.get_local_data())
21
22 full_data = obj.get_full_data()
23 if rank == 0: print (rank, 'freeform', full_data)
```

The printout in line 8 shows the a array.

```
(0, array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11],
           [12, 13, 14, 15]]))
```

The "not" distribution strategy stores full copies of the data on every node:

```
(0, 'not', array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11],
                  [12, 13, 14, 15]]))
(1, 'not', array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11],
                  [12, 13, 14, 15]]))
(2, 'not', array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11],
                  [12, 13, 14, 15]]))
```

The "equal", "fftw" and "freeform" distribution strategies are all subtypes of the *slicing* distributor that cuts the global array along its first axis. Therefore they only differ by the lengths of their subdivisions. The "equal" scheme tries to distribute the global array as equally as possible among the processes. If the array's size makes it necessary, the first processes will get an additional row. In this example the first array axis has a length of four but there are three MPI processes; hence, one gets a distribution of $(2, 1, 1)$:

```
(0, 'equal', array([[0, 1, 2, 3],
                    [4, 5, 6, 7]]))
(1, 'equal', array([[ 8,  9, 10, 11]]))
(2, 'equal', array([[12, 13, 14, 15]]))
```

The "fftw" distribution strategy is very similar to "equal" but uses functions from FFTWFrigo (1999). If the length of the first array axis is large compared to the number of processes they

will practically yield the same distribution pattern but for small arrays they may differ. For performance reasons FFTW prefers multiples of two over a uniform distribution, hence one gets $(2, 2, 0)$:

```
(0, 'fftw', array([[0, 1, 2, 3],
                   [4, 5, 6, 7]]))
(1, 'fftw', array([[ 8,  9, 10, 11],
                   [12, 13, 14, 15]]))
(2, 'fftw', array([], shape=(0, 4), dtype=int64))
```

A "`freeform`" array is built from a process-local perspective: each process gets its individual local data. In our example, we use `a+rank` as the local data arrays – each being of shape $(4, 4)$ – during the initialization of the *distributed_data_object*. By this, a global shape of $(12, 4)$ is produced. The local data reads:

```
(0, 'freeform', array([[ 0,  1,  2,  3],
                       [ 4,  5,  6,  7],
                       [ 8,  9, 10, 11],
                       [12, 13, 14, 15]]))
(1, 'freeform', array([[ 1,  2,  3,  4],
                       [ 5,  6,  7,  8],
                       [ 9, 10, 11, 12],
                       [13, 14, 15, 16]]))
(2, 'freeform', array([[ 2,  3,  4,  5],
                       [ 6,  7,  8,  9],
                       [10, 11, 12, 13],
                       [14, 15, 16, 17]]))
```

This yields a global shape of $(12, 4)$. In oder to consolidate the data the method `obj.get_full_data()` is used, cf. section Sec. A.1.3.

```
(0, 'freeform', array([[ 0,  1,  2,  3],
                       [ 4,  5,  6,  7],
                       [ 8,  9, 10, 11],
                       [12, 13, 14, 15],
                       [ 1,  2,  3,  4],
                       [ 5,  6,  7,  8],
                       [ 9, 10, 11, 12],
                       [13, 14, 15, 16],
                       [ 2,  3,  4,  5],
                       [ 6,  7,  8,  9],
                       [10, 11, 12, 13],
                       [14, 15, 16, 17]]))
```

A.1.2 Initialization

There are several different ways of initializing a *distributed_data_object*. In all cases its shape and data type must be specified implicitly or explicitly. In the previous section we encountered the basic way of supplying an initial data array which then gets distributed:

```
In [3]: a = np.arange(12).reshape((3, 4))
In [4]: obj = distributed_data_object(a)
# equivalent to line above
In [5]: obj = distributed_data_object(global_data=a)
```

The initial data is interpreted as global data. The default distribution strategy[1] is a *global-type* strategy, which means that the distributor which is constructed at initialization time derives its concrete data partitioning from the desired global shape and data type. A more explicit example for an initialization is:

```
In [6]: obj = distributed_data_object(global_data=a,
                                      dtype=np.complex)
```

In contrast to `a`'s data type which is *integer* we enforce the *distributed_data_object* to be *complex*. Without initial data – cf. `np.empty` – one may use the `global_shape` keyword argument:

```
In [7]: obj = distributed_data_object(global_shape=(2,3),
                                      dtype=np.float)
# equivalent to line above
In [8]: obj = distributed_data_object(global_shape=(2,3))
```

If the data type is specified neither implicitly by some initial data nor explicitly via `dtype`, *distributed_data_object* uses `float` as a default[2]. In contrast to *global-type*, *local-type* distribution strategies like "`freeform`" are defined by local shape information. The aptly named analoga to `global_data` and `global_shape` are `local_data` and `local_shape`, cf. section Sec. A.1.1:

```
In [9]: obj = distributed_data_object(
                    local_data=a,
                    distribution_strategy='freeform')
```

If redundant but conflicting information is provided – like integer-type initialization array vs. `dtype=complex` – the explicit information gained from `dtype` is preferred over implicit information provided by `global_data`/`local_data`. On the contrary, if data is provided, explicit information from `global_shape`/`local_shape` is discarded. In summary, `dtype` takes precedence over `global_data`/`local_data` which in turn takes precedence over `global_shape`/`local_shape`.

Please note that besides numpy arrays, *distributed_data_objects* are valid input for `global_data`/`local_data`, too. If necessary, a redistribution of data will be performed internally. When using `global_data` this will be the case if the distribution strategies of the input and ouput *distributed_data_objects* do not match. When *distributed_data_objects* are used as `local_data`

[1] Depending on whether *pyfftw* is available or not, the *equal-* or the *fftw*-distribution strategy is used, respectively; cf. section Sec. A.1.1.

[2] This mimics numpys behavior.

their full content will be concentrated on the individual processes. This means that if one uses the same *distributed_data_object* as `local_data` in, for example, two processes, the resulting *distributed_data_object* will have twice the memory footprint.

A.1.3 Getting and Setting Data

There exist three different methods for getting and setting a *distributed_data_object*'s data:

- `get_full_data` consolidates the full data into a numpy array,
- `set_full_data` distributes a given full-size array,
- `get_data` extracts a part of the data and returns it packed in a new *distributed_data_object*
- `set_data` modifies parts of the *distributed_data_object*'s data,
- `get_local_data` returns the process' local data,
- `set_local_data` directly modifies the process' local data.

In principle, one could mimic the behavior of `set_full_data` with `set_data` but the former is faster since there are no indexing checks involved. *distributed_data_objects* support large parts of numpy's indexing functionality, via the methods `get_data` and `set_data`[3]. This includes simple and advanced indexing, slicing and boolean extraction. Note that multidimensional advanced indexing is currently not supported by the slicing distributor: something like

```
obj[(np.array([[1,2], [0,1]]), np.array([[0,1], [2,3]]))]
```

will throw an exception.

```
In [10]: a = np.arange(12).reshape(3, 4)
In [11]: obj = distributed_data_object(a)
In [12]: obj
Out[12]: <distributed_data_object>
         array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

# Simple indexing
In [13]: obj[2,1]
Out[13]: 9

# Advanced indexing
In [14]: index_tuple = (np.array([1, 1, 2, 2, 2, 2]),
                        np.array([2, 3, 0, 1, 2, 3]))
In [15]: obj[index_tuple]
Out[15]: <distributed_data_object>
         array([ 6,  7,  8,  9, 10, 11])
```

[3] These are the methods getting called through Python's `obj[...]=...` notation.

```
19
20 # Slicing
21 In [16]: obj[:, ::-2]
22 Out[16]: <distributed_data_object>
23         array([[ 3,  1],
24                [ 7,  5],
25                [11,  9]])
26
27 # Boolean extraction
28 In [17]: obj[obj>5]
29 Out[17]: <distributed_data_object>
30         array([ 6,  7,  8,  9, 10, 11])
```

All those indexing variants can also be used for setting array data, for example:

```
1 In [18]: a = np.arange(12).reshape(3, 4)
2 In [19]: obj = distributed_data_object(a)
3 In [20]: obj[obj>5] = [11, 22, 33, 44, 55, 66]
4 In [21]: obj
5 Out[21]: <distributed_data_object>
6         array([[ 0,  1,  2,  3],
7                [ 4,  5, 11, 22],
8                [33, 44, 55, 66]])
```

Allowed types for input data are scalars, tuples, lists, numpy ndarrays and *distributed_data_objects*. Internally the individual processes then extract the locally relevant portion of it.
As it is extremely costly, D2O tries to avoid inter-process communication whenever possible. Therefore, when using the `get_data` method the returned data portions remain on their processes. In case of a *distributed_data_object* with a slicing distribution strategy the *freeform distributor* is used for this, cf. section Sec. A.1.1.

A.1.4 Local Keys

The distributed nature of D2O adds an additional degree of freedom when getting (setting) data from (to) a *distributed_data_object*. The indexing discussed in section Sec. A.1.3 is based on the assumption that the involved key- and data-objects are the same for every MPI node. But in addition to that, D2O allows the user to specify node-individual keys and data. This, for example, can be useful when data stored as a *distributed_data_object* must be piped into a software module which needs very specific portions of the data on each MPI process. If one is able to describe those data portions as array-indexing keys – like slices – then the user can do this data redistribution within a single line. The following script – executed by two MPI processes – illustrates the point of local keys.

```
mpirun -n 2 python local_keys.py
```

```
1 # local_keys.py
```

```
2  from mpi4py import MPI
3  import numpy as np
4  from d2o import distributed_data_object
5  rank = MPI.COMM_WORLD.rank
6
7  # initializing some data
8  obj = distributed_data_object(np.arange(16)*2)
9
10 print (rank, obj)
11
12 # getting data using the same slice on both processes
13 print (rank, obj.get_data(key=slice(None, None, 2)))
14
15 # getting data using different slices
16 print (rank, obj.get_data(key=slice(None, None, 2+rank),
17                           local_keys=True))
18
19 # getting data using different distributed_data_objects
20 key_tuple = (distributed_data_object([1, 3, 5, 7]),
21              distributed_data_object([2, 4, 6, 8]))
22 key = key_tuple[rank]
23 print (rank, obj.get_data(key=key, local_keys=True))
```

The first print statement shows the starting data: the even numbers ranging from 0 to 30:

```
(0, <distributed_data_object>
    array([ 0,  2,  4,  6,  8, 10, 12, 14]))
(1, <distributed_data_object>
    array([16, 18, 20, 22, 24, 26, 28, 30]))
```

In line 13 we extract every second entry from `obj` using `slice(None, None, 2)`. Here, no inter-process communication is involved; the yielded data remains on the original node. The output of the print statement reads:

```
(0, <distributed_data_object>
    array([ 0,  4,  8, 12]))
(1, <distributed_data_object>
    array([16, 20, 24, 28]))
```

In line 17 the processes ask for different slices of the global data using the keyword `local_keys = True`: process 0 requests every second element whereas process 1 requests every third element from `obj`. Now communication is required to redistribute the data and the results are stored in the individual processes.

```
(0, <distributed_data_object>
    array([ 0,  4,  8, 12, 16, 20, 24, 28]))
(1, <distributed_data_object>
    array([ 0,  6, 12, 18, 24, 30]))
```

In line 23 we use *distributed_data_objects* as indexing objects. Process 0 requests the elements at positions 1, 3, 5 and 7; process 1 for those at 2, 4, 6 and 8. The peculiarity here is that the keys are not passed to `obj` as a complete set of local *distributed_data_object* instances. In fact, the processes only hand over their very local instance of the keys. D2O is aware of this and uses the `d2o_librarian` in order to reassemble them, cf. section Sec. A.1.5. The output reads:

```
(0, <distributed_data_object>
    array([ 2,  6, 10, 14]))
(1, <distributed_data_object>
    array([ 4,  8, 12, 16]))
```

The `local_keys` keyword is also available for the `set_data` method. In this case the keys as well as the data updates will be considered local objects. The behaviour is analogous to the one of `get_data`: The individual processes store the locally relevant part of the `to_key` using their distinct `data[from_key]`.

A.1.5 The d2o Librarian

A *distributed_data_object* as an abstract entity in fact consists of a set of Python objects that reside in memory of each MPI process. Global operations on a *distributed_data_object* necessitate that all those local instances of a *distributed_data_object* receive the same function calls; otherwise unpredictable behavior or a deadlock could happen. Let us discuss an illustrating example, the case of extracting a certain piece of data from a *distributed_data_object* using slices, cf. section Sec. A.1.3. Given a request for a slice of data, the MPI processes check which part of their data is covered by the slice, and build a new *distributed_data_object* from that. Thereby they communicate the size of their local data, maybe make sanity checks, and more. If this `get_data(slice(...))` function call is not made on every process of the cluster, a deadlock will occur as the 'called' processes wait for the 'uncalled' ones. However, especially when using the `local_keys` functionality described in section Sec. A.1.4 algorithmically one would like to work with different, i.e. node-individual *distributed_data_objects* at the same time. This raises the question: given only one local Python object instance, how could one make a global call on the complete *distributed_data_object* entity it belongs to? For this the `d2o_librarian` exists. During initialization every *distributed_data_object* registers itself with the `d2o_librarian` which returns a unique index. Later, this index can be used to assemble the full *distributed_data_object* from just a single local instance. The following code illustrates the workflow.

```
mpirun -n 4 python librarian.py
```

```
1 # librarian.py
2 from mpi4py import MPI
3 import numpy as np
4 from d2o import distributed_data_object, d2o_librarian
5
6 comm = MPI.COMM_WORLD
```

```
rank = comm.rank

# initialize four different distributed_data_objects
obj = distributed_data_object(np.arange(16).reshape((4,4)))
obj_list = (obj, 2*obj, 3*obj, 4*obj)

# every process gets its part of the respective full array
individual_object = obj_list[rank]
individual_index = individual_object.index
index_list = comm.allgather(individual_index)

for index in index_list:
    # resemble the current d2o on every node
    current_object = d2o_librarian[index]
    if rank == 0: print('Index: ' + str(index))
    # take a slice of data
    print (rank, current_object[:, 2:4].get_local_data())
```

The output reads:

```
Index: 1
(0, array([[2, 3]]))
(1, array([[6, 7]]))
(2, array([[10, 11]]))
(3, array([[14, 15]]))
Index: 2
(0, array([[4, 6]]))
(1, array([[12, 14]]))
(2, array([[20, 22]]))
(3, array([[28, 30]]))
Index: 3
(0, array([[6, 9]]))
(1, array([[18, 21]]))
(2, array([[30, 33]]))
(3, array([[42, 45]]))
Index: 4
(0, array([[ 8, 12]]))
(1, array([[24, 28]]))
(2, array([[40, 44]]))
(3, array([[56, 60]]))
```

The D2O-librarian's core-component is a *weak dictionary* wherein *weak references* to the local *distributed_data_object* instances are stored. Its peculiarity is that those weak references do not prevent Python's garbage collector from deleting the object once no regular references to it are left. By this, the librarian can keep track of the *distributed_data_objects* without, at the same time, being a reason to hold them in memory.

A.1.6 Copy Methods

D2O's array copy methods were designed to avoid as much Python overhead as possible. Nevertheless, there is a speed penalty compared to pure numpy arrays for a single process; cf. section Sec. 3.4 for details. This is important as binary operations like addition or multiplication of an array need a copy for returning the results. A special feature of D2O is that during a full copy one may change certain array properties such as the data type and the distribution strategy:

```
In [22]: a = np.arange(4)
In [23]: obj = distributed_data_object(a) # dtype == np.int
In [24]: p = obj.copy(dtype=np.float,
                      distribution_strategy='not')
In [25]: (p.distribution_strategy, p)
Out[25]: ('not', <distributed_data_object>
                 array([ 0.,  1.,  2.,  3.]))
```

When making empty copies one can also change the global or local shape:

```
In [26]: obj = distributed_data_object(global_shape=(4,4),
                                       dtype=np.float)
# only the shape gets changed
In [27]: obj.copy_empty(global_shape=(2,2))
Out[27]: <distributed_data_object>
         array([[  6.90860823e-310,   9.88131292e-324],
                [  9.88131292e-324,   1.97626258e-323]])
```

A.1.7 Fast Iterators

A large class of problems requires iteration over the elements of an array one by one Ka-Ping Yee (2016). Whenever possible, Python uses special *iterators* for this in order to keep computational costs at a minimum. A toy example is

```
In [28]: l = [9, 8, 7, 6]
In [29]: for item in l:
             print item
....:
9
8
7
6
```

Inside Python, the `for` loop requests an iterator object from the list `l`. Then the loop pulls elements from this iterator until it is exhausted. If an object is not able to return an iterator, the `for` loop will extract the elements using `__getitem__` over and over again. In the case of *distributed_data_objects* the latter would be extremely inefficient as every `__getitem__` call incorporates a significant amount of communication. In order to circumvent this, the iterators of *distributed_data_objects* communicate the process' data in chunks that are as big as possible.

Thereby we exploit the knowledge that the array elements will be fetched one after another by the iterator. An examination of the performance difference is done in appendix Sec. A.2.

A.2 Iterator Performance

As discussed in section Sec. A.1.7, iterators are a standard tool in Python by which objects control their behavior in `for` loops and list comprehensions Ka-Ping Yee (2016). In order to speed up the iteration process, *distributed_data_objects* communicate their data as chunks chosen to be as big as possible. Thereby D2O builds upon the knowledge that elements will be fetched one after another by the iterator as long as further elements are requested.[4] Additionally, by its custom iterator interface D2O avoids that the full data consolidation logic is invoked for every entry. Because of this, the performance gain is roughly a factor of 30 even for single-process scenarios as demonstrated in the following example:

```
In [1]: length = 1000
In [2]: obj = distributed_data_object(np.arange(length))

In [3]: def using_iterators(obj):
            for i in obj:
                pass

In [4]: def not_using_iterators(obj):
            for j in xrange(length):
                obj[j]

In [5]: %timeit not_using_iterators(obj)
        10 loops, best of 3: 104 ms per loop

In [6]: %timeit using_iterators(obj)
        100 loops, best of 3: 2.92 ms per loop
```

[4] This has the downside, that if the iteration was stopped prematurely, data has been communicated in vain.

Bibliography

R. Alves Batista, A. Dundovic, M. Erdmann, K.-H. Kampert, D. Kuempel, G. Müller, G. Sigl, A. van Vliet, D. Walz, and T. Winchen. CRPropa 3 - a public astrophysical simulation framework for propagating extraterrestrial ultra-high energy particles. *J. Cosmology Astropart. Phys.*, 5:038, May 2016. doi: 10.1088/1475-7516/2016/05/038.

Apache Software Foundation. Hadoop, 2016. URL `https://hadoop.apache.org`.

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2015. URL `http://www.mcs.anl.gov/petsc`.

G. Battaglia, A. Helmi, H. Morrison, P. Harding, E. W. Olszewski, M. Mateo, K. C. Freeman, J. Norris, and S. A. Shectman. The radial velocity dispersion profile of the Galactic halo: constraining the density profile of the dark halo of the Milky Way. *MNRAS*, 364:433–442, December 2005. doi: 10.1111/j.1365-2966.2005.09367.x.

Giuseppina Battaglia, Amina Helmi, Heather Morrison, Paul Harding, Edward W. Olszewski, Mario Mateo, Kenneth C. Freeman, John Norris, and Stephen A. Shectman. Erratum: The radial velocity dispersion profile of the galactic halo: constraining the density profile of the dark halo of the milky way. *Monthly Notices of the Royal Astronomical Society*, 370(2): 1055–1056, 2006. doi: 10.1111/j.1365-2966.2006.10688.x. URL +`http://dx.doi.org/10.1111/j.1365-2966.2006.10688.x`.

Rainer Beck. Galactic and extragalactic magnetic fields. *Space Science Reviews*, 99(1):243–260, Oct 2001. ISSN 1572-9672. doi: 10.1023/A:1013805401252. URL `https://doi.org/10.1023/A:1013805401252`.

Rainer Beck and Marita Krause. Revised equipartition & minimum energy formula for magnetic field strength estimates from radio synchrotron observations. *Astron. Nachr.*, 326:414–427, 2005. doi: 10.1002/asna.200510366.

M. R. Bell and T. A. Enßlin. Faraday synthesis. The synergy of aperture and rotation measure synthesis. *A&A*, 540:A80, April 2012. doi: 10.1051/0004-6361/201118672.

M. R. Bell, H. Junklewitz, and T. A. Enßlin. Faraday caustics. Singularities in the Faraday spectrum and their utility as probes of magnetic field properties. *aap*, 535:A85, November 2011. doi: 10.1051/0004-6361/201117254.

C. L. Bennett, R. S. Hill, G. Hinshaw, M. R. Nolta, N. Odegard, L. Page, D. N. Spergel, J. L. Weiland, E. L. Wright, M. Halpern, N. Jarosik, A. Kogut, M. Limon, S. S. Meyer, G. S. Tucker, and E. Wollack. First-Year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Foreground Emission. *ApJS*, 148:97–117, September 2003. doi: 10.1086/377252.

M. Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo. *ArXiv e-prints*, January 2017.

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN 0-89871-397-8 (paperback).

Vanessa Böhm, Stefan Hilbert, Maksim Greiner, and Torsten A. Enßlin. Bayesian weak lensing tomography: Reconstructing the 3d large-scale distribution of matter with a lognormal prior. *Phys. Rev. D*, 96:123510, Dec 2017. doi: 10.1103/PhysRevD.96.123510. URL `https://link.aps.org/doi/10.1103/PhysRevD.96.123510`.

A. Brandenburg and K. Subramanian. Astrophysical magnetic fields and nonlinear dynamo theory. *Phys. Rep.*, 417:1–209, October 2005. doi: 10.1016/j.physrep.2005.06.005.

S. Brooks, A. Gelman, G. Jones, and X.L. Meng. *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC Handbooks of Modern Statistical Methods. CRC Press, 2011. ISBN 9781420079425. URL `https://books.google.de/books?id=qfRsAIKZ4rIC`.

J. Buchner, A. Georgakakis, K. Nandra, L. Hsu, C. Rangel, M. Brightman, A. Merloni, M. Salvato, J. Donley, and D. Kocevski. X-ray spectral modelling of the AGN obscuring region in the CDFS: Bayesian model selection and catalogue. *A&A*, 564:A125, April 2014. doi: 10.1051/0004-6361/201322971.

B. J. Burn. On the depolarization of discrete radio sources by Faraday dispersion. *MNRAS*, 133: 67, 1966. doi: 10.1093/mnras/133.1.67.

D. Buscombe. Spatially explicit spectral analysis of point clouds and geospatial data. *Computers and Geosciences*, 86:92–108, January 2016. doi: 10.1016/j.cageo.2015.10.004.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995. ISSN 1064-8275. doi: 10.1137/0916069. URL `http://dx.doi.org/10.1137/0916069`.

T. Tony Cai, Tengyuan Liang, and Harrison H. Zhou. Law of log determinant of sample covariance matrix and optimal estimation of differential entropy for high-dimensional

gaussian distributions. *Journal of Multivariate Analysis*, 137(Supplement C):161 – 172, 2015. ISSN 0047-259X. doi: https://doi.org/10.1016/j.jmva.2015.02.003. URL http://www.sciencedirect.com/science/article/pii/S0047259X1500038X.

Anthony Challinor, Pablo Fosalba, Daniel Mortlock, Mark Ashdown, Benjamin Wandelt, and Krzysztof Gorski. All-sky convolution for polarimetry experiments. *Physical Review D*, 62 (12), November 2000. ISSN 0556-2821, 1089-4918. doi: 10.1103/PhysRevD.62.123002. URL http://arxiv.org/abs/astro-ph/0008228.

Weizhu Chen, Zhenghao Wang, and Jingren Zhou. Large-scale l-bfgs using mapreduce. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1332–1340. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5333-large-scale-l-bfgs-using-mapreduce.pdf.

Y. Chen, A. Wiesel, and A. O. Hero. Robust Shrinkage Estimation of High-Dimensional Covariance Matrices. *IEEE Transactions on Signal Processing*, 59:4097–4107, September 2011. doi: 10.1109/TSP.2011.2138698.

J. M. Cordes. NE2001: A New Model for the Galactic Electron Density and its Fluctuations. In D. Clemens, R. Shah, and T. Brainerd, editors, *Milky Way Surveys: The Structure and Evolution of our Galaxy*, volume 317 of *Astronomical Society of the Pacific Conference Series*, page 211, December 2004.

J. M. Cordes and T. J. W. Lazio. NE2001.I. A New Model for the Galactic Distribution of Free Electrons and its Fluctuations. *ArXiv Astrophysics e-prints*, July 2002.

Intel Corporation. Intel mpi library, 2016. URL https://software.intel.com/en-us/intel-mpi-library.

R. T. Cox. Probability, frequency and reasonable expectation. *American Journal of Physics*, 14 (1):1–13, January 1946. doi: 10.1119/1.1990764. URL http://dx.doi.org/10.1119/1.1990764.

A. Dadone and B. Grossman. Ghost-Cell Method for Inviscid Two-Dimensional Flows on Cartesian Grids. *AIAA Journal*, 42:2499–2507, December 2004. doi: 10.2514/1.697.

Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

Lisandro Dalcín, Rodrigo Paz, and Mario Storti. {MPI} for python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2005.03.010. URL http://www.sciencedirect.com/science/article/pii/S0743731505000560.

L. Davis, Jr. and J. L. Greenstein. The Polarization of Starlight by Aligned Dust Grains. *ApJ*, 114:206, September 1951. doi: 10.1086/145464.

B R Dawson, M Fukushima, and P Sokolsky. Past, present, and future of uhecr observations. *Progress of Theoretical and Experimental Physics*, 2017(12):12A101, 2017. doi: 10.1093/ptep/ptx054. URL +http://dx.doi.org/10.1093/ptep/ptx054.

S. Dorn and T. A. Enßlin. Stochastic determination of matrix determinants. *Phys. Rev. E*, 92(1): 013302, July 2015. doi: 10.1103/PhysRevE.92.013302.

S. Dorn, T. A. Enßlin, M. Greiner, M. Selig, and V. Boehm. Signal inference with unknown response: Calibration-uncertainty renormalized estimator. *Phys. Rev. E*, 91(1):013311, January 2015a. doi: 10.1103/PhysRevE.91.013311.

S. Dorn, M. Greiner, and T. A. Enßlin. All-sky reconstruction of the primordial scalar potential from WMAP temperature data. *J. Cosmology Astropart. Phys.*, 2:041, February 2015b. doi: 10.1088/1475-7516/2015/02/041.

B. Draine. On the Interpretation of the λ 2175 Å Feature. In L. J. Allamandola and A. G. G. M. Tielens, editors, *Interstellar Dust*, volume 135 of *IAU Symposium*, page 313, 1989.

R. D. Ekers, J. Lequeux, A. T. Moffet, and G. A. Seielstad. A Measurement of the Galactic Magnetic Field Using the Pulsating Radio Source PSR 0833-45. *ApJ*, 156:L21, April 1969. doi: 10.1086/180341.

T. A. Enßlin and C. Weig. Inference with minimal Gibbs free energy in information field theory. *Phys. Rev. E*, 82(5):051112, November 2010. doi: 10.1103/PhysRevE.82.051112.

T. A. Enßlin, M. Frommert, and F. S. Kitaura. Information field theory for cosmological perturbation reconstruction and nonlinear signal analysis. *Phys. Rev. D*, 80(10):105005, November 2009. doi: 10.1103/PhysRevD.80.105005.

Torsten A Enßlin and Mona Frommert. Reconstruction of signals with unknown spectra in information field theory with parameter uncertainty. *Physical Review D*, 83(10):105014, May 2011. doi: 10.1103/PhysRevD.83.105014.

Inc. Enthought. Distarray: Think globally, act locally, 2016. URL http://docs.enthought.com/distarray/.

C. Evoli, D. Gaggero, A. Vittino, G. Di Bernardo, M. Di Mauro, A. Ligorini, P. Ullio, and D. Grasso. Cosmic-ray propagation with DRAGON2: I. numerical solver and astrophysical ingredients. *J. Cosmology Astropart. Phys.*, 2:015, February 2017. doi: 10.1088/1475-7516/2017/02/015.

G. R. Farrar, R. Jansson, I. J. Feain, and B. M. Gaensler. Galactic magnetic deflections and Centaurus A as a UHECR source. *Journal of Cosmology and Astro-Particle Physics*, 1:023, January 2013. doi: 10.1088/1475-7516/2013/01/023.

Douglas P. Finkbeiner. A full-sky h-alpha template for microwave foreground prediction. *The*

Astrophysical Journal Supplement Series, 146(2):407, 2003. URL http://stacks.iop.org/0067-0049/146/i=2/a=407.

J. Fitzsimons, K. Cutajar, M. Osborne, S. Roberts, and M. Filippone. Bayesian Inference of Log Determinants. *ArXiv e-prints*, April 2017.

R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6(2):163–168, 1963. URL http://www3.oup.co.uk/computer_journal/hdb/Volume_06/Issue_02/060163.sgm.abs.html.

Daniel Foreman-Mackey. corner.py: Scatterplot matrices in python. *The Journal of Open Source Software*, 24, 2016. doi: 10.21105/joss.00024. URL http://dx.doi.org/10.5281/zenodo.45906.

Charles Francis and Erik Anderson. Galactic spiral structure. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 465(2111):3425–3446, 2009. ISSN 1364-5021. doi: 10.1098/rspa.2009.0036. URL http://rspa.royalsocietypublishing.org/content/465/2111/3425.

P. Frank, T. Steininger, and T. A. Enßlin. Field dynamics inference via spectral density estimation. *Phys. Rev. E*, 96(5):052104, November 2017. doi: 10.1103/PhysRevE.96.052104.

Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301661. URL http://doi.acm.org/10.1145/301618.301661.

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*, volume 2. CRC press Boca Raton, FL, 2014.

G. Giacinti, M. Kachelrieß, D. Semikoz, and G. Sigl. Deflection of ultra-high energy heavy nuclei in the Galactic magnetic field. In *European Physical Journal Web of Conferences*, volume 53 of *European Physical Journal Web of Conferences*, page 6004, June 2013. doi: 10.1051/epjconf/20135306004.

T. Gold. Rotating Neutron Stars as the Origin of the Pulsating Radio Sources. *Nature*, 218: 731–732, May 1968. doi: 10.1038/218731a0.

Henry Gomersall. pyfftw: a pythonic wrapper around fftw, 2016. URL `https://hgomersall.github.io/pyFFTW`. We use the mpi branch available at `https://github.com/fredRos/pyFFTW`.

K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *ApJ*, 622:759–771, April 2005. doi: 10.1086/427976.

Krzysztof M Gorski, Eric Hivon, AJ Banday, Benjamin D Wandelt, Frode K Hansen, Mstvos Reinecke, and Matthia Bartelmann. Healpix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

M. Greiner, D. H. F. M. Schnitzeler, and T. A. Enßlin. Tomography of the Galactic free electron density with the Square Kilometer Array. *A&A*, 590:A59, May 2016. doi: 10.1051/0004-6361/201526717.

J. L. Han. Magnetic Fields in Our Galaxy: How much do we know? III. Progress in the Last Decade. *Chinese Journal of Astronomy and Astrophysics Supplement*, 6(2):211–217, December 2006.

W. E. Harris. A Catalog of Parameters for Globular Clusters in the Milky Way. *AJ*, 112:1487, October 1996. doi: 10.1086/118116.

D.A. Harville. *Matrix Algebra From a Statistician's Perspective*. Springer New York, 2008. ISBN 9780387783567. URL `https://books.google.de/books?id=kZGBQijgGV8C`.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109, 1970.

Jason W. T. Hessels, Scott M. Ransom, Ingrid H. Stairs, Paulo C. C. Freire, Victoria M. Kaspi, and Fernando Camilo. A radio pulsar spinning at 716 hz. *Science*, 311(5769):1901–1904, 2006. ISSN 0036-8075. doi: 10.1126/science.1123430. URL `http://science.sciencemag.org/content/311/5769/1901`.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(49):409–436, 1952.

R.V. Hogg and E.A. Tanis. *Probability and Statistical Inference*. Number Bd. 978,Nr. 0-58475 in Probability and Statistical Inference. Pearson/Prentice Hall, 2010. ISBN 9780321584755. URL `https://books.google.de/books?id=ihyyQwAACAAJ`.

Zongliang Hu, Kai Dong, Wenlin Dai, and Tiejun Tong. A comparison of methods for estimating the determinant of high-dimensional covariance matrix. *The international journal of biostatistics*, 13(2), 2017.

M.F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics - Simulation and Computation*, 18(3):1059–1076, 1989. doi: 10.1080/03610918908812806. URL http://dx.doi.org/10.1080/03610918908812806.

T R Jaffe, J P Leahy, A J Banday, S M Leach, S R Lowe, and A Wilkinson. Modelling the Galactic magnetic field on the plane in two dimensions. *MNRAS*, 401(2):1013–1028, January 2010. doi: 10.1111/j.1365-2966.2009.15745.x. URL http://mnras.oxfordjournals.org/cgi/doi/10.1111/j.1365-2966.2009.15745.x.

T R Jaffe, K M Ferrière, A J Banday, A W Strong, E Orlando, J F Macias-Perez, L Fauvet, C Combet, and E Falgarone. Comparing polarized synchrotron and thermal dust emission in the Galactic plane. *MNRAS*, 431(1):683–694, May 2013. doi: 10.1093/mnras/stt200. URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2013MNRAS.431..683J&link_type=ABSTRACT.

Ronnie Jansson and Glennys R Farrar. The Galactic Magnetic Field. *ApJL*, 761(1):L11, December 2012. doi: 10.1088/2041-8205/761/1/L11. URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2012ApJ...761L..11J&link_type=ABSTRACT.

E. T. Jaynes and R. Baierlein. Probability Theory: The Logic of Science. *Physics Today*, 57:76–77, October 2004. doi: 10.1063/1.1825273.

H. Jeffreys. *The Theory of Probability*. Oxford Classic Texts in the Physical Sciences. OUP Oxford, 1998. ISBN 9780191589676. URL https://books.google.de/books?id=vh9Act9rtzQC.

H. Junklewitz, M. R. Bell, M. Selig, and T. A. Enßlin. RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *A&A*, 586:A76, February 2016. doi: 10.1051/0004-6361/201323094.

Guido van Rossum Ka-Ping Yee. Pep 234 – iterators, 2016. URL https://www.python.org/dev/peps/pep-0234/.

P. R. Kafle, S. Sharma, G. F. Lewis, and J. Bland-Hawthorn. On the Shoulders of Giants: Properties of the Stellar Halo and the Milky Way Mass Distribution. *ApJ*, 794:59, October 2014. doi: 10.1088/0004-637X/794/1/59.

F. J. Kerr. The Large-Scale Distribution of Hydrogen in the Galaxy. *ARA&A*, 7:39, 1969. doi: 10.1146/annurev.aa.07.090169.000351.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983. doi: 10.1126/science.220.4598.671.

R. Kissmann. PICARD: A novel code for the Galactic Cosmic Ray propagation problem. *Astroparticle Physics*, 55:37–50, March 2014. doi: 10.1016/j.astropartphys.2014.02.002.

J. Knollmüller, T. Steininger, and T. A. Enßlin. Inference of signals with unknown correlation structure from nonlinear measurements. *ArXiv e-prints*, November 2017.

Jakob Knollmüller and Torsten A. Enßlin. Noisy independent component analysis of autocorrelated components. *Phys. Rev. E*, 96:042114, Oct 2017. doi: 10.1103/PhysRevE.96.042114. URL https://link.aps.org/doi/10.1103/PhysRevE.96.042114.

J.C. Lemm. *Bayesian Field Theory*. Bayesian Field Theory. Johns Hopkins University Press, 2003. ISBN 9780801877971.

E. H. Lieb and H.-T. Yau. A rigorous examination of the Chandrasekhar theory of stellar collapse. *ApJ*, 323:140–144, December 1987. doi: 10.1086/165813.

C. C. Lin and F. H. Shu. On the Spiral Structure of Disk Galaxies. *ApJ*, 140:646, August 1964. doi: 10.1086/147955.

Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, Aug 1989. ISSN 1436-4646. doi: 10.1007/BF01589116. URL http://dx.doi.org/10.1007/BF01589116.

M. Lutz. *Programming Python: Powerful Object-Oriented Programming*. O'Reilly Media, 2010. ISBN 9781449302757. URL https://books.google.de/books?id=q8W3WQbNWmkC.

D. Maoz. *Astrophysics in a Nutshell*. In a Nutshell. Princeton University Press, 2007. ISBN 9780691125848. URL https://books.google.de/books?id=UTytFkQTLFkC.

R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN 9780135974445. URL https://books.google.de/books?id=0HYhAQAAIAAJ.

S. Matarrese, F. Lucchin, and S. A. Bonometto. A path-integral approach to large-scale matter distribution originated by non-Gaussian fluctuations. *ApJ*, 310:L21–L26, November 1986. doi: 10.1086/184774.

Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. Building a framework for predictive science. *CoRR*, abs/1202.1056, 2012. URL http://arxiv.org/abs/1202.1056.

Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3–4):159–416, 1994.

Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

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, June 1953. doi: 10.1063/1.1699114. URL http://dx.doi.org/10.1063/1.1699114.

I. Moskalenko. Modeling cosmic ray propagation and associated interstellar emissions. In *39th COSPAR Scientific Assembly*, volume 39 of *COSPAR Meeting*, page 1281, July 2012.

J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Comput. J.*, 7:308–313, 1965. doi: 10.1093/comjnl/7.4.308.

J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. ISBN 9780387303031. URL `https://books.google.de/books?id=eNlPAAAAMAAJ`.

N. Oppermann, H. Junklewitz, G. Robbers, and T. A. Enßlin. Probing magnetic helicity with synchrotron radiation and Faraday rotation. *A&A*, 530:A89+, June 2011. doi: 10.1051/0004-6361/201015545.

N. Oppermann, H. Junklewitz, G. Robbers, M. R. Bell, T. A. Enßlin, A. Bonafede, R. Braun, J. C. Brown, T. E. Clarke, I. J. Feain, B. M. Gaensler, A. Hammond, L. Harvey-Smith, G. Heald, M. Johnston-Hollitt, U. Klein, P. P. Kronberg, S. A. Mao, N. M. McClure-Griffiths, S. P. O'Sullivan, L. Pratley, T. Robishaw, S. Roy, D. H. F. M. Schnitzeler, C. Sotomayor-Beltran, J. Stevens, J. M. Stil, C. Sunstrum, A. Tanna, A. R. Taylor, and C. L. Van Eck. An improved map of the Galactic Faraday sky. *A&A*, 542:A93, June 2012. doi: 10.1051/0004-6361/201118526.

N. Oppermann, H. Junklewitz, M. Greiner, T. A. Enßlin, T. Akahori, E. Carretti, B. M. Gaensler, A. Goobar, L. Harvey-Smith, M. Johnston-Hollitt, L. Pratley, D. H. F. M. Schnitzeler, J. M. Stil, and V. Vacca. Estimating extragalactic Faraday rotation. *A&A*, 575:A118, March 2015. doi: 10.1051/0004-6361/201423995.

F. Pacini. Rotating Neutron Stars, Pulsars and Supernova Remnants. *Nature*, 219:145–146, July 1968. doi: 10.1038/219145a0.

L Page, G Hinshaw, E Komatsu, M R Nolta, D N Spergel, C L Bennett, C Barnes, R Bean, O Doré, J Dunkley, and M Halpern. Three-Year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Polarization Analysis. *ApJS*, 170:335–376, 2007. doi: 10.1086/513699. URL `http://adsabs.harvard.edu/abs/2007ApJS..170..335P`.

Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL `http://ipython.org`.

Planck Collaboration, R. Adam, P. A. R. Ade, N. Aghanim, Y. Akrami, M. I. R. Alves, F. Argüeso, M. Arnaud, F. Arroja, M. Ashdown, and et al. Planck 2015 results. I. Overview of products and scientific results. *A&A*, 594:A1, September 2016a. doi: 10.1051/0004-6361/201527101.

Planck Collaboration, R. Adam, P. A. R. Ade, N. Aghanim, M. I. R. Alves, M. Arnaud, M. Ashdown, J. Aumont, C. Baccigalupi, A. J. Banday, and et al. Planck 2015 results. X. Dif-

fuse component separation: Foreground maps. *A&A*, 594:A10, September 2016b. doi: 10.1051/0004-6361/201525967.

Natàlia Porqueres, Torsten A. Enßlin, Maksim Greiner, Vanessa Böhm, Sebastian Dorn, Pilar Ruiz-Lapuente, and Alberto Manrique. Cosmic expansion history from SNe Ia data via information field theory – the charm code. *Astron. Astrophys.*, 599:A92, 2017. doi: 10.1051/0004-6361/201629527.

D. Pumpe, M. Greiner, E. Müller, and T. A. Enßlin. Dynamic system classifier. *Phys. Rev. E*, 94 (1):012132, July 2016. doi: 10.1103/PhysRevE.94.012132.

D. Pumpe, M. Gabler, T. Steininger, and T. A. Enßlin. Search for quasi-periodic signals in magnetar giant flares. *ArXiv e-prints*, August 2017.

D. Pumpe, M. Reinecke, and T. A. Enßlin. Denoising, Deconvolving and Decomposing multi-Dimensional Photon Observations- The D4PO Algorithm. *ArXiv e-prints*, February 2018.

R. J. Reynolds, F. L. Roesler, and F. Scherb. The Intensity Distribution of Diffuse Galactic Hα Emission. *ApJ*, 192:L53, September 1974. doi: 10.1086/181589.

B. Ruiz-Granados, J. A. Rubiño-Martín, and E. Battaner. Constraining the regular Galactic magnetic field with the 5-year WMAP polarization measurements at 22 GHz. *A&A*, 522:A73, November 2010. doi: 10.1051/0004-6361/200912733.

G.B. Rybicki and A.P. Lightman. *Radiative Processes in Astrophysics*. Physics textbook. Wiley, 2008. ISBN 9783527618187. URL `https://books.google.de/books?id=eswe2StAspsC`.

R. Schlickeiser. *Cosmic Ray Astrophysics*. Springer, 2002.

D. H. F. M. Schnitzeler. Modelling the Galactic distribution of free electrons. *MNRAS*, 427: 664–678, November 2012. doi: 10.1111/j.1365-2966.2012.21869.x.

M. Selig and T. A. Enßlin. Denoising, deconvolving, and decomposing photon observations. Derivation of the D^{3}PO algorithm. *A&A*, 574:A74, February 2015. doi: 10.1051/0004-6361/201323006.

M. Selig, N. Oppermann, and T. A. Enßlin. Improving stochastic estimates with inference methods: Calculating matrix diagonals. *Phys. Rev. E*, 85(2):021134, February 2012. doi: 10.1103/PhysRevE.85.021134.

M. Selig, M. R. Bell, H. Junklewitz, N. Oppermann, M. Reinecke, M. Greiner, C. Pachajoa, and T. A. Enßlin. NIFTY - Numerical Information Field Theory. A versatile PYTHON library for signal inference. *A&A*, 554:A26, June 2013. doi: 10.1051/0004-6361/201321236.

M. Selig, V. Vacca, N. Oppermann, and T. A. Enßlin. The denoised, deconvolved, and decomposed Fermi γ-ray sky. An application of the D^{3}PO algorithm. *A&A*, 581:A126, September 2015. doi: 10.1051/0004-6361/201425172.

C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3): 379–423, 1948. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL http://dx.doi.org/10.1002/j.1538-7305.1948.tb01338.x.

Jack Sherman and Winifred J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Ann. Math. Statist.*, 21(1):124–127, 03 1950. doi: 10.1214/aoms/1177729893. URL https://doi.org/10.1214/aoms/1177729893.

J. Skilling. Nested sampling for general bayesian computation. *Bayesian Analysis*, 1:833–860, 2006.

T. Steininger, J. Dixit, P. Frank, M. Greiner, S. Hutschenreuter, J. Knollmüller, R. Leike, N. Porqueres, D. Pumpe, M. Reinecke, M. Šraml, C. Varady, and T. Enßlin. NIFTy 3 - Numerical Information Field Theory - A Python framework for multicomponent signal inference on HPC clusters. *ArXiv e-prints*, August 2017.

T. Steininger, T. A. Enßlin, M. Greiner, T. Jaffe, E. van der Velden, J. Wang, M. Haverkorn, J. R. Hörandel, J. Jasche, and J. P. Rachen. Inferring Galactic magnetic field model parameters using IMAGINE - An Interstellar MAGnetic field INference Engine. *ArXiv e-prints*, January 2018.

Theo Steininger, Maksim Greiner, Frederik Beaujean, and Torsten Enßlin. d2o: a distributed data object for parallel high-performance computing in python. *Journal of Big Data*, 3(1):17, Sep 2016. ISSN 2196-1115. doi: 10.1186/s40537-016-0052-5. URL https://doi.org/10.1186/s40537-016-0052-5.

Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. The top500 project, 2015. URL http://www.top500.org/lists/2015/11/.

J. H. Taylor and J. M. Cordes. Pulsar distances and the galactic distribution of free electrons. *ApJ*, 411:674–684, July 1993. doi: 10.1086/172870.

MPICH Team. Mpich2: High-performance portable mpi, 2016a. URL www.mcs.anl.gov/mpich2.

ScaLAPACK Team. Scalapack web page, 2016b. URL www.netlib.org/scalapack/.

Excellence Cluster Universe. Excellence cluster universe, 2016. URL http://www.universe-cluster.de/c2pap.

V. Vacca, N. Oppermann, T. Enßlin, J. Jasche, M. Selig, M. Greiner, H. Junklewitz, M. Reinecke, M. Brüggen, E. Carretti, L. Feretti, C. Ferrari, C. A. Hales, C. Horellou, S. Ideguchi, M. Johnston-Hollitt, R. F. Pizzo, H. Röttgering, T. W. Shimwell, and K. Takahashi. Using rotation measure grids to detect cosmological magnetic fields: A Bayesian approach. *A&A*, 591:A13, June 2016. doi: 10.1051/0004-6361/201527291.

Ellert van der Velden. Imagine: Testing a bayesian pipeline for galactic magnetic field model optimization. Master's thesis, Radboud University, Nijmegen, The Netherlands, 2017.

Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011. ISSN 1521-9615. doi: http://doi.ieeecomputersociety.org/10.1109/MCSE.2011.37.

A Waelkens, T Jaffe, M Reinecke, F S Kitaura, and T A Enßlin. Simulating polarized Galactic synchrotron emission at all frequencies. The Hammurabi code. *A&A*, 495(2):697–706, February 2009. doi: 10.1051/0004-6361:200810564. URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2009A%26A...495..697W&link_type=EJOURNAL.

G.S. Watson. *Statistics on spheres*. University of Arkansas lecture notes in the mathematical sciences. Wiley, 1983. ISBN 9780471888666. URL https://books.google.de/books?id=tBjvAAAAMAAJ.

Cornelius Weig and Torsten A. Enßlin. Bayesian analysis of spatially distorted cosmic signals from poissonian data. *Monthly Notices of the Royal Astronomical Society*, 409(4):1393–1411, 2010. doi: 10.1111/j.1365-2966.2010.17122.x. URL +http://dx.doi.org/10.1111/j.1365-2966.2010.17122.x.

D.C.B. Whittet. *Dust in the Galactic Environment, 2nd Edition*. Series in Astronomy and Astrophysics. Taylor & Francis, 2002. ISBN 9780750306249. URL https://books.google.de/books?id=k21lk4sORpEC.

N. Wiener. *Extrapolation, Interpolation and Smoothing of Stationary Time Series, with Engineering Applications*. Technology Press and Wiley, New York, 1949. note: Originally issued in Feb 1942 as a classified Nat. Defense Res. Council Rep.

M. Wolleben, T. L. Landecker, W. Reich, and R. Wielebinski. An absolutely calibrated survey of polarized emission from the northern sky at 1.4 GHz. Observations and data reduction. *A&A*, 448:411–424, March 2006. doi: 10.1051/0004-6361:20053851.

Max A. Woodbury. Inverting modified matrices. *Statistical Research Group, Princeton University, Princeton, N. J.*, Memo. Rep.(42):4pp, 1950.

J. M. Yao, R. N. Manchester, and N. Wang. A New Electron-density Model for Estimation of Pulsar and FRB Distances. *ApJ*, 835:29, January 2017. doi: 10.3847/1538-4357/835/1/29.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

E. G. Zweibel. The microphysics and macrophysics of cosmic raysa). *Physics of Plasmas*, 20 (5):055501, May 2013. doi: 10.1063/1.4807033.

F. Zwicky. On the Masses of Nebulae and of Clusters of Nebulae. *ApJ*, 86:217, October 1937. doi: 10.1086/143864.

Danksagung

Mein größter Dank gilt Torsten Enßlin. Als Betreuer hat er sich in einem Maß für diese Arbeit und mich eingesetzt, das weit über das Selbstverständliche hinaus geht. Sowohl seine Art der wissenschaftlichen Arbeit, als auch der Führungsstil mit dem er seine Arbeitsgruppe leitet, sind nach wie vor für mich inspirierend und haben mir optimale Arbeitsbedingungen geschaffen. Fachlich wie persönlich konnte in den letzten Jahren vieles von ihm lernen; sein Einfluss hat mich dauerhaft geprägt. Seine Kultur der hilfsbereiten Zusammenarbeit spiegelt sich in der Arbeitsgruppe wider und so wurde ich von meinen Kollegen herzlich aufgenommen und insbesondere in der Anfangszeit bestens unterstützt. Hier gilt mein Dank Vanessa Böhm, Sebastian Dorn, Niels Oppermann, Martin Reinecke, Marco Selig, Valentina Vacca und ganz besonders Maksim Greiner. Auch mit meinen späteren Kollegen Philipp Frank, Sebastian Hutschenreuter, Jakob Knollmüller, Reimar Leike, Natalia Porqueres, Daniel Pumpe und Matevž Šraml war es eine Freude zusammenzuarbeiten und ich konnte vieles von ihnen lernen. Außerdem danke ich auch Frederik Beaujean und Jens Jasche (Exzellenzcluster Universe), Marjike Haverkorn (Radboud-Universität Nijmegen) und ganz besonders Tess Jaffe (NASA Goddard Space Flight Center), Ellert van der Velden (Swinburne University of Technology) und Jiaxin Wang (SISSA) für ihre überragende Unterstützung. Im Rahmen dieser Arbeit hatte ich die Ehre eine Reihe von Studenten mitbetreuen zu dürfen. Dabei danke ich Adnan Akhundov, Mihai Baltac, Jait Dixit, Maximilian Kurthen, Stephan Rabanser, Csongor Várady und Sebastian Weiß für die vielen tollen Stunden der intensiven Zusammenarbeit. Ferner danke ich Eiichiro Kumatsu und Thorsten Naab für die Unterstützung als Mitglieder in meinem PhD committee.
Mein besonderer Dank gilt meiner Familie, die mich über die gesamte Zeit dieser Arbeit unterstützt hat. Ich bin unendlich dankbar, euch als Eltern, Schwestern, Kinder und zur Frau zu haben.