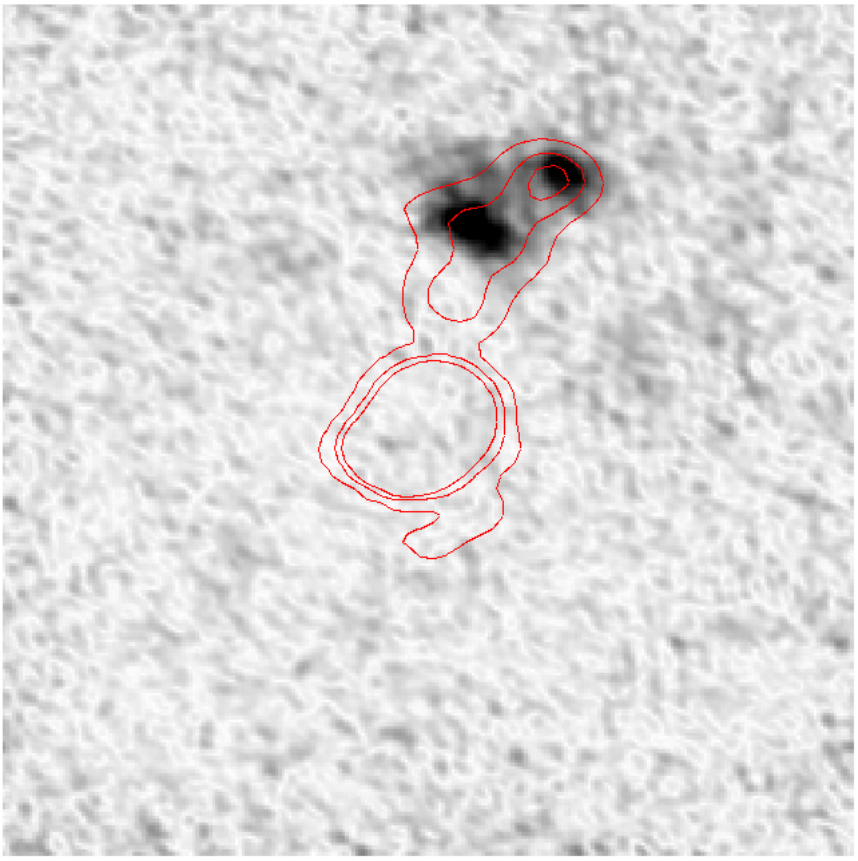
CHALMERS
UNIVERSITY OF TECHNOLOGY



Probing galaxy halos using background polarized radio sources

Master's thesis in Physics and Astronomy

ANTON NILSSON

Department of Earth and Space Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Probing galaxy halos using background polarized radio sources

Anton Nilsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Earth and Space Sciences
Radio Astronomy and Astrophysics group
Chalmers University of Technology
Gothenburg, Sweden 2016

Probing galaxy halos using background polarized radio sources
Anton Nilsson

Supervisor: Cathy Horellou, Department of Earth and Space Sciences
Examiner: Cathy Horellou, Department of Earth and Space Sciences

Department of Earth and Space Sciences
Radio Astronomy and Astrophysics group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Polarized emission of the radio source J133920+464115 at Faraday depth $+20.75\,\mathrm{rad\,m^{-2}}$ in grayscale. The contours show the total intensity.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2016

Probing galaxy halos using background polarized radio sources
Anton Nilsson
Department of Earth and Space Sciences
Chalmers University of Technology

Abstract

Linearly polarized radio emission from synchrotron sources undergoes Faraday rotation when passing through a magneto-ionized medium. This might be used to search for evidence of ionized halos around galaxies, using polarized radio sources as a probe. In this thesis I analyze the polarization properties of background radio sources from the Taylor et al. (2009) catalog based on observations with the Very Large Array, and correlate these to the angular separation between the radio sources and foreground galaxies. I find a decrease in the amount of polarized radio sources near the galaxies, interpreting this as radio sources being depolarized by galaxy halos.

I also investigate searching for polarization in data from the Low Frequency Array (LOFAR). To this end, I reanalyze the observation of the M51 field by Mulcahy et al. (2014), focusing on the background radio sources in the field. I find several polarized sources, many of which contain complex polarization.

Keywords: galaxy halos, RM synthesis, M51, polarization

Acknowledgements

There are several people who have helped me during this journey, and to whom I owe great thanks:

- First and foremost, I would like to thank my supervisor Cathy Horellou for giving me the chance to work with such an interesting subject. Your guidance and knowledge have been invaluable during the entire process.
- I wish to thank Stephen Bourke for his help with imaging and the software.
- Many thanks go to Andreas Horneffer and David Mulcahy for making the LOFAR dataset of the M51 field available to me, without which the second half of this thesis would have been impossible.
- I want to thank Tobia Carozzi, whose code I did not use in the end, but who provided valuable insight into primary beam correction.
- I also thank David Mulcahy, Anna Scaife, Alex Clarke and Therese Cantwell for organizing the annual meeting of the LOFAR Magnetism Key Science Project in Manchester. You, as well as the other participants, gave me a great experience.
- I am grateful for the financial support from the Swedish National Facility for Radio Astronomy that allowed me to participate in said meeting.
- I would like to thank Henrik Junklewitz for good advice about the parameters for fsclean, which also helped me choose the parameters for CLEAN.
- Finally, I would like to thank my fiancée Sigrid for her unwavering support during the most difficult times. I am lucky to be with you.

Additionally, several tools have also been of help to me:

- This research has made use of the NASA/IPAC Extragalactic Database (NED) which is operated by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.
- This research has made use of the VizieR catalogue access tool, CDS, Strasbourg, France. The original description of the VizieR service was published in A&AS 143, 23
- This work used the astronomy & astrophysics package for Matlab (Ofek 2014).

Anton Nilsson, Gothenburg, Nov 2016

Contents

List of Figures

List of Tables

1
Introduction

There is increasing observational evidence that normal galaxies are surrounded by large halos of ionized matter. Studying the properties of these halos will tell us much about the formation of galaxies. It will also provide insight into the missing baryon problem, where the greatest part of the number of baryons in the Universe is still undetected (Hernández-Monteagudo et al. 2015). As the halos do not emit enough radiation for direct observation they have to be studied by indirect methods. The absorption lines of background quasars, with lines of sight passing near foreground galaxies, have been used:

- Mg II absorption has been observed to correlate with the stellar mass of the galaxy and its specific star formation rate (Chen et al. 2010).
- In a sample of 195 galaxies at redshift $z < 0.176$, hydrogen gas has been found out to a distance of 500 kpc (Liang and Chen 2014).
- The volume density of galactic halos has been found to scale with distance from the galaxy as $r^{-0.8\pm0.3}$, from observations of the regions around 44 $z < 0.35$ galaxies (Werk et al. 2014).
- Evidence for a halo has been found around the Andromeda galaxy by analyzing the absorption spectra of 18 quasars (Lehner et al. 2015).

It is also of interest to examine the rotation measure[1] (RM) of background polarized radio sources. RM is related to the density of ionized gas and the magnetic field along the line of sight.

5 GHz and ultraviolet observations of 77 quasars show Mg II absorption to correlate with RM (Bernet, Miniati, and Lilly 2010; Bernet, Miniati, Lilly, et al. 2008). Comparison with optical data shows that the large-RM quasars have sightlines passing within 50 kpc of a galaxy (Bernet, Miniati, and Lilly 2013).

The correlation of Mg II absorption with RM does not appear at 1.4 GHz (Bernet, Miniati, and Lilly 2012), except for quasars with flat spectral index (Farnes, O'Sullivan, et al. 2014). Two hypotheses for this lack of correlation are that it is due to the increased depolarization at 1.4 GHz (Bernet, Miniati, and Lilly 2012), and that different locations in the source are probed at different wavelengths (Farnes, O'Sullivan, et al. 2014).

These results indicate that RM is a feasible probe of galactic halos, though perhaps less so at longer wavelengths. Another possible probe is the degree of polarization.[2] A clear example is the case of the giant radio galaxy Fornax A (Fomalont et al. 1989). NGC 1310 is a spiral galaxy somewhat smaller than the Milky Way, located in front of one of the radio lobes of Fornax A. The radio lobe is strongly polarized,

[1] The concept of rotation measure is introduced in Section 1.2.1.

[2] Degree of polarization is introduced in Section 1.2

but a silhouette of the intervening galaxy is visible as an area of low polarization at 1.4 GHz. This depolarized region does not extend much past the galaxy itself (Schulman and Fomalont 1992). However, the HI disk of NGC 1310 is smaller than usual, possibly due to ram pressure stripping (Horellou et al. 2001). In that case any larger halo may also have been stripped. Similarly to the method used in part of this study, Bonafede et al. (2011) have analyzed the degree of polarization of background sources behind massive galaxy clusters. They found a lower degree of polarization for those sources compared to the ones at larger radii, and interpret this as depolarization by the intervening intracluster medium.

In this study I search for statistical differences in the polarization of background sources, and investigate the polarization properties of individual background radio sources using rotation measure (RM) synthesis.[3]

This chapter briefly introduces some background concepts required to understand what I have done in this study.

In Chapter 2 I analyze the polarized sources detected by Taylor et al. (2009) and look for systematic variations in RM, degree of polarization, and source density depending on their proximity in the sky to galaxies.

Chapter 3 deals with a reanalysis of the LOFAR observation of the M51 field previously published by Mulcahy et al. (2014), this time focusing on the background radio sources in the field.

1.1 Important concepts in radio astronomy

In this section I will introduce only the most important concepts. *Tools of Radio Astronomy* (Wilson et al. 2009), from which the information in this section (though not Section 1.1.1) is taken, goes into more detail on the subject.

Traditional radio telescopes have a dish to collect and focus radiation, and an antenna to convert it into an electrical signal so it can be detected. To observe different areas of the sky the telescope is mechanically pointed in different directions.

While the telescope used in the second part of this study, LOFAR, is not a traditional radio telescope, it is useful to consider those first. I will describe LOFAR in Section 3.1.

An important property of a radio telescope is the **primary beam**, the sensitivity of the telescope as a function of direction (relative to the direction in which the telescope is pointing, if applicable). It is more sensitive toward the center, decreasing as you go farther out. Even farther out, it generally increases again, with some oscillations. These peaks are known as **sidelobes**.

[3] RM synthesis is introduced in Section A.2

A related concept is the **resolution** of the telescope, the angular distance between two features where they can still be distinguished. The theoretical diffraction-limited resolution is given by

$$\theta = 1.22\frac{\lambda}{D} \tag{1.1}$$

where λ is the wavelength of the observed radiation, and D is the diameter of the telescope.

An important quantity is the spectral density of flux (or **flux density**) S_ν. The flux is defined as the power reaching the observer per collecting area, and the flux density is the flux per unit frequency. It is often measured in Jansky, with $1\,\mathrm{Jy} = 10^{-26}\,\mathrm{W\,m^{-2}\,Hz^{-1}}$.

The **specific intensity**, the flux density per solid angle, is also useful. It can be expressed as Jy/sr, Jy/pix (in a digital image) or Jy/beam (i.e. Jansky per beam area).

1.1.1 Radio synchrotron emission

While there are several mechanisms by which radio waves can be emitted in an astronomical context, the dominant source of polarized radio emission is **synchrotron radiation**. This phenomenon is explained in more detail in *Galactic and Intergalactic Magnetic Fields* (Klein and Fletcher 2015), from which the information in this subsection is taken.

Synchrotron radiation is emitted when relativistic electrons move in a magnetic field. The force exerted on them by the field is perpendicular to both their velocity and the field direction, and so will cause them to spiral along the field lines. An accelerating charge emits radiation, and radiation produced in this manner is what is known as synchrotron radiation.

The specific intensity of synchrotron radiation depends on the magnetic field and the energy spectrum of the electrons. From observation of cosmic rays on Earth we know that the energy spectrum in the Milky Way follows a power law:

$$N(E)\,\mathrm{d}E \propto E^{-g}\,\mathrm{d}E \tag{1.2}$$

with $g = 2.4$. Given this energy distribution, the specific intensity is given by

$$I_\nu \propto B_\perp^{1-\alpha} \cdot \nu^\alpha \tag{1.3}$$

where $B_\perp$ is the magnetic field component perpendicular to the line of sight and α is the **spectral index**:

$$\alpha = \frac{1-g}{2} \tag{1.4}$$

For $g = 2.4$ the spectral index becomes $\alpha = -0.7$. The concept of spectral index is not unique to synchrotron radiation. More generally it is the exponent of ν when I_ν follows a power law distribution.

Synchrotron radiation is highly (linearly) polarized, in a direction perpendicular to the magnetic field.

The strongest radio synchrotron sources are radio-loud **quasars** (also known as quasi-stellar objects, or **QSOs**) and **radio galaxies**. Quasars are very distant, unresolved sources. They are thought to be the regions around supermassive black holes in the centers of galaxies. The radiation is produced by material being heated as it is falling into the accretion disk.

These supermassive black holes also produce jets, directed perpendicular to the accretion disk. At the termination of these jets the material spills out into enormous lobes. These lobes (called "**radio lobes**" if emitting in the radio) are polarized and are where most of the radio emission is located in radio galaxies, though it is also possible to detect polarization from the jets themselves.

1.2 Polarized emission

The subject of polarized emission is dealt with more comprehensively in *Tools of Radio Astronomy* (Wilson et al. 2009), from which the information in this section (though not the subsections) is taken.

Figure 1.1 shows the ellipse followed by the electrical component vector of elliptically polarized radiation. Instead of expressing polarization in E_a, E_b, ϕ and χ as in Figure 1.1, the quantities used in this thesis will be the four Stokes parameters:

- $I = E_a^2 + E_b^2$, the total intensity of the light.
- $Q = I\cos(2\phi)\cos(2\chi)$, one component of linear polarization.
- $U = I\cos(2\phi)\sin(2\chi)$, the other component of linear polarization.
- $V = I\sin(2\phi)$, the circular polarization.

As synchrotron radiation is linearly polarized, I will henceforth ignore circular polarization and assume $V = 0$. Then, the polarization can be written as a complex number:

$$P = Q + iU = pIe^{2i\chi} \tag{1.5}$$

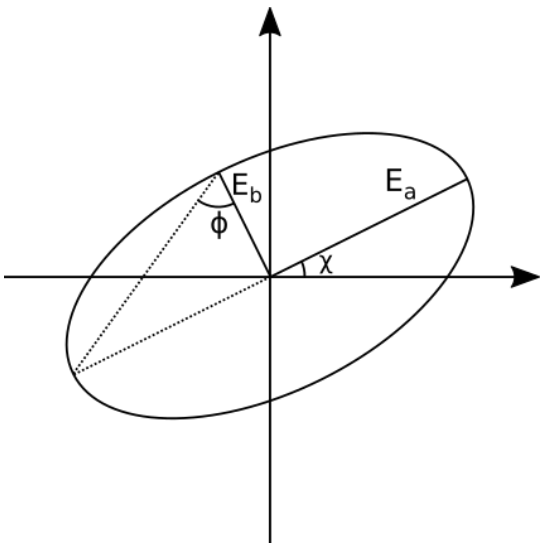


Figure 1.1: The ellipse followed by the E-vector in elliptically polarized radiation.

In addition, other quantities are often used:

- $PI = |P| = \sqrt{Q^2 + U^2}$, the total polarized intensity.
- $p = PI/I$, the degree of polarization.
- $\chi = \frac{1}{2}\arctan(\frac{U}{Q})$, the angle of polarization.

For synchrotron radiation $p = 72\,\%$ when it is emitted, but various forms of depolarization (to be introduced in Section 1.2.2) reduce the observed degree of polarization.

1.2.1 Faraday rotation

The concept of Faraday rotation is explained in more detail in *Galactic and Intergalactic Magnetic Fields* (Klein and Fletcher 2015), from which the information in this subsection is taken.

Some media, including magnetized ionized plasma, are circularly birefringent. This means that the speed of circularly polarized light (or other electromagnetic radiation) is different depending on whether the wave is polarized in a clockwise or counterclockwise direction.

Linearly polarized electromagnetic waves can be expressed as the sum of two circularly polarized components, one clockwise and one counterclockwise. The polarization angle depends on the relative phases of these components. As they move at different velocities their relative phase will change, and the polarization will rotate as the light passes through the medium. This is what is known as **Faraday rotation**, and is illustrated in Figure 1.2.

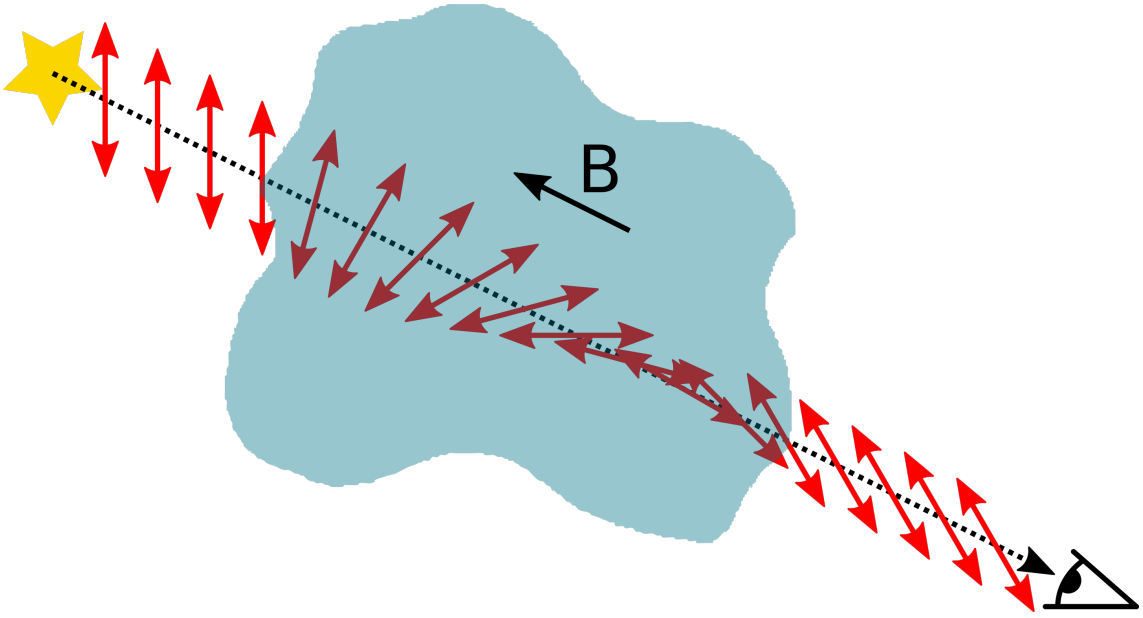


Figure 1.2: An illustration of linearly polarized emission (dashed line) undergoing Faraday rotation when passing through a magneto-ionized medium (blue), resulting in a change of polarization angle (red arrows).

After passing through magnetized ionized plasma, the polarization angle is given by

$$\chi = \chi_0 + K \int_0^r n_e(s) B_\parallel(s) \lambda^2 \, \mathrm{d}s \tag{1.6}$$

where χ_0 is the initial angle, r is the distance to the source, $n_e(s)$ is the electron density as a function of the distance along the line of sight, and $B_\parallel(s)$ is the magnetic field component along the line of sight (with direction toward us), as a function of the line of sight. K is a constant given by

$$K = 0.81 \, \frac{\mathrm{rad \, m^{-2}}}{\mathrm{cm^{-3} \, \mu G \, pc}} \tag{1.7}$$

We can express Equation 1.6 as

$$\chi = \chi_0 + \phi(r) \lambda^2 \tag{1.8}$$

where

$$\phi(r) = K \int_0^r n_e(s) B_\parallel(s) \, \mathrm{d}s \tag{1.9}$$

ϕ is called the **Faraday depth** of the source and is given in $\mathrm{rad \, m^{-2}}$. It is important to note that there is not a one-to-one correspondence between distance and Faraday depth. The magnetic field may point in any direction, and so the Faraday depth may increase or decrease any number of times along the line of sight.

A related concept is the **rotation measure**, or RM, defined as the change in polarization angle χ per unit λ^2. It is measured in the same units as the Faraday depth. Observationally, an estimate of RM can be obtained by observing χ at several wavelengths and making a linear fit of χ versus λ^2. RM is the slope of the line. As few as two wavelengths can be used, though in that case one has to deal with $n\pi$-ambiguities as the light may have been rotated several times. If the polarization of a radio source is confined to a single Faraday depth, the values of ϕ and RM are the same.

In more complex cases, we may define the **Faraday dispersion function** $F(\phi)$ as the polarization (on complex form) emitted per Faraday depth. The observed polarization for a Faraday complex source is:

$$P(\lambda^2) = \int_{-\infty}^{\infty} F(\phi) e^{2i\phi\lambda^2} \, \mathrm{d}\phi \tag{1.10}$$

1.2.2 Depolarization

When a polarized source is observed, several effects may have caused the observed degree of polarization to be lower than what was emitted. This is known as **depolarization**. When observing a source, the observed polarization vector will be the sum of several components. If these contributions should happen to be differently aligned, the magnitude of the resulting polarization vector may be smaller than that of its components, as seen in Figure 1.3.

The ways depolarization can occur are **bandwidth depolarization**, **beam depolarization** and **Faraday depolarization**.

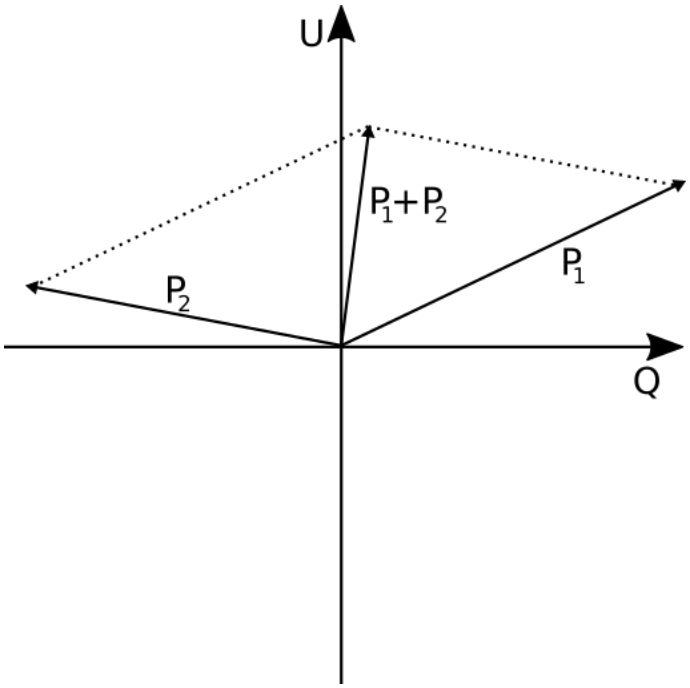


Figure 1.3: An illustration of depolarization. Observe that the resulting polarization vector $P_1 + P_2$ is smaller than either component P_1 or P_2.

1.2.2.1 Bandwidth depolarization

If a frequency channel is wide enough, or if many channels are averaged, the polarization angle may vary sufficiently within the channel to introduce depolarization. This affects high RM and long wavelengths more.

1.2.2.2 Beam depolarization

Beam depolarization is when the differing polarization components are along different lines of sight, but still within the telescope beam. This can occur for various reasons:

- Different parts of the source can be polarized in different directions, due to a tangled magnetic field.
- There can be several independent sources within the telescope beam.
- There can be a non-uniform Faraday rotating screen in front of the source. As a result, different polarized regions in the source will be Faraday-rotated by different amounts.

In the last case depolarization will be stronger at longer wavelengths, while in the first two it will be wavelength-independent. All types of beam depolarization can be reduced by increasing the resolution sufficiently.

Burn (1966) modeled depolarization due to a randomly fluctuating foreground screen, modeled as a great number of cells much smaller than the extent of the source, as:

$$p(\lambda^2) = p_i e^{-2\sigma^2\lambda^4} \tag{1.11}$$

where p_i is the initial degree of polarization and σ^2 is the variance of the RM contribution of the cells. He used this to argue that any depolarization from the Milky Way would be insignificant, but did not consider depolarization from intervening galaxies.

1.2.2.3 Faraday depolarization

Consider a structure that is both emitting and Faraday rotating. If it is extended along the line of sight, an observer will see contributions from all of its Faraday depths. Even if the polarization is coherent at different depths, the fact that the light will be rotated means that emission from different depths will be differently polarized, and as such depolarization will occur. This effect is stronger at longer wavelengths.

Burn (1966) modeled Faraday depolarization in a uniform slab with thickness $\Delta\phi$ in Faraday space, and showed that the degree of polarization varies with λ^2 as:

$$p(\lambda^2) = p_i \frac{\sin(\Delta\phi\,\lambda^2)}{\Delta\phi\,\lambda^2} \tag{1.12}$$

2

Searching for galaxy halos through the polarization of background radio sources

In this chapter I analyze the polarized radio sources from the Taylor et al. (2009) catalog to search for evidence of a circumgalactic magneto-ionized medium around nearby galaxies. I look for a correlation between angular distance from foreground galaxies and either degree of polarization, rotation measure or number density of radio sources.

2.1 The method

The catalogs I use are listed in Tables 2.1 and 2.2. To match the catalogs to each other I use MATLAB. The code, and more technical information, can be found in Appendix B.1.

I remove background sources with:

- Galactic latitude closer than 20 ^{o} to the galactic plane, to minimize effects from the Milky Way.
- degree of polarization larger than 20 %, to avoid a few strong sources skewing the sample.
- total intensity larger than 1 Jy, for the same reason.

This leaves 26 721 polarized radio sources.

I use the work of Oppermann et al. (2015) to obtain the reconstructed rotation measures (hereafter RRM) of the radio sources, which is the RM with the Galactic foreground removed. Because of the uncertainties it is not sufficient to remove the value in their Galactic foreground map from the source RM. Therefore they provide a catalog of sources (including those from the Taylor et al. (2009) catalog) where they have calculated the RRM probability distribution for each source. They include 1 000 samples for each source, following the probability distribution. For the RRM of a source I take the average of these samples, and for |RRM| I take the absolute value of each sample before averaging.

Table 2.1: Catalogs used to determine background radio source properties.

Catalog	Contains	Rows	Main selection criteria
NVSS (Condon et al. 1998)[a]	Radio sources of the whole sky above -40° declination, at 20 cm with $45''$ resolution, observed with the Very Large Array.	1 773 484	$S \gtrsim 2.5$ mJy
Taylor et al. (2009)[b]	RMs calculated between 18 and 20 cm of polarized sources from NVSS data.	37 543	$P > 8\times$ local rms noise
Oppermann et al. (2015)[c]	Reconstructed RMs of (among others) Taylor sources, removing Galactic foreground.	41 632	N/A

[a] The NRAO (National Radio Astronomy Observatory) VLA (Very Large Array) Sky Survey: `http://vizier.u-strasbg.fr/viz-bin/VizieR?-source=VIII%2F65`

[b] `http://vizier.u-strasbg.fr/viz-bin/VizieR?-source=J%2FApJ%2F702%2F1230`

[c] `http://wwwmpa.mpa-garching.mpg.de/ift/faraday/2014/index.html`

Table 2.2: Samples used for the foreground galaxies.

Sample	Contains	Rows	Main selection criteria
SINGS (Kennicutt et al. 2003)[a]	Nearby galaxies.	75	Chosen to include a wide range of properties.
RC3 (de Vaucouleurs et al. 1995)[b]	Galaxies bright in the sky.	23 011	Diameter $> 1'$, B-band magnitude < 15.5 and redshift $z < 0.05$, but contains galaxies not meeting criteria.

[a] The Spitzer Infrared Nearby Galaxies Survey: `http://vizier.u-strasbg.fr/viz-bin/VizieR-3?-source=J/ApJ/703/1569/table1`

[b] The Third Reference Catalogue of Bright Galaxies: `http://vizier.u-strasbg.fr/viz-bin/VizieR?-source=VII%2F155`

I relate the polarized radio sources to galaxies in either the SINGS or RC3 catalog (The SINGS catalog was obtained from Muñoz-Mateos et al. (2009).). When using the galaxies from RC3 I exclude Andromeda and the Magellanic Clouds. Their angular extent is so large as to overwhelm all other galaxies. Galaxies in areas without sources (i.e. declination lower than $-40\,^{\circ}$ or Galactic latitude smaller than $20\,^{\circ}$) are excluded. This leaves 65 SINGS galaxies and 16 267 RC3 galaxies. Figure 2.1 contains the distributions of angular extents (given by D25, which is the diameter of the galaxy measured to the 25th magnitude per square arcsecond), distances (for SINGS) and redshifts (for RC3).

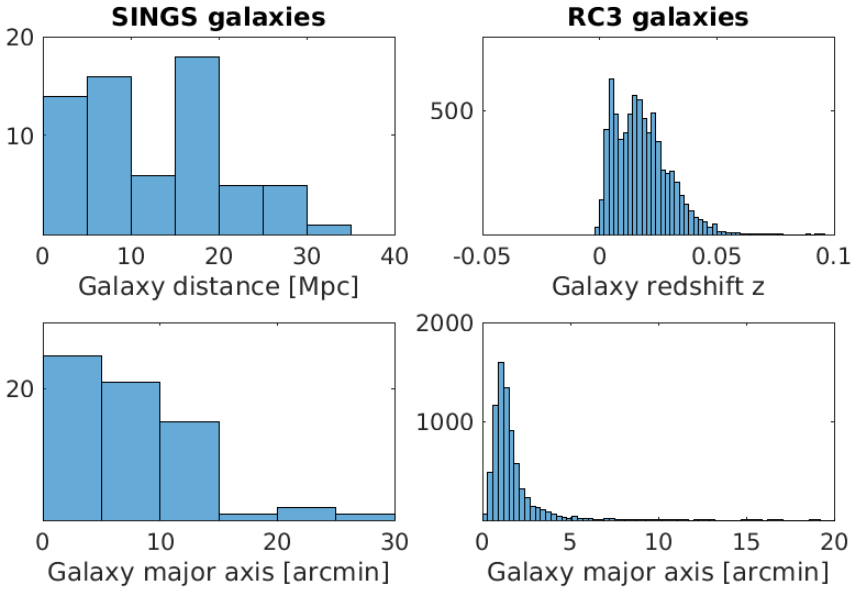


Figure 2.1: Galaxy angular extents and distances (SINGS) or redshifts (RC3) for the galaxy samples used. The number of RC3 galaxies with given redshifts is 7 500. Note that I have included several galaxies from RC3 with larger angular extent than is shown, of which the largest is $70.1\,'$.

To calculate the angular distance and angle I use the following equations derived from the spherical law of cosines:

$$\cos(\Delta r) = \sin(\mathrm{Dec}_{gal})\sin(\mathrm{Dec}_{src}) + \cos(\mathrm{Dec}_{gal})\cos(\mathrm{Dec}_{src})\cos(\Delta\mathrm{RA}) \tag{2.1}$$

$$\cos\theta = \frac{\sin(\mathrm{Dec}_{src}) - \cos(\Delta r)\sin(\mathrm{Dec}_{gal})}{\sin(\Delta r)\cos(\mathrm{Dec}_{gal})} \tag{2.2}$$

For each radio source and galaxy it is possible to define a r_{norm}, which is the angular distance between the radio source and the galaxy normalized against the angular extent of the galaxy. As there are two main possibilities regarding how ionized matter around galaxies might be distributed, I use two definitions of r_{norm}:

- Circular r_{norm} is the distance from the galaxy divided by (half) the galaxy major axis. If the ionized matter is distributed in a sphere this should give the clearest results.
- Elliptical r_{norm} is defined so that the source lies on an ellipse with the same center, orientation and proportions as the galaxy (D25), but scaled by a factor r_{norm}. If the ionized matter is distributed in a disk this definition should be better.

See Figure 2.2 for an illustration. If a source is near several galaxies I count it once for each galaxy.

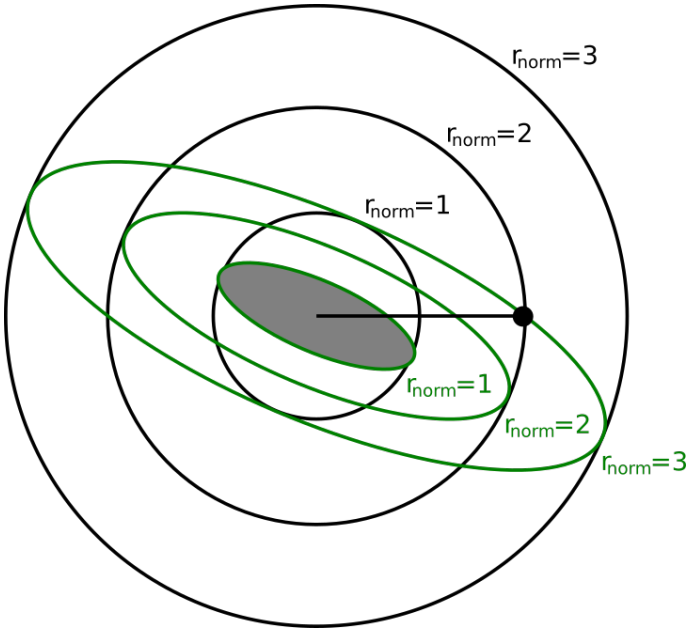


Figure 2.2: An illustration of the ways to define r_{norm}. The shaded area represents the galaxy. Black is circular r_{norm} and green is elliptical r_{norm}.

In addition to plotting the examined quantities, I do Monte Carlo simulations to see if the regions near the galaxies are unusual. To create a sample for comparison I take the observed galaxies and put them in random locations (and orientations) on the sky, avoiding those locations that would have been filtered out initially. For each sample I take the median of the examined quantities in a low r_{norm} range and a higher one, and compute the difference. For example:

$$\Delta p = p(0 < r_{norm} < 10) - p(10 < r_{norm} < 20) \tag{2.3}$$

I can then compare the same difference in the real data to this distribution. The reason for subtracting instead of only looking at the inner region is to avoid effects of where galaxies are likely to be located.

2.2 Results

In this section I present the results of the analysis. I first present a confounding apparent association between RC3 galaxies and bright polarized radio sources, and then investigate degree of polarization, the (absolute value of the) foreground-reconstructed rotation measure and the number of sources near SINGS and RC3 galaxies. Last, I look at the relationship of degree of polarization and RRM for the whole sky and near RC3 galaxies.

2.2.1 Association of RC3 galaxies with bright and polarized radio sources

Figure 2.3 shows the intensity, polarized intensity and source count as a function of r_{norm} for the RC3 galaxies. Notice the marked increases in I and PI, and the smaller decrease in source count, at low r_{norm}. This points to a correlation of RC3 galaxies and bright radio sources, as there is no plausible mechanism by which the galaxies would affect background sources in this way. No such association is seen for the SINGS galaxies.

When looking at the RC3 galaxies I include Monte Carlo simulations with the $4 < r_{norm} < 10$ as the inner region, to minimize the influence of this.

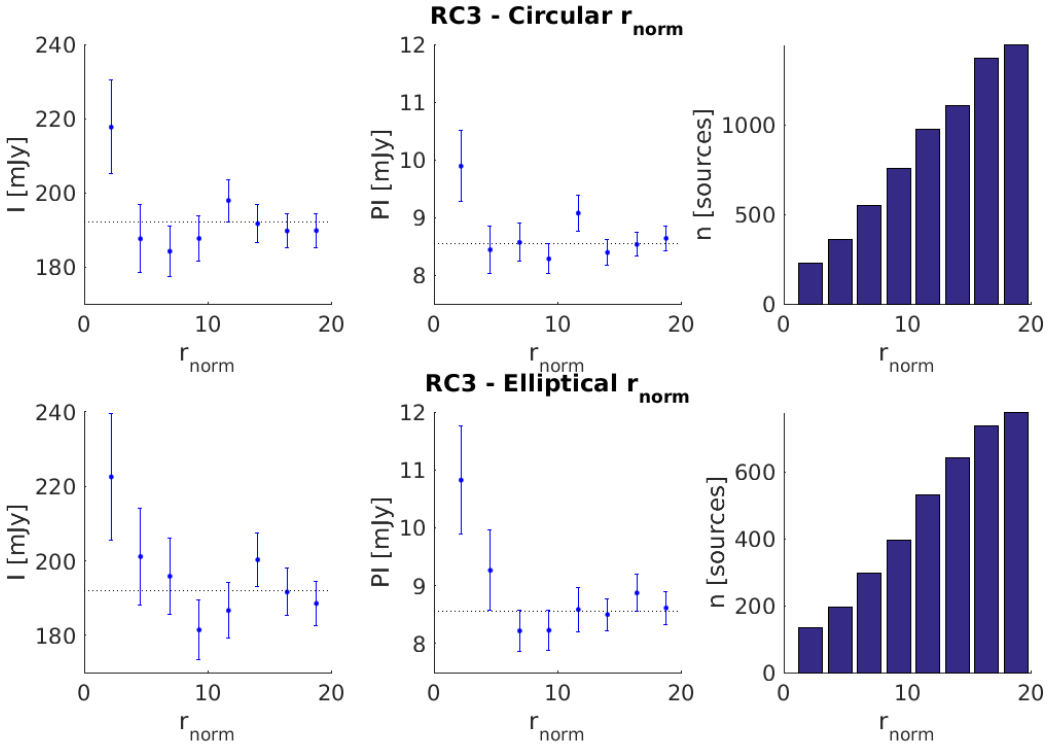


Figure 2.3: Total intensity, polarized intensity and number of sources as a function of r_{norm} for sources near galaxies in RC3. Note the increase in the first two quantities, and the smaller decrease in the third, at low r_{norm}. This indicates an association between RC3 galaxies and radio sources. Error bars are the standard errors.

2.2.2 Degree of polarization

When looking for galaxy halos the most obvious probe is the degree of polarization of background sources. One would expect the material in the halo to depolarize the sources, so the degree of polarization would be lower at low r_{norm}.

The degree of polarization of sources near SINGS galaxies is shown in Figure 2.4 as a function of r_{norm}. Also included are histograms of $\Delta p = p(0 < r_{norm} < 10) - p(10 < r_{norm} < 20)$ of 1000 Monte Carlo simulations (as described in Section 2.1), along with the same quantity for the real galaxies. No relationship can be seen for either circular or elliptical r_{norm}.

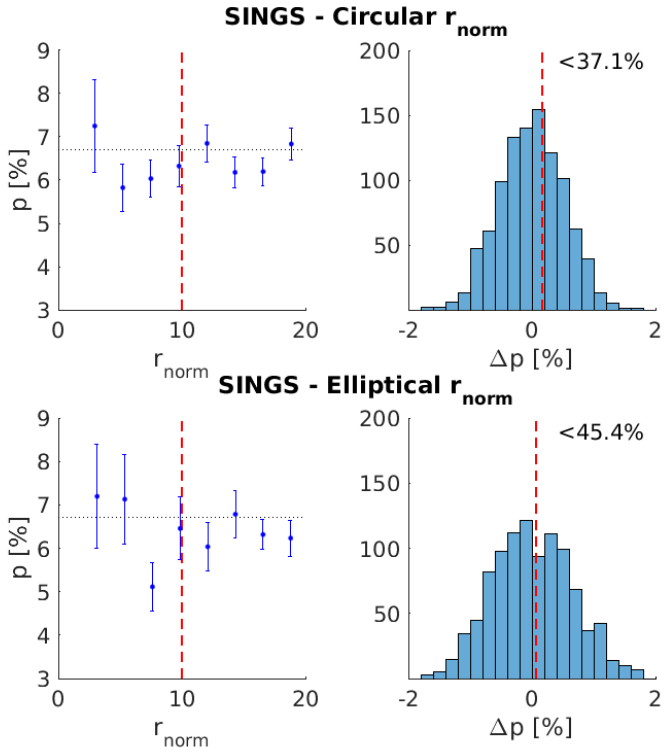


Figure 2.4: (left) Degree of polarization near SINGS galaxies. Error bars are the standard errors. The red line at $r_{norm} = 10$ is the border between the regions used in the Monte Carlo simulation. (right) The disribution of $p(0 < r_{norm} < 10) - p(10 < r_{norm} < 20)$ (median values) in the Monte Carlo simulations. The red line is the value for the real galaxies. The number to the upper right is the fraction of simulations where Δp is higher than in the real case.

Figure 2.5 shows the same plots for the RC3 galaxies. The degree of polarization within 10 r_{norm} is higher than outside.

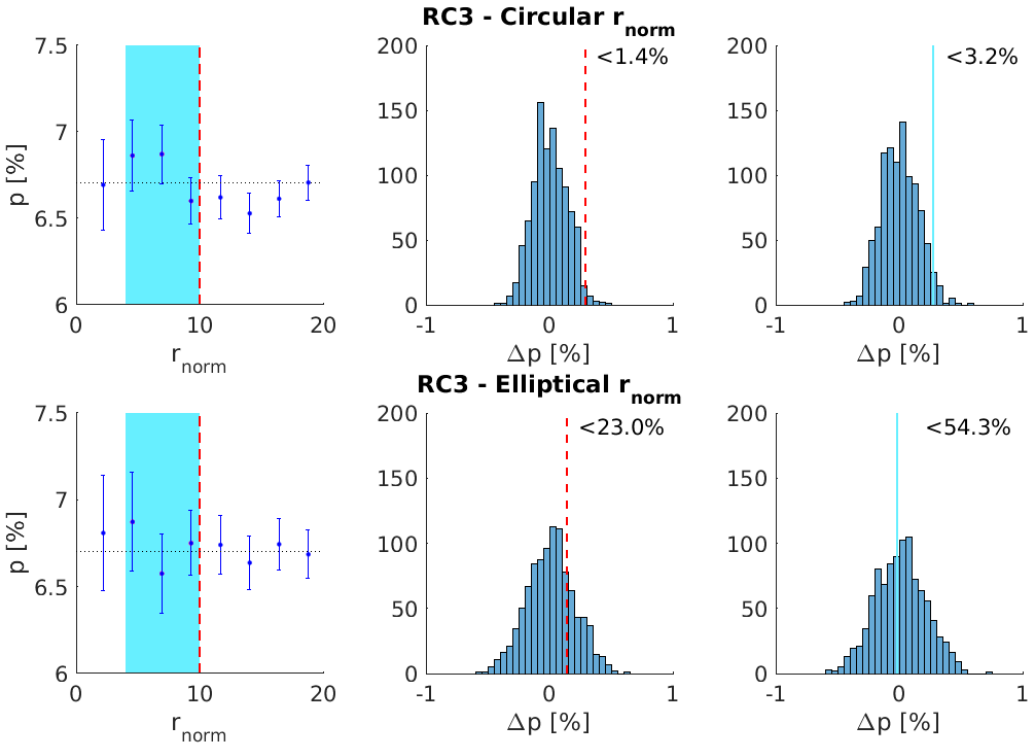


Figure 2.5: (left) Degree of polarization near RC3 galaxies. Error bars are the standard errors. The red line at $r_{norm} = 10$ is the border between the regions used in the left Monte Carlo simulation. The light blue area is the inner region used in the rightmost Monte Carlo simulation. (center, right) The disributions of $p(0 < r_{norm} < 10) - p(10 < r_{norm} < 20)$ and $p(4 < r_{norm} < 10) - p(10 < r_{norm} < 20)$ (median values), respectively, in the Monte Carlo simulations. The line is the value for the real galaxies. The number to the upper right is the fraction of simulations where Δp is higher than in the real case.

2.2.3 Rotation measure

Another possible probe is the (foreground-corrected) rotation measure of the sources. As positive and negative RM would otherwise cancel, I look at the absolute value. The results for the SINGS galaxies are seen in Figure 2.6. In the case of elliptical r_{norm} the Monte Carlo analysis differs greatly depending on the choice of boundary between the inside and outside regions, so two choices are included, 10 and 7 r_{norm}. The 7 r_{norm} case, as well as circular r_{norm}, show that sources in the regions close to galaxies tend to have low |RRM|.

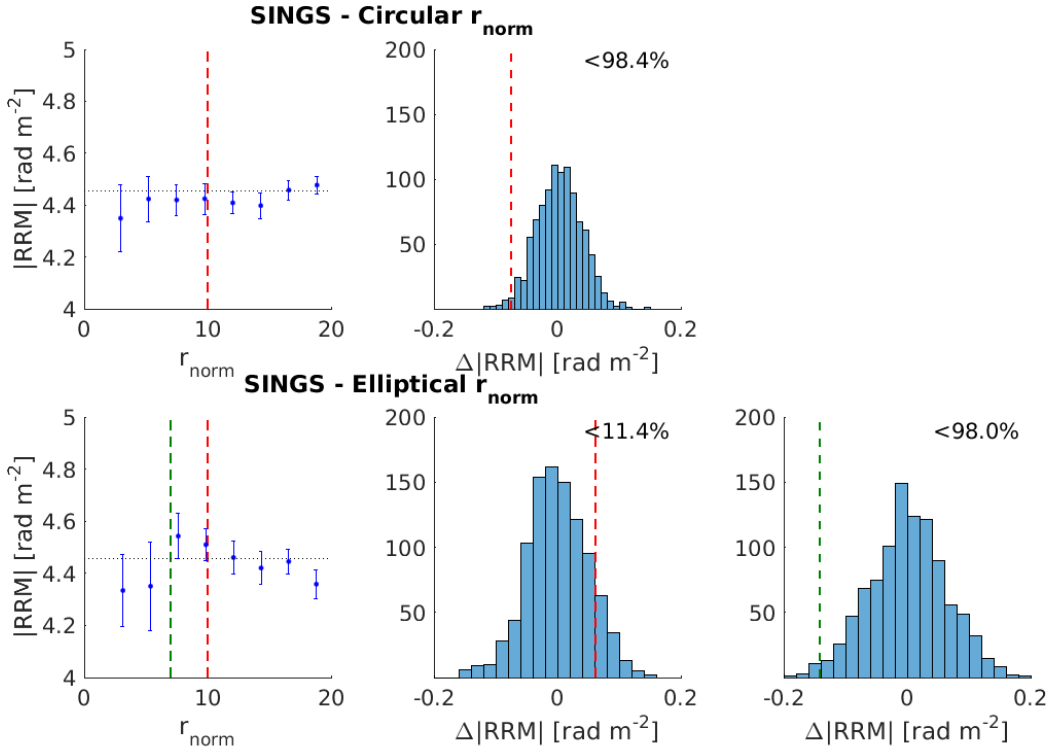


Figure 2.6: (left) Foreground-reconstructed rotation measure near SINGS galaxies. Error bars are the standard errors. The lines at $r_{norm} = 10$ and $r_{norm} = 7$ (for elliptical r_{norm}) are the borders between the regions used in the Monte Carlo simulations. (center, right) The disributions of $|\text{RRM}|(0 < r_{norm} < 10) - |\text{RRM}|(10 < r_{norm} < 20)$ and $|\text{RRM}|(0 < r_{norm} < 7) - |\text{RRM}|(7 < r_{norm} < 20)$ (median values), respectively, in the Monte Carlo simulations. The red line is the value for the real galaxies. The number to the upper right is the fraction of simulations where $\Delta|\text{RRM}|$ is higher than in the real case.

The same plots for the RC3 galaxies can be found in Figure 2.7. Here, the absolute RRM is higher near the galaxies when using elliptical r_{norm}, while no pattern appears with circular r_{norm}.

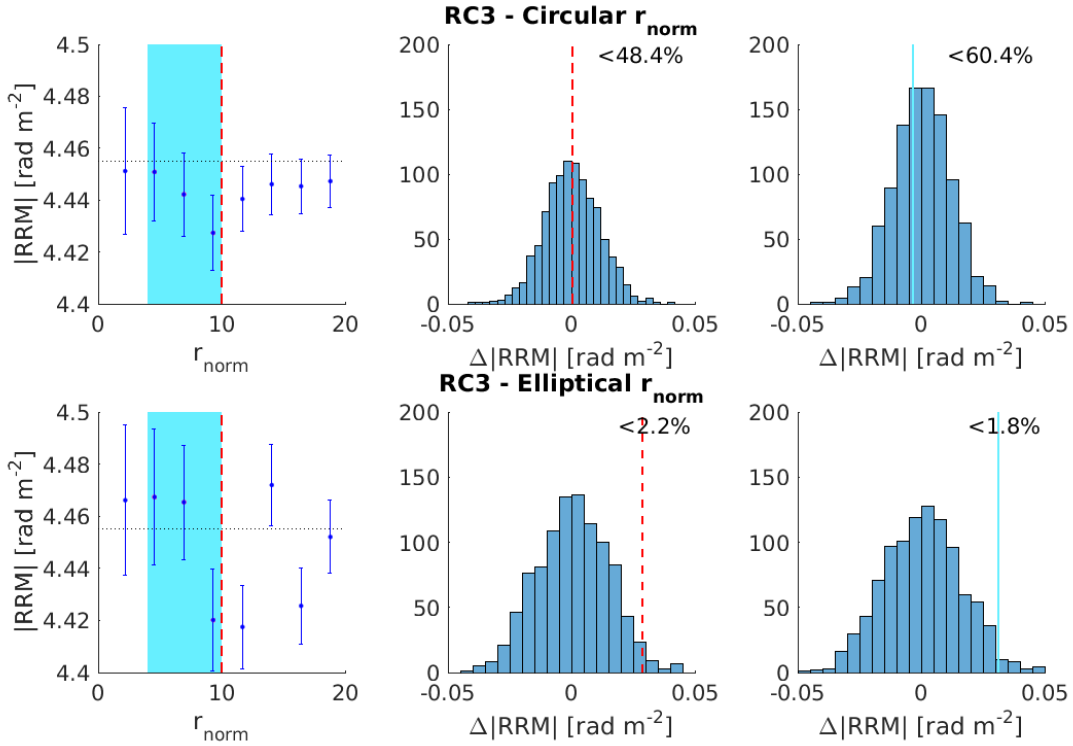


Figure 2.7: (left) Foreground-reconstructed rotation measure near RC3 galaxies. Error bars are the standard errors. The red line at $r_{norm} = 10$ is the border between the regions used in the left Monte Carlo simulation. The light blue area is the inner region used in the right Monte Carlo simulation. (center, right) The disributions of $|\mathrm{RRM}|(0 < r_{norm} < 10) - |\mathrm{RRM}|(10 < r_{norm} < 20)$ and $|\mathrm{RRM}|(4 < r_{norm} < 10) - |\mathrm{RRM}|(10 < r_{norm} < 20)$ (median values), respectively, in the Monte Carlo simulations. The line is the value for the real galaxies. The number to the upper right is the fraction of simulations where $\Delta|\mathrm{RRM}|$ is higher than in the real case.

2.2.4 Source count

A third probe is the amount of sources in a region of the sky. For the SINGS galaxies, this can be found in Figure 2.8. For both circular and elliptical r_{norm} there are markedly fewer sources near the galaxies than farther out. Choosing 13 as the r_{norm} boundary makes the difference even more pronounced.

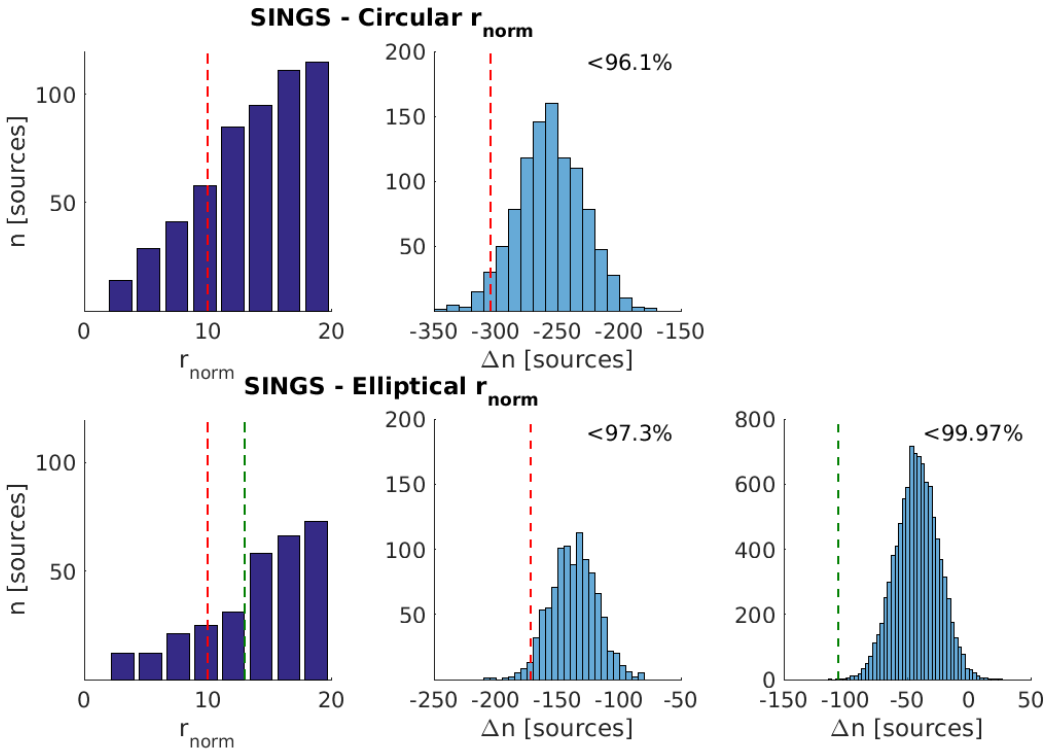


Figure 2.8: (left) Source count near SINGS galaxies. The lines at $r_{norm} = 10$ and $r_{norm} = 13$ (for elliptical r_{norm}) are the borders between the regions used in the Monte Carlo simulations. (center, right) The disributions of $n(0 < r_{norm} < 10) - n(10 < r_{norm} < 20)$ and $n(0 < r_{norm} < 13) - n(13 < r_{norm} < 20)$, respectively, in the Monte Carlo simulations. The red line is the value for the real galaxies. The number to the upper right is the fraction of simulations where Δn is higher than in the real case.

For RC3 galaxies the same plots are reproduced in Figure 2.9. In this case the number of sources closer to the galaxies tends to be lower, especially when ignoring the innermost region.

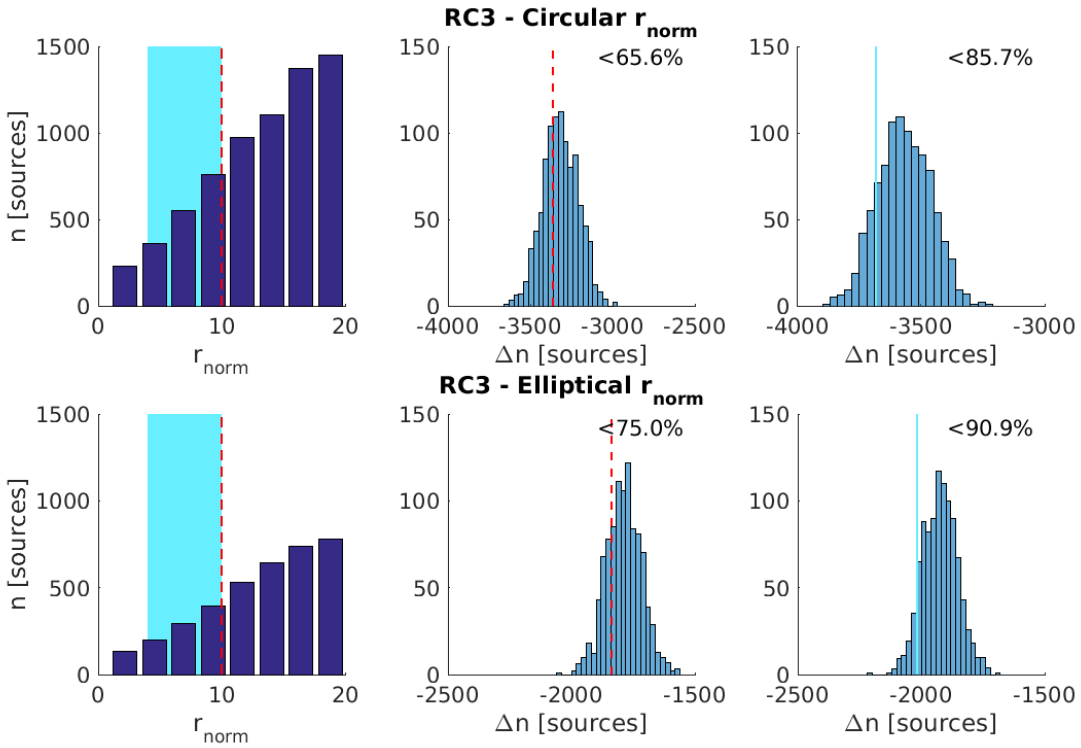


Figure 2.9: (left) Source count near RC3 galaxies. The red line at $r_{norm} = 10$ is the border between the regions used in the left Monte Carlo simulation. The light blue area is the inner region used in the right Monte Carlo simulation. (center, right) The disributions of $n(0 < r_{norm} < 10) - n(10 < r_{norm} < 20)$ and $n(4 < r_{norm} < 10) - n(10 < r_{norm} < 20)$, respectively, in the Monte Carlo simulations. The line is the value for the real galaxies. The number to the upper right is the fraction of simulations where Δn is higher than in the real case.

2.2.5 Relationship between polarization degree and rotation measure

Figure 2.10 shows degree of polarization plotted against RRM and |RRM|. p is higher near |RRM| = 0, but also at larger |RRM|. Interestingly, low |RRM| radio sources near RC3 galaxies seem to have a lower p than other low |RRM| radio sources.

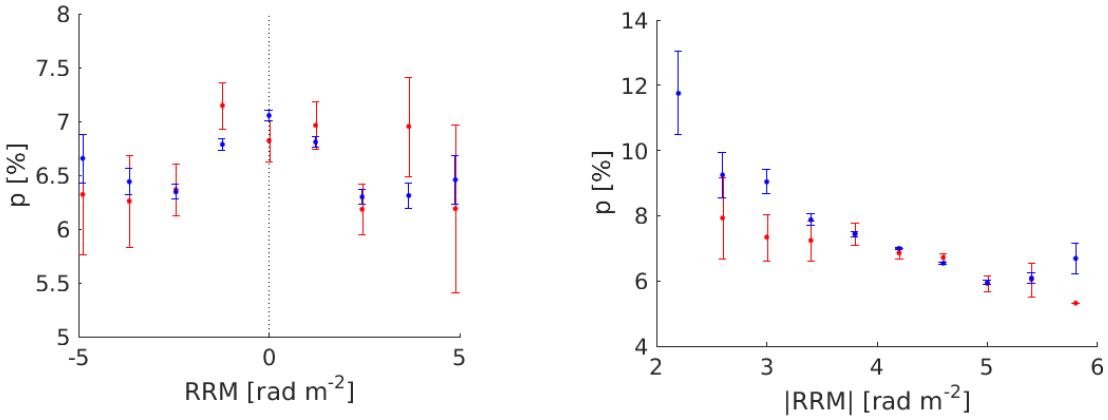


Figure 2.10: Degree of polarization plotted against RRM (left) and |RRM| (right), for the whole sky (blue) and for sources with $r_{norm} < 10$ with respect to RC3 galaxies (red). Error bars are the standard errors.

2.3 Discussion

The most striking result is the low number of polarized sources near SINGS galaxies (Figure 2.8), especially in the case of elliptical r_{norm}. I interpret this as sources being depolarized to below the threshold of the Taylor et al. (2009) catalog. While one might expect to see a decrease in the mean degree of polarization in that case, the fact that the least polarized sources are not included lessens this effect significantly. This might explain the lack of effect in degree of polarization (Figure 2.4).

The decrease in |RRM| near SINGS galaxies (Figure 2.6) is counterintuitive, but it might be due to the fact that large-RRM sources tend to have lower polarization. If these sources are depolarized out of the catalog disproportionally often it would lead to a smaller median RRM.

While the RC3 sample is much larger than SINGS, it suffers from some problems that need to be dealt with before it can be as useful for this analysis. The cause of the apparent association with strong radio sources needs to be found and corrected for, and the distribution of galaxy properties needs to be examined.

One possible explanation for the apparent RC3 association with strong sources (Figure 2.3) is that many of the galaxies belong to the same galaxy clusters as these sources. As RC3 contains predominantly bright galaxies, the associated sources would likely be closer and therefore brighter as well. Another explanation, supported

by the association only being visible at low r_{norm}, is that the strong radio sources might actually be radio lobes of the galaxies. Given the low number of sources that by chance happen to be at low r_{norm}, relatively few such radio lobes would be needed to produce a visible effect.

The same tendency of fewer sources near galaxies can be seen for the RC3 galaxies as well, but not as strongly (Figure 2.9). This might be due to the proportion of different galaxy types.

The large RRM near RC3 galaxies when using elliptical r_{norm} (Figure 2.7) seems to contradict the result for the SINGS galaxies. If both are real, a possible explanation might be that RC3 galaxies are more likely to have halos with a more regular magnetic field.

The degree of polarization around RC3 galaxies (Figure 2.5) for circular r_{norm} is larger than average inside of $10\,\mathrm{r_{norm}}$, and smaller just outside. Looking at Figure 2.3, this seems to be mostly an effect of the total intensity. It is possible that interference from the galaxies themselves affects the intensity of the sources. As the RC3 galaxies generally have smaller angular size than the SINGS galaxies, a certain r_{norm} corresponds to a smaller angular separation, making this interference more significant.

Sources with low |RRM| near RC3 galaxies seem to have a smaller degree of polarization. It is not clear why this effect does not extend to sources with higher |RRM|.

A way to find out whether the lower number of sources near galaxies is due to depolarization could be to look at the fraction of NVSS sources that are polarized. This would exclude an effect where there are fewer observed sources overall near galaxies.

The effects of galaxy type need to be examined. If the decrease in sources is stronger near late-type galaxies with more star formation, it would agree with the results of Chen et al. (2010) and strengthen the hypothesis that the effect is due to depolarization in halos.

Sensitive wide-band low-frequency telescopes like LOFAR (and the future SKA) offer new interesting possibilities for this type of study. I will discuss this more in Section 3.7.

3

Detecting polarization in LOFAR data using RM synthesis

In this chapter I explore the use of RM synthesis to examine polarized radio sources in data from the radio telescope array LOFAR. RM synthesis offers several advantages over a linear fit of the polarization angle, χ, versus λ^2, such as the ability to identify sources with several polarized components along the line of sight (Brentjens and de Bruyn 2005). Appendix A.2 contains an overview of RM synthesis. I image and apply RM synthesis to the data from the LOFAR observation of the M51 field by Mulcahy et al. (2014).

This chapter assumes some knowledge of aperture synthesis. Readers unfamiliar with this are directed to Appendix A.1.

3.1 The data

To search for polarization in background sources I used LOFAR data of the M51 field. Those data have been calibrated and results have been published by Mulcahy et al. (2014). I will describe the M51 field, introduce LOFAR, and describe the observation.

3.1.1 The M51 field

M51, also called the "Whirlpool Galaxy", is a face-on spiral galaxy and one of the most studied galaxies in the sky. The M51 field is shown in Figure 3.1.

Mulcahy et al. (2014) focused on radio emission from M51 itself, but did not observe polarization in the galaxy. They found 6 polarized background radio sources in the field.

Mao et al. (2015) observed M51 at $1-2\,\mathrm{GHz}$ using the Karl G. Jansky Very Large Array. They detected polarization from M51 at approximately $-9\,\mathrm{rad\,m^{-2}}$, and 11 polarized background sources.

Farnes, Green, et al. (2013) used the Giant Metrewave Radio Telescope to observe M51 at $602-618\,\mathrm{MHz}$. They did not observe any polarization in the galaxy, but found 15 polarized background sources.

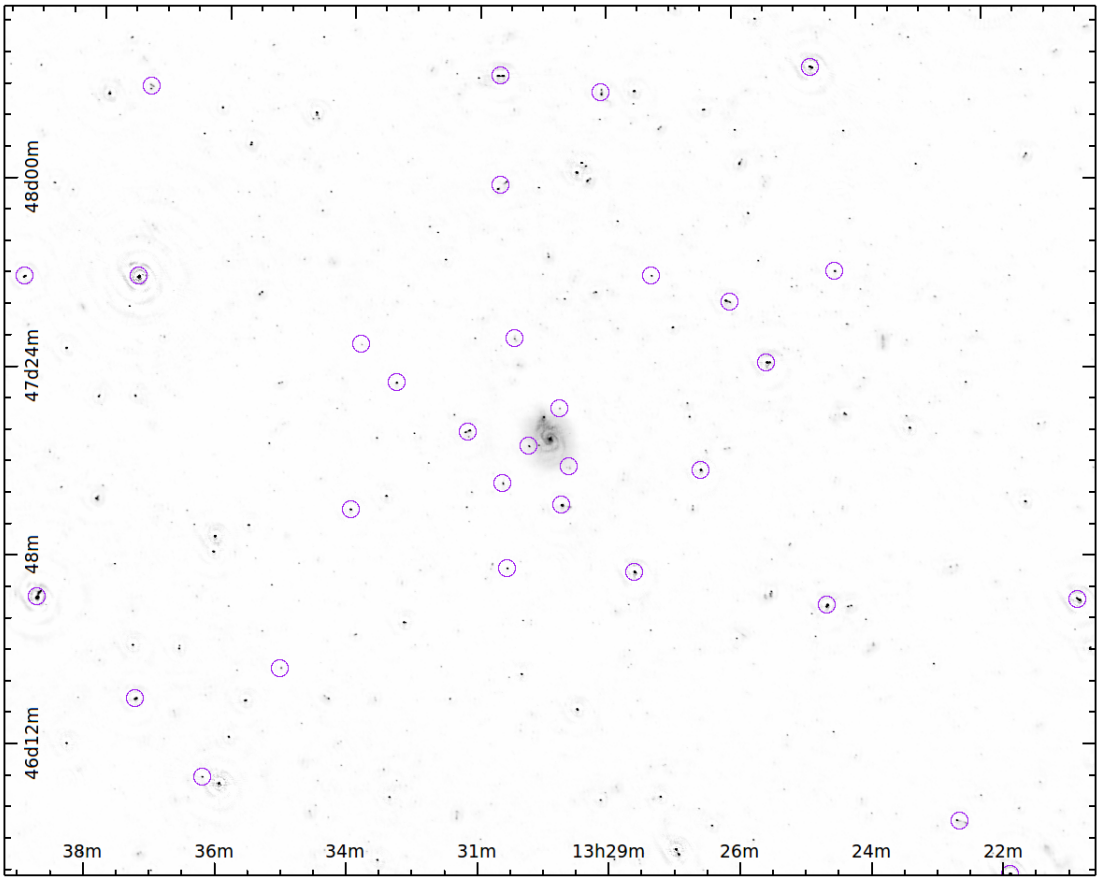


Figure 3.1: Part of the M51 field as observed by LOFAR in this study. The circles mark the locations of sources searched for polarization. Note that more examined sources lie outside the bounds of this figure.

3.1.2 LOFAR

LOFAR (LOw Frequency ARray) is often described as the first software telescope. It is described in more detail in van Haarlem et al. (2013).

While the main part is in the Netherlands, there are stations spread out over Europe. Each station has an array of dipoles, functioning as antennas. They point straight up, and the primary beam covers a large portion of the sky. The synthesized beam is created by interferometry, with the phase center determined by delays added electronically. This means that there are no moving parts. This also makes it possible to have several beams simultaneously, by splitting the signal. Data from the different stations is then combined using aperture synthesis.

LOFAR has two separate arrays, HBA (High Band Array) and LBA (Low Band Array). The bandwidth of the HBA, used for the observation, is $110 - 240\,\mathrm{MHz}$.

LOFAR is a pathfinder for the SKA (Square Kilometer Array), which will be built on the same principles. It will have a larger collecting area and wider frequency range.

3.1.3 The observation

The observation used in this thesis was done by Mulcahy et al. (2014), using the HBA array of LOFAR. I was given the calibrated data to reanalyze. Details of the observation and calibration can be found in Mulcahy et al. (2014), but some properties have been reproduced in Table 3.1. The frequency channels are divided into eight blocks at even intervals in the frequency range (see Figure 3.3).

The theoretical limits for what is possible to detect with RM synthesis (c.f. Appendix A.2) are:

- $\delta\phi \approx \frac{2\sqrt{3}}{\Delta\lambda^2} = \frac{2\sqrt{3}}{6.7-2.9} = 0.91\,\mathrm{rad\,m^{-2}}$
- max-scale $\approx \frac{\pi}{\lambda^2_{\mathrm{min}}} = \frac{\pi}{2.9} = 1.1\,\mathrm{rad\,m^{-2}}$

The fact that the maximum scale and resolution are nearly the same causes all polarization to show up as narrow peaks.

Table 3.1: Specifics of the observation by Mulcahy et al. (2014).

Observation date	22-23 April 2013
Observation duration	7 h, 55 m, 57.7 s
Total bandwidth	115.9 − 175.8 MHz
Number of channels	1944
Channel width	24.41 kHz
Time step	14.02 s

3.2 Source imaging

Here I will describe the process to image each source. The parameters of the different steps can be found in Table 3.2.

Before starting, I inspected the location of the source in an image spanning the entire field. If there were other sources nearby I modified the location of the field to image, to include them. In the cases when that was not possible, I made the image size larger than the default.

The first step was to shift the phase center of the data to the location of the source. I then averaged the data in time and frequency. Both of these steps were done in NDPPP[1].

I then made I, Q and U images for each channel. This was done using wsclean[2] (Offringa et al. 2014). I used the LOFAR primary beam correction included in the program.

To minimize artifacts, I calculated the PSF for the longest wavelength, and fit the restoring beam for all frequency images to that. The beam varied from image to image. The average major axis was 17.4″, the average minor axis 11.1″ and the average position angle 76.7°.

[1] New Default Pre-Processing Pipeline, part of the LOFAR software

[2] `https://sourceforge.net/projects/wsclean`

After that I used pyrmsynth[3] to do RM synthesis on the images. I later redid this with other parameters for the sources where polarization was found by the preliminary peak detection described in Section 3.4.

The large amount of data (870 GB) and the computational complexity of the imaging process make performance an issue. For large datasets the gridding of the data, which scales linearly with the number of visibilities, dominates (Offringa et al. 2014). Averaging in time by a factor of 170, as I do, reduces the number of visibilities by the same amount. This is the main reason I phase shift the data and create small images. When averaging in time, the image will be smeared out, but this effect is smaller toward the phase center. Making only a small image avoids this problem. On the hardware I used, imaging one source (with the RM synthesis parameters of the first pass and image size $256 \times 256''$) took $6 - 18$ h.

Step	**Parameter**	**Value**
Averaging	Channel averaging	2
	Time averaging	170
Imaging	Image size	256×256[a]
	Pixel size	$1''$
	Weighting	Briggs
	Robustness parameter	0
	Gain	0.1
	Max iterations	25000[a]
	Threshold	$300\,\mu$Jy/beam
	Major iteration gain	0.85
	Special arguments	-joinchannels
RM synthesis, first pass	Max Faraday depth	$100\,\mathrm{rad\,m^{-2}}$
	Sampling	$0.25\,\mathrm{rad\,m^{-2}}$
	Gain	0.1
	Max iterations	5000
	Threshold	$300\,\mu\mathrm{Jy\,beam^{-1}\,rmsf^{-1}}$
	Spectral index	None
RM synthesis, second pass	Max Faraday depth	$200\,\mathrm{rad\,m^{-2}}$
	Sampling	$0.1\,\mathrm{rad\,m^{-2}}$
	Gain	0.1
	Max iterations	25000
	Threshold	$300\,\mu\mathrm{Jy\,beam^{-1}\,rmsf^{-1}}$

[a] Some images were larger, to include sources that would otherwise disturb the image. In those cases, the maximum number of CLEAN iterations was scaled with the image area.

Table 3.2: Parameters of the data reduction.

[3] https://github.com/mrbell/pyrmsynth

3.3 Resulting images

Before searching for polarization in the sources, it is instructive to look at the noise properties of the resulting images, and some issues with the data.

3.3.1 Noise

The noise in the total intensity image varies between sources and locations within each imaged field, but is generally on the order of a few μJy/pixel (or a few 100 μJy/beam). The noise has large-scale components, and so depends on the size of the examined region, as seen in the left plot of Figure 3.2.

The noise in a slice of the source in Faraday depth also has large-scale components, but does not vary significantly with location (though it still varies between sources). It is generally on the order of $0.1 - 1\,\mu$Jy/pixel (around 100 μJy/beam). The middle plot of Figure 3.2 shows the dependence on region size.

Most importantly, and intuitively, the noise in the Faraday spectrum (created by summing F inside the region at each ϕ) increases with region size. This can be seen in the right plot of Figure 3.2. Because of this dependence I use the same size of the region when obtaining the noise as when I take the measurements, when I search for polarization in the sources.

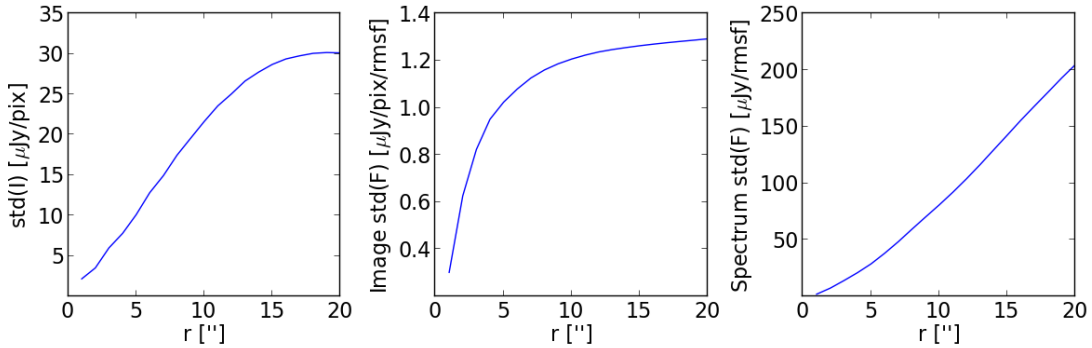


Figure 3.2: Standard deviation in a circular region depending on radius, taken near J132626+473741. (left) Example of total intensity noise, which varies significantly depending on location. (center) Noise in a slice of the Faraday cube (average across all Faraday depths). (right) Noise in the Faraday spectrum.

3.3.2 Issues with the data

One issue, also found in many other observations, is the **instrumental polarization** (sometimes called leakage). This is when the total intensity spills over into Q and U. It is not frequency dependent (except insofar as I is), so ideally it would show up as a peak at zero in the Faraday spectrum (Brentjens and de Bruyn 2005). It is however complicated by the ionospheric correction, which separates the peak into several different components in the region around zero (Sotomayor-Beltran et al. 2013). Noise in total intensity also contributes somewhat to the noise in Faraday space through this mechanism.

Figure 3.3 shows the total intensity of J133923+464008 plotted against frequency. This plot illustrates two issues:

- One large scale effect where the fourth and fifth frequency blocks appear displaced upward, and the sixth slightly downward. This is more pronounced in stronger sources.
- One small scale effect showing distinct structure, with oscillations and multi-channel peaks.

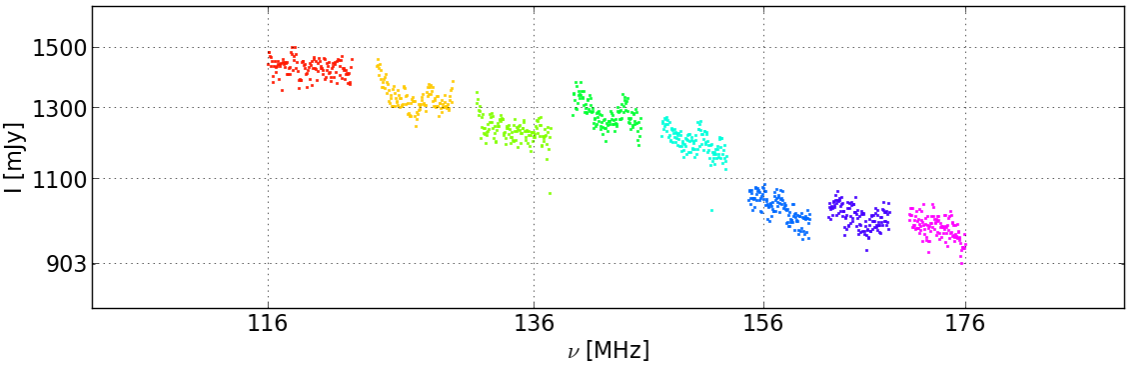


Figure 3.3: Total intensity of J133923+464008 plotted against frequency. The eight frequency blocks are shown in different colors. The axes are logarithmic.

The sources in this observation generally seem to be brighter than was found in the TGSS First Alternative Data Release [4] (Intema et al. 2016) by a factor of approximately 1.3, as can be seen in Figure 3.4.

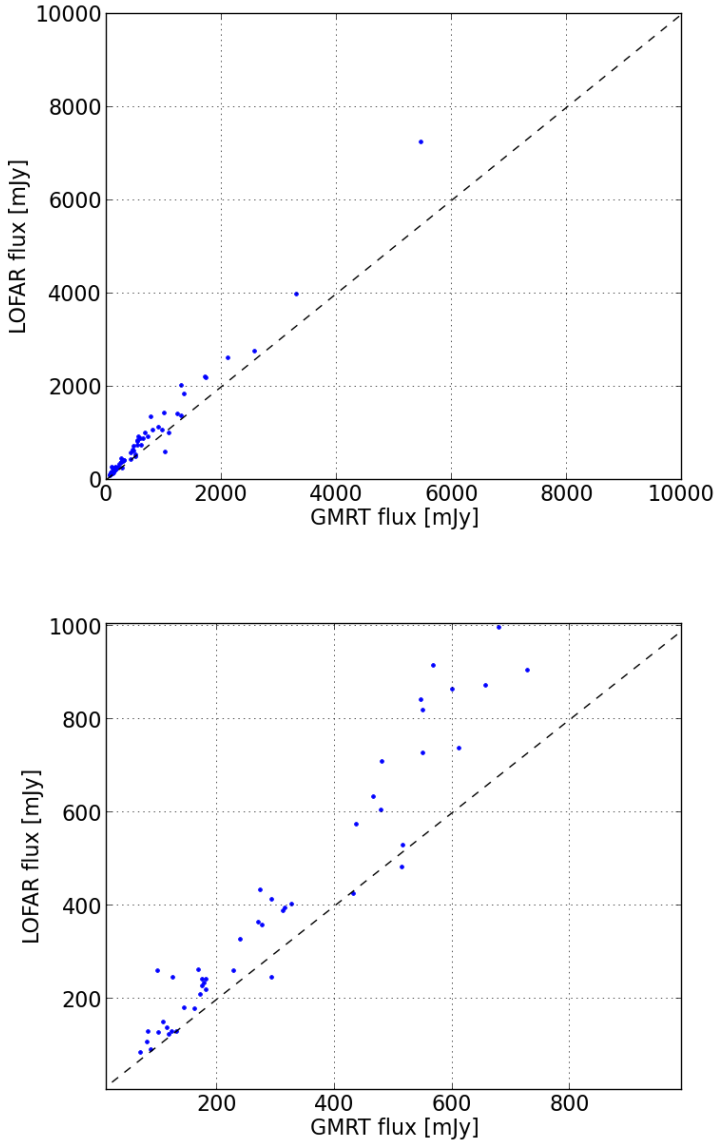


Figure 3.4: Flux density of the examined sources, compared to their intensity as given by Intema et al. (2016). I observe an increase by a factor of approximately 1.3. The upper plot shows the entire range, while the lower plot omits the brighter sources. The diagonal is shown as a dashed line.

[4]TIFR (Tata Institute of Fundamental Research) GMRT (Giant Metrewave Radio Telescope) Sky Survey: `http://vo.astron.nl/tgssadr/q/cone/form`

Figure 3.5 shows the intensities from Mulcahy et al. (2014) compared to TGSS, along with the corresponding intensities in this study. It is also curious that Mulcahy et al. (2014) and I derive different fluxes from the same data.

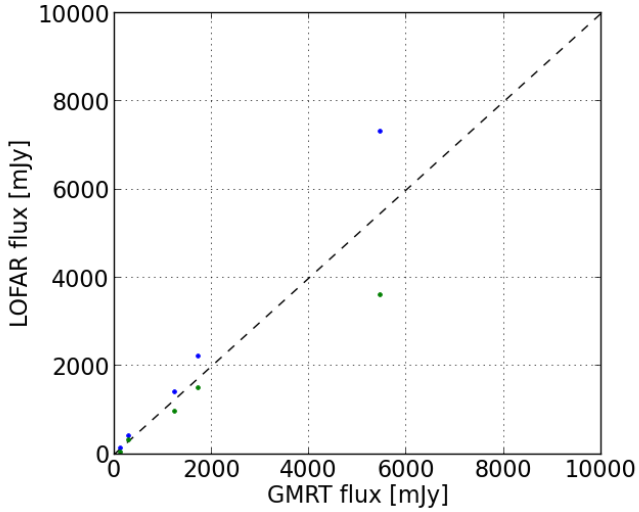


Figure 3.5: Intensity of the sources found by Mulcahy et al. (2014) compared to intensity from Intema et al. (2016). Green points use the intensity from Mulcahy et al. (2014), blue points use the intensity in this study.

I compared the flux densities of the strongest source, J134145+465716 (also named 4C+47.38), to those found in other studies. I did this by looking up the source in the NED database (Helou et al. 1991), and plotted the fluxes against frequency. The results, seen in Figure 3.6, indicates that the flux density from TGSS is the most accurate.

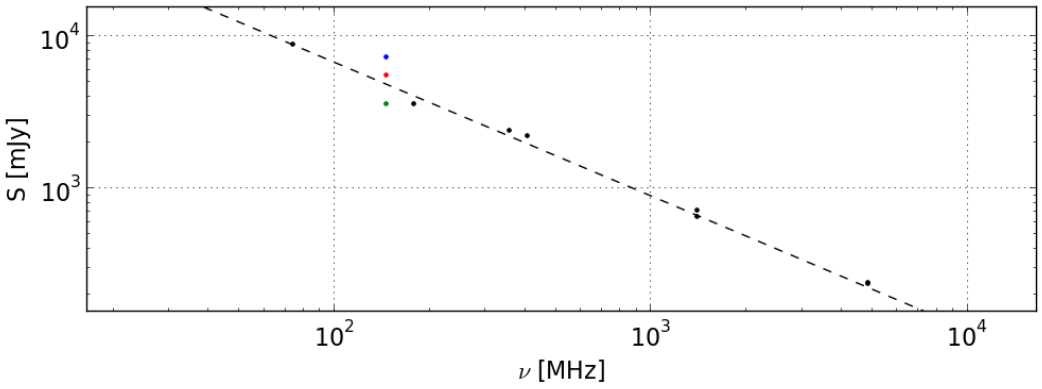


Figure 3.6: The flux densities of 4C+47.38 from the NED database (black), along with the fitted spectral power law (dashed). The colored points are the flux densities from this study (blue), TGSS (red) and Mulcahy et al. (2014) (green).

3.4 Finding polarized sources

I imaged sources that had already been observed to have polarization. They were:

- 7 sources from Mulcahy et al. (2014).
- 11 sources from Mao et al. (2015).
- 15 sources from Farnes, Green, et al. (2013).
- 43 sources from Taylor et al. (2009), restricted to a those within 2.85 ° of M51.

As there was some overlap, the total number of sources I imaged was 61.

I visually inspected the total intensity image of each field, and located any sources. To obtain the noise and a background, in both the Faraday spectrum and total intensity, I sampled several sourceless locations within the field, using the same region size as when I then measured the source. The Faraday background was calculated by taking the average in each Faraday spectrum (excluding $-10\,\mathrm{rad\,m^{-2}} < \phi < 10\,\mathrm{rad\,m^{-2}}$ to avoid instrumental polarization), and averaging those. To obtain the total intensity background I averaged the enclosed flux density of the regions. To determine the noise in the Faraday spectrum I computed the standard deviation in each Faraday spectrum (again excluding $-10\,\mathrm{rad\,m^{-2}} < \phi < 10\,\mathrm{rad\,m^{-2}}$), and took the quadratic mean of those. For the noise in total intensity, I took the standard deviation within each region and computed the quadratic mean of those. The threshold for detecting a peak was $5 \times \text{noise} + \text{background}$.

Most of the found peaks were due to instrumental polarization. To distinguish them from actual polarization I looked at their height relative to the main peak of the instrumental polarization. Figure 3.7 shows all the found peaks normalized to the main instrumental peak, except for those in two complex sources considered separately. I chose to discard the peaks between $-10\,\mathrm{rad\,m^{-2}}$ and $10\,\mathrm{rad\,m^{-2}}$, except for the outliers.

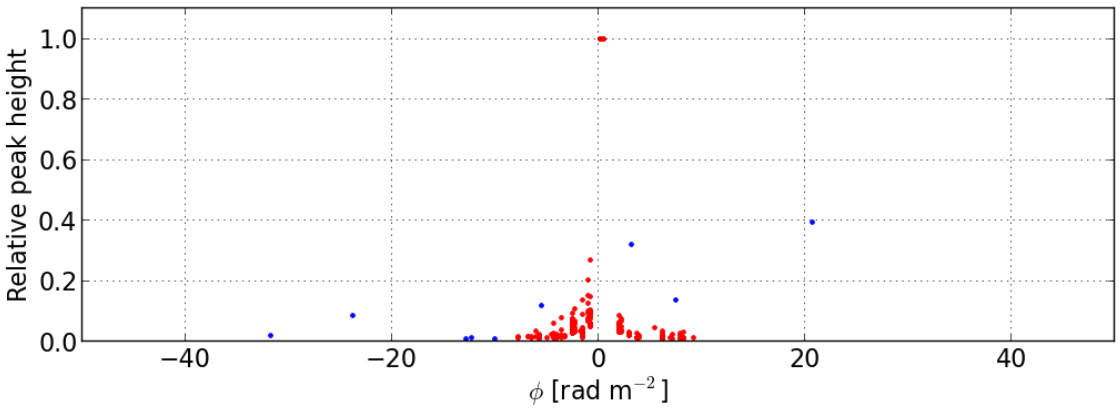


Figure 3.7: All peaks (except from two sources) above 5σ, normalized to the largest instrumental polarization peak of their respective sources. Those discarded as instrumental polarization are colored red.

For the peaks that were kept, I redid RM synthesis with higher sampling and range in Faraday space. The parameters can be found in Table 3.2, under *RM synthesis, second pass*.

The resulting Faraday cubes were smoothed in the image plane (using a Gaussian function with a kernel of radius 10″) to make these regions more visible. An example can be seen in Figure 3.8. These areas of polarization were sometimes difficult to distinguish from the artifacts from RM synthesis, so the choice of which to include was somewhat subjective. The Faraday spectra were taken from the unsmoothed Faraday cubes.

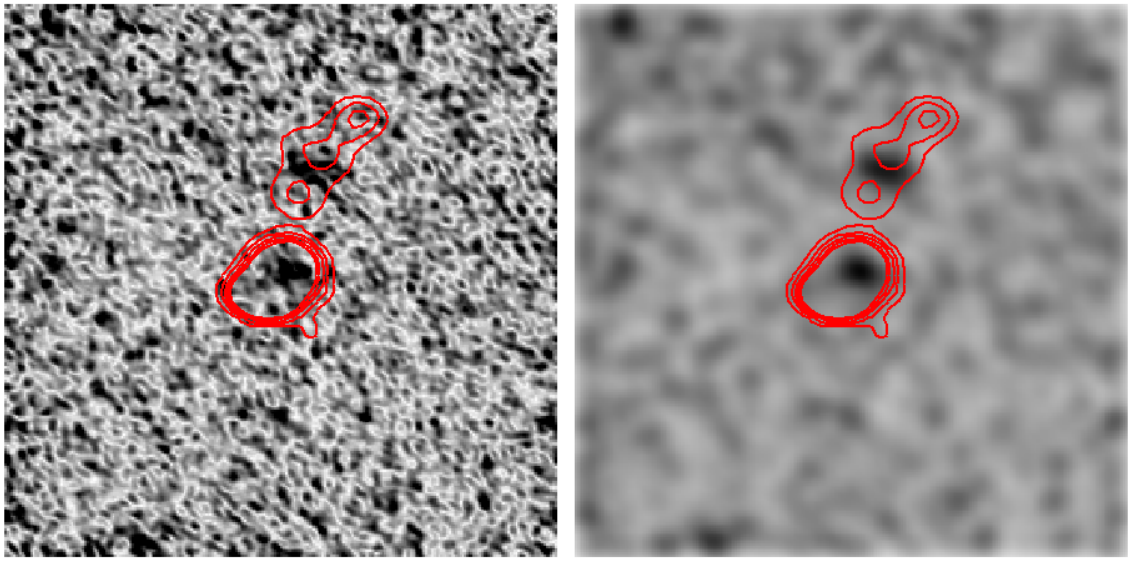


Figure 3.8: Polarization in J133923+464008 and J133920+464115 at $\phi = -23.4\,\mathrm{rad\,m^{-2}}$ before and after smoothing to make the sources clearer. Contours are total intensity.

3.5 Sources with detected polarization

The detected sources can be found in Table 3.3, with Table 3.4 comparing the observed Faraday depths to those from other observations. Following are more detailed examinations of each source. For the resolved sources I include images from both this observation and the higher-resolution (5″) FIRST[5] survey (Helfand et al. 2015).

[5]Faint Images of the Radio Sky at Twenty-Centimeters:`http://sundog.stsci.edu/`

Table 3.3: Polarized sources found near M51. Note that many sources are complex in Faraday space. The ϕ and |F| columns represent only the main polarization component, if one can be distinguished.

Name	**RA (J2000)**	**Dec (J2000)**	**I [mJy]**	$\boldsymbol{\phi}$ **[rad m^{-2}]**	**|F| [mJy/rmsf]**
J132444+463936	13h24m43.6s	+46d39m47.0s	460 ± 0.37		
J132626+473741	13h26m29.0s	+47d37m48.0s	130 ± 0.17	$+3.0 \pm 0.14$	2.4 ± 0.1
J132818+464622	13h28m18.0s	+46d46m23.7s	990 ± 0.45	-12.5 ± 0.12	0.7 ± 0.1
J132853+481747	13h28m53.9s	+48d18m26.0s	290 ± 5.9		
J133707+485801	13h37m07.7s	+48d58m03.5s	2200 ± 3.4	$+8.9 \pm 0.10$	1.3 ± 0.1
J133729+481808	13h37m30.3s	+48d17m45.9s	130 ± 0.11		
J133738+474148	13h37m38.5s	+47d41m48.8s	4000 ± 1.3		
J133920+464115	13h39m20.1s	+46d41m16.1s	770 ± 1.9	$+20.5 \pm 0.13$	12 ± 0.3
J133923+464008	13h39m23.9s	+46d40m02.3s	2700 ± 1.2		
J134145+465716	13h41m45.4s	+46d57m13.2s	7200 ± 4.1	$+23.2 \pm 0.01$	5.4 ± 0.2

Table 3.4: Faraday depth from this analysis compared with Mulcahy et al. (2014), Taylor et al. (2009), and Farnes, Green, et al. (2013). No sources from Mao et al. (2015) were found.

Source	**This study (150 MHz)**	**Mulcahy et al. (2014) (150 MHz)**	**Taylor et al. (2009) (1.4 GHz)**
J134145+465716	$+23.2 \pm 0.01$	$+23.5 \pm 0.1$	$+30.6 \pm 1.4$
J133920+464115	$+20.5 \pm 0.13$	$+20.5 \pm 0.1$	$+5.5 \pm 7.3$
J133707+485801	$+8.9 \pm 0.10$	$+9.2 \pm 0.1$	-8.9 ± 3.2
J132626+473741	$+3.0 \pm 0.14$	$+3.2 \pm 0.1$	
Name	**This study (150 MHz)**	**Farnes, Green, et al. (2013) (610 MHz)**	**Taylor et al. (2009) (1.4 GHz)**
J132818+464622	-12.5 ± 0.12	-6.683 ± 0.013	$+10.0 \pm 10.3$

3.5.1 J133923+464008 and J133920+464115

This source consists of two parts: one stronger part, J133923+464008, and one weaker, J133920+464115. Figure 3.9 shows an image of the source.

The weaker part is strongly polarized in two locations, both with Faraday depth around 20 rad m^{-2}. This result agrees with that of Mulcahy et al. (2014). The northwest of these has its maximum at 20.5 rad m^{-2} while the southeast is at 20.7 rad m^{-2}. There are also several regions with weaker polarization. The polarized regions can be seen in Figure 3.10.

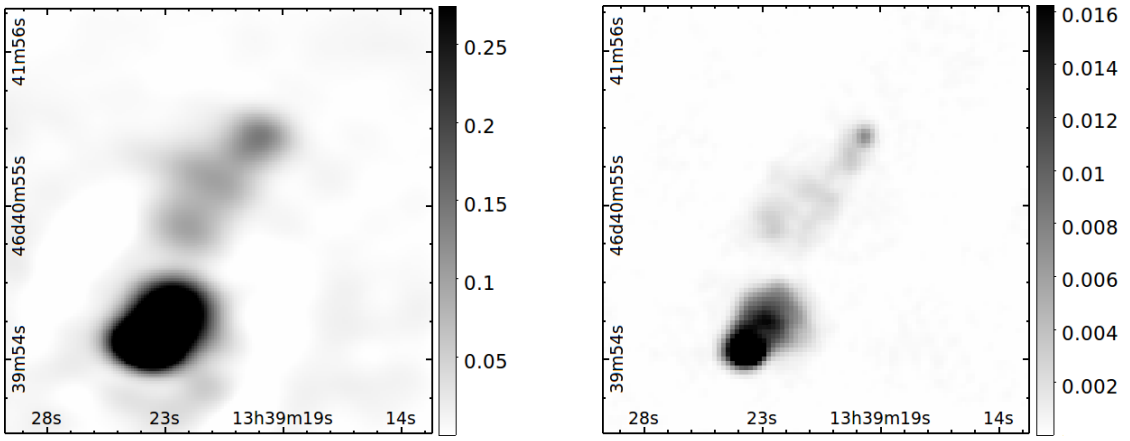


Figure 3.9: J133923+464008 and J133920+464115 as seen in this observation (left) and in FIRST (right).

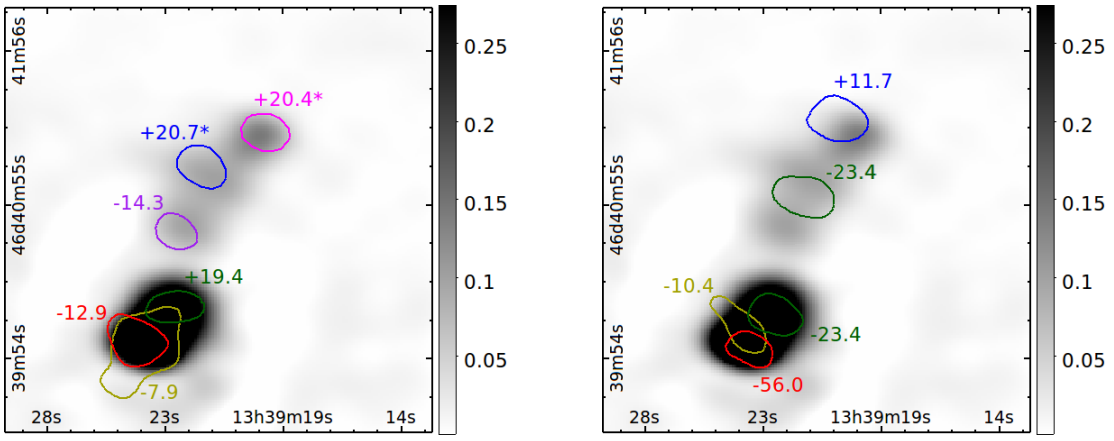


Figure 3.10: Polarized regions of J133923+464008 and J133920+464115. The images in Faraday space have been smoothed, see Figure 3.8. The region contours are at 0.4 mJy beam^{-1} rmsf^{-1} which is five times the rms noise, except for the strong polarization marked by an asterisk. There the contours are at 1.8 mJy beam^{-1} rmsf^{-1}.

3.5.2 J134145+465716 (4C+47.38)

J134145+465716 is one of the brightest in the M51 field. It consists of two unresolved sources, of which the southeast is highly polarized at $+23.2\,\mathrm{rad\,m^{-2}}$. The Faraday spectra of the two parts can be found in Figure 3.11.

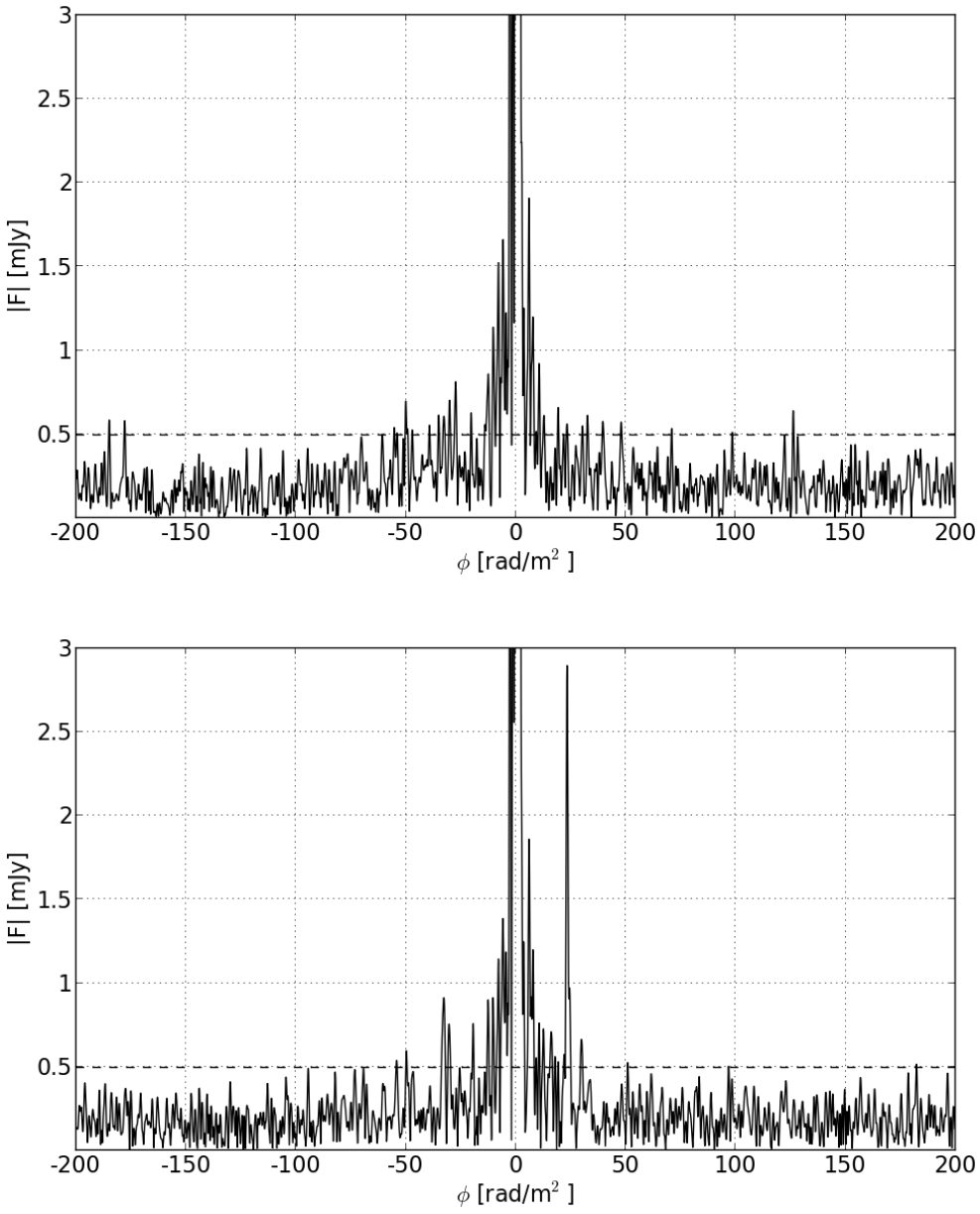


Figure 3.11: Faraday spectra of J134145+465716. Upper: northwest part. Lower: southeast part. The dashed line is the 5σ detection treshold.

3.5.3 J133707+485801

J133707+485801 shows polarization at $+8.9\,\mathrm{rad\,m^{-2}}$. Though the source is unresolved the polarization and the intensity have an offset, as seen in Figure 3.12. The Faraday spectrum at the polarized region can be seen in Figure 3.13.

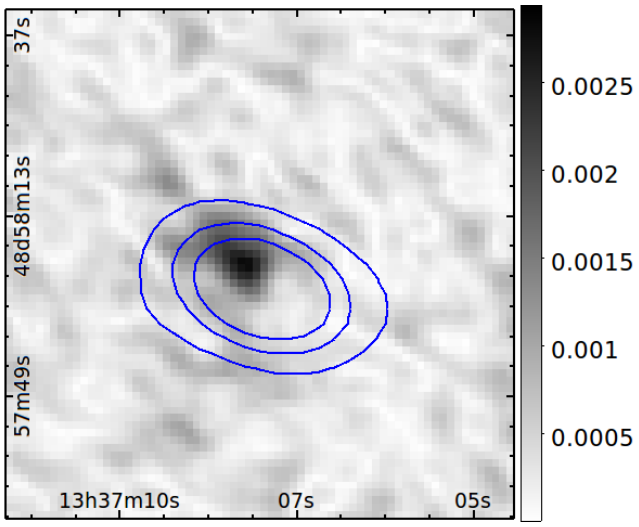


Figure 3.12: 133707+485801 in Faraday space at $\phi = +8.9\,\mathrm{radm^{-2}}$. Contours are the total intensity.

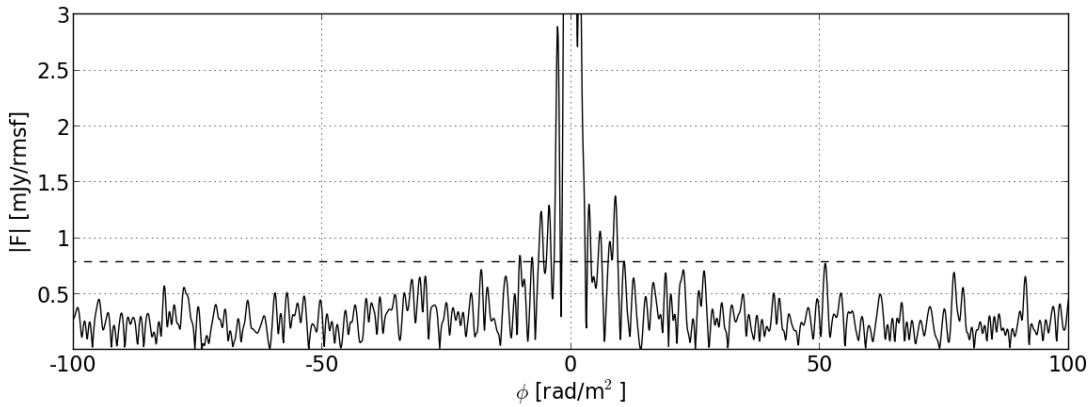


Figure 3.13: Faraday spectrum of 133707+485801. The dashed line is the 5σ detection treshold.

3.5.4 J133729+481808

Figure 3.14 shows J133729+481808. The source seems to consist of one double-lobed radio galaxy and one unrelated source. It shows several polarized regions, seen in Figure 3.15. While the polarization at $-2.6\,\mathrm{rad\,m^{-2}}$ (colored red) is within the range of instrumental polarization, it is notable that no polarization at that depth is observed in the stronger southern source. Also note the region at $+7.5\,\mathrm{rad\,m^{-2}}$, which is barely detected in total intensity but one of the most polarized regions of the source.

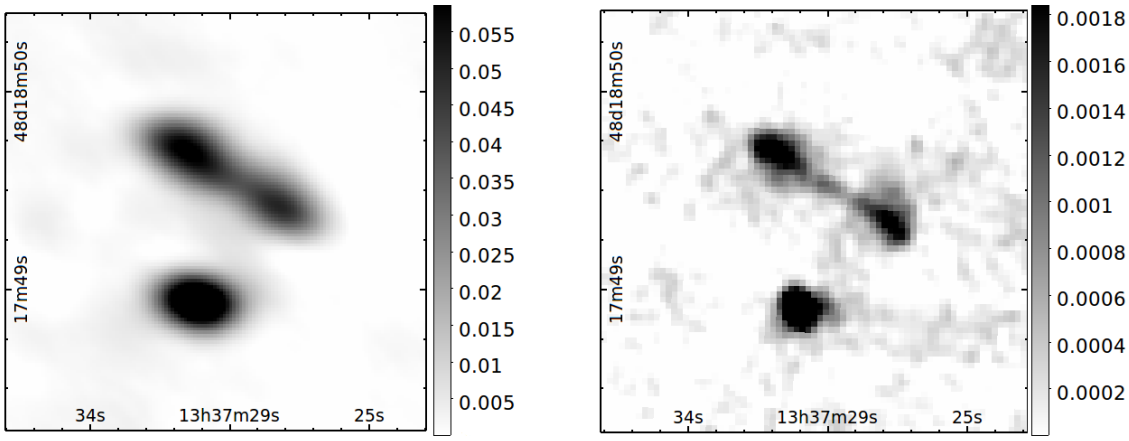


Figure 3.14: J133729+481808 as seen in this observation (left) and in FIRST (right).

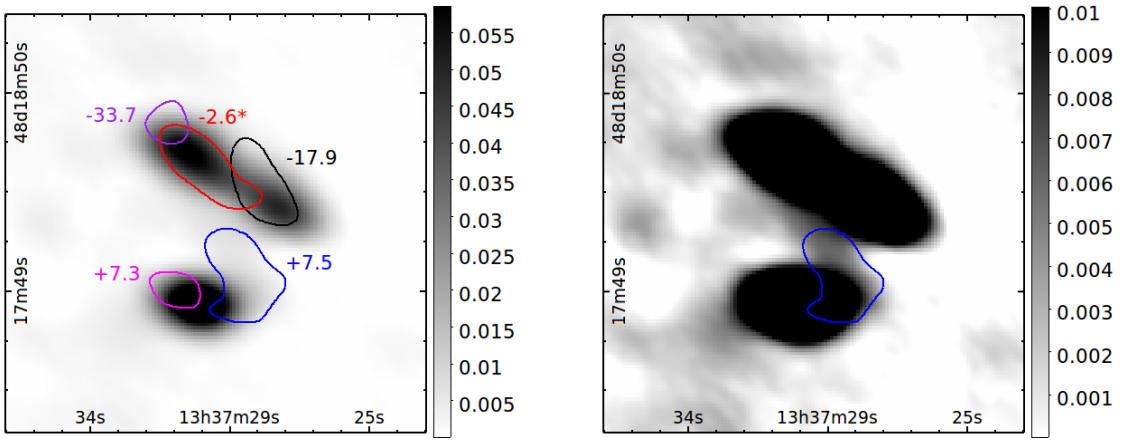


Figure 3.15: (left) Polarized regions of J133729+481808. The region marked with an asterisk is within the range of instrumental polarization, but I have chosen to include it for reasons explained in the text. (right) Changing the color scale reveals very weak emission in total intensity corresponding to one of the polarized regions. The images in Faraday space have been smoothed, see Figure 3.8. The region contours are at $0.4\,\mathrm{mJy\,beam^{-1}\,rmsf^{-1}}$ which is five times the rms noise.

3.5.5 J132818+464622

J132818+464622 is unresolved in this observation, but consists of two parts in FIRST (each unresolved). Figure 3.16 shows the Faraday spectrum. There is a peak at $-12.5\,\mathrm{rad\,m^{-2}}$. It is of the same strength as the weaker instrumental polarization peaks, but farther out than in any other case. Neither is J132818+464622 one of the brighter sources, so that peak is likely to be real. The other peaks are consistent with either instrumental polarization or artifacts.

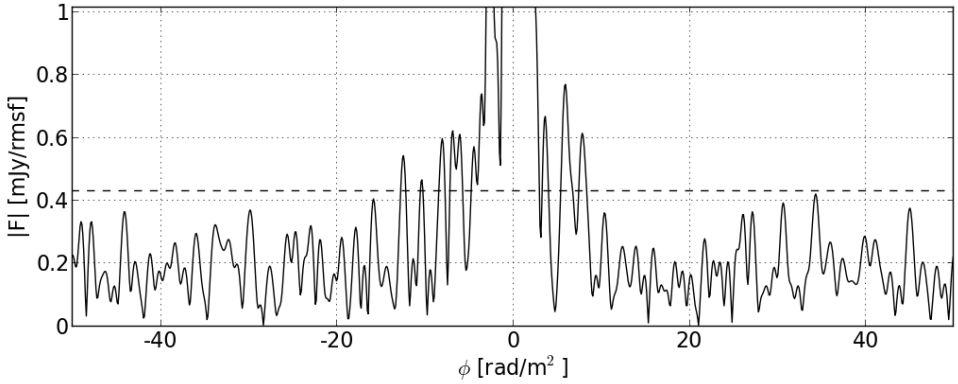


Figure 3.16: Faraday spectrum of J132818+464622. The dashed line is the 5σ detection treshold.

3.5.6 J133738+474148

J133738+474148, another of the brightest sources, is unresolved in the LOFAR observation, and shows little structure in FIRST. It is Faraday complex, showing polarization at many different Faraday depths. The spectrum can be seen in Figure 3.17.

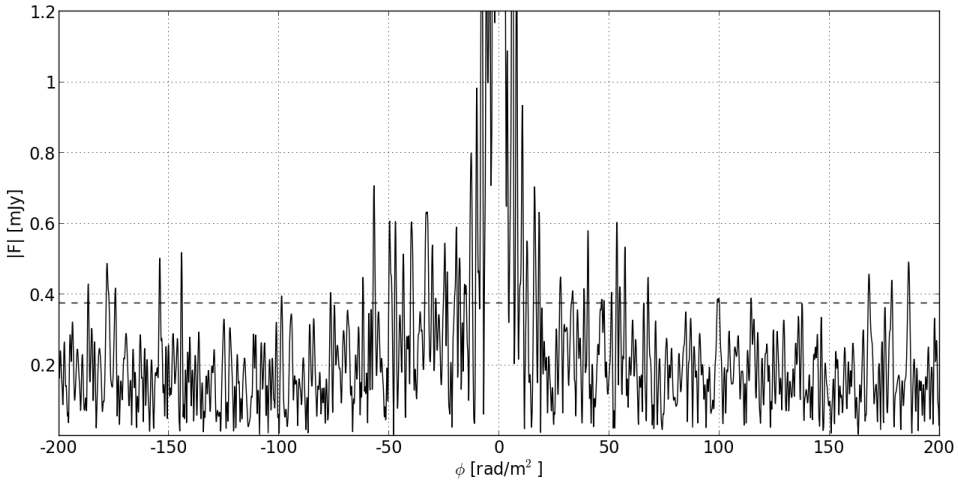


Figure 3.17: Faraday spectrum of J133738+474148. The dashed line is the 5σ detection treshold.

3.5.7 J132818+464622

Figure 3.18 shows J132818+464622. It appears to be a radio galaxy with two lobes. The polarization seems to be confined to a relatively small region, as seen in Figure 3.19.

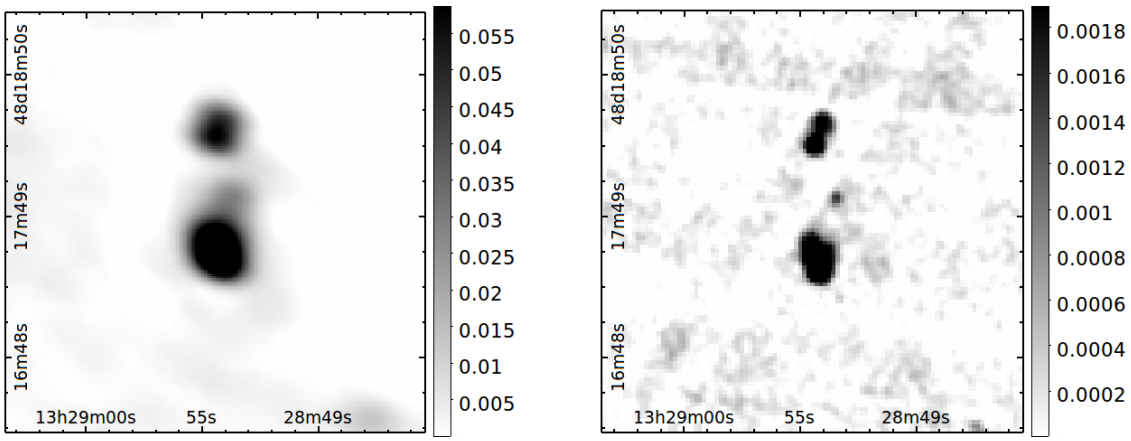


Figure 3.18: J132818+464622 as seen in this observation (left) and in FIRST (right).

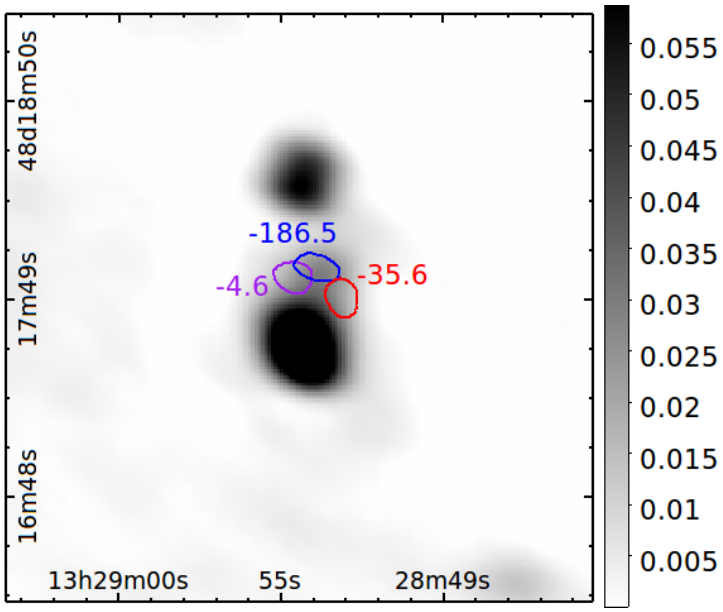


Figure 3.19: Polarized regions of J132818+464622. The images in Faraday space have been smoothed, see Figure 3.8. The region contours are at 0.3 mJy beam^{-1} rmsf^{-1} which is five times the rms noise.

3.5.8 J132444+463936

J132444+463936, shown in Figure 3.20, consists of two parts. They both show polarization at different RM, and another region offset from the northern part is polarized as well. The polarization can be seen in Figure 3.21.

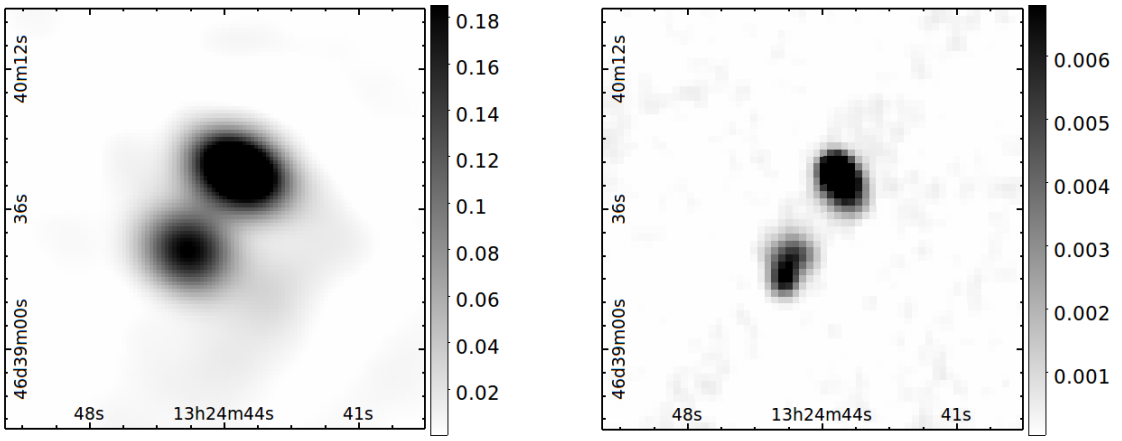


Figure 3.20: J132444+463936 as seen in this observation (left) and in FIRST (right).

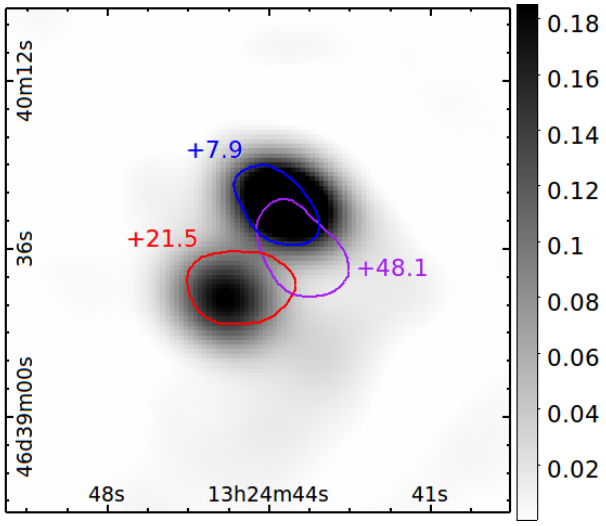


Figure 3.21: Polarized regions of J132444+463936. The images in Faraday space have been smoothed, see Figure 3.8. The region contours are at 0.3 mJy beam^{-1} rmsf^{-1} which is five times the rms noise.

3.5.9 J132626+473741

J132626+473741 is the fainter part of a resolved source, seen in Figure 3.22 along with its FIRST counterpart. J132626+473741 shows strong polarization at $+3.0\,\mathrm{rad\,m^{-2}}$ and weaker at $-23.9\,\mathrm{rad\,m^{-2}}$. The polarization at $+3.0\,\mathrm{rad\,m^{-2}}$ is imaged in Figure 3.23 and appears to extend even beyond the total intensity. The spectrum can be found in Figure 3.24.

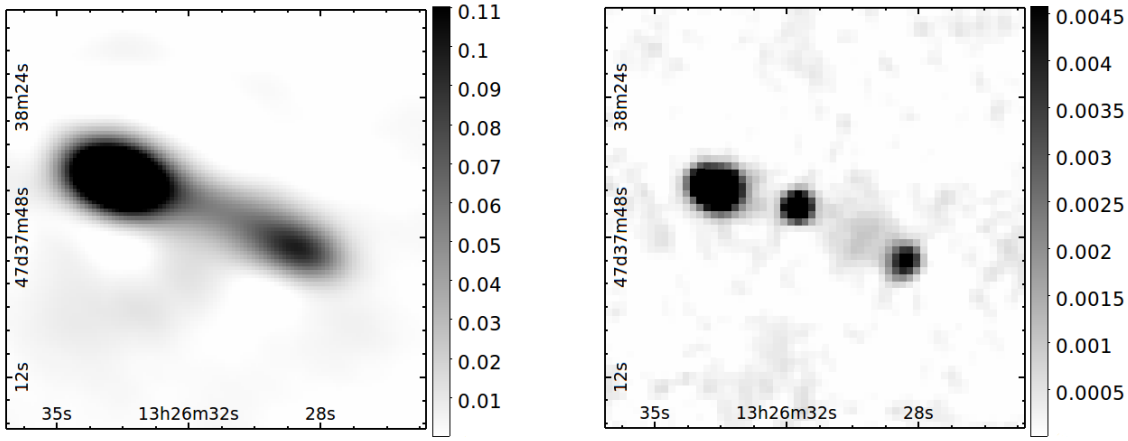


Figure 3.22: J132626+473741 as seen in this observation (left) and in FIRST (right).

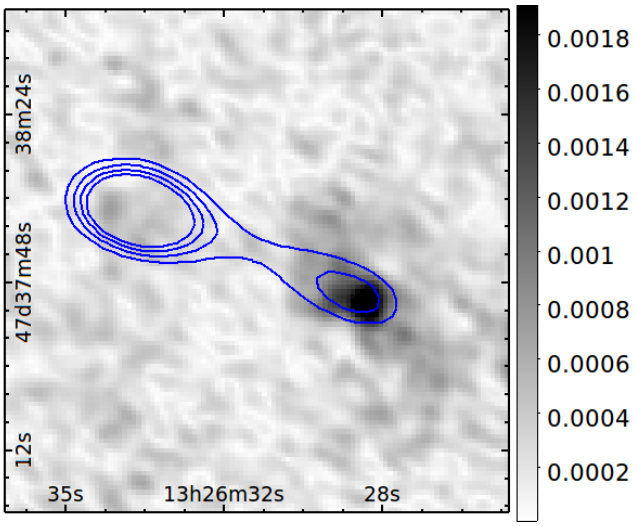


Figure 3.23: J132626+473741 in Faraday space at $\phi = +3.0\,\mathrm{rad\,m^{-2}}$. Contours are the total intensity.

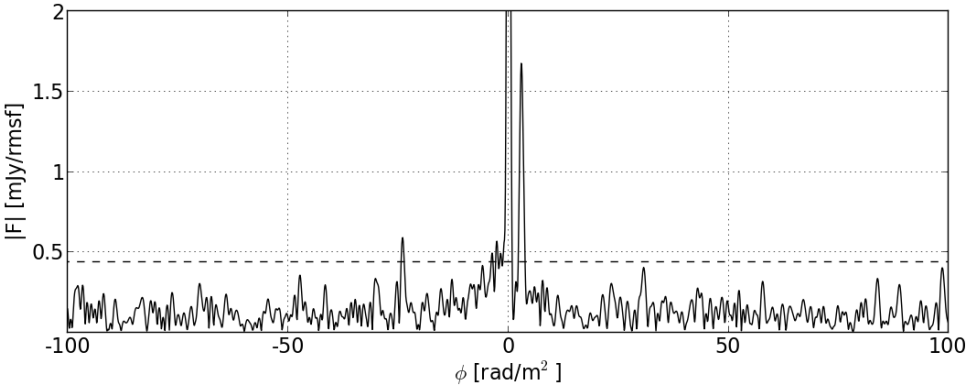


Figure 3.24: Faraday spectrum of J132626+473741. The dashed line is the 5σ detection treshold.

3.6 A closer look at J133920+464115

The source J133920+464115 has a region with a very distinct peak in $F(\phi)$, which makes it well-suited for illustrating a few concepts. Figure 3.25 shows this peak, and Figure 3.26 shows the strongly polarized region at several Faraday depths.

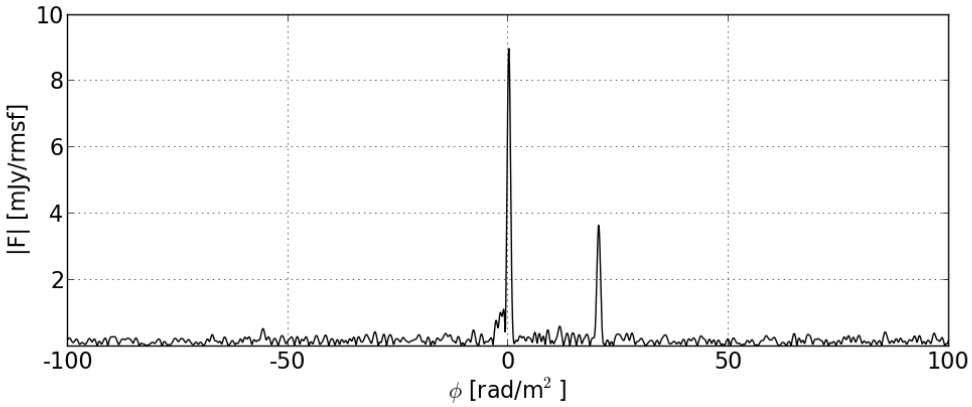


Figure 3.25: The strong polarization peak at $\phi = 20.45\,\mathrm{rad\,m^{-2}}$. Taken at the red region in Figure 3.28.

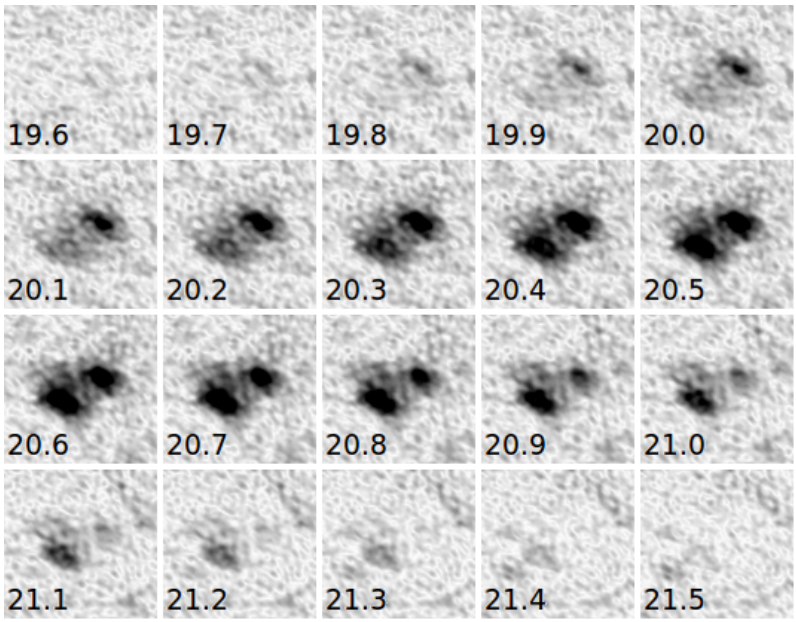


Figure 3.26: The two strong polarization peaks in J133920+464115, at Faraday depths between 19.6 rad m^{-2} and 21.5 rad m^{-2}. The northwest peak is located in the red region in Figure 3.28.

Due to the polarization being dominated by a single peak it is possible to see the oscillations in Q and U. (Though the instrumental polarization peak is higher, it shows up as an offset which is easy to correct for.) Figure 3.27 shows $q = Q/I$ and $u = U/I$ plotted against λ^2 in a range of the full band. A sine and cosine have been fitted to q and u, respectively. Their angular frequencies (with regard to λ^2) are, as expected, 20.45 rad m^{-2} and the offsets are approximately a quarter of a wavelength.

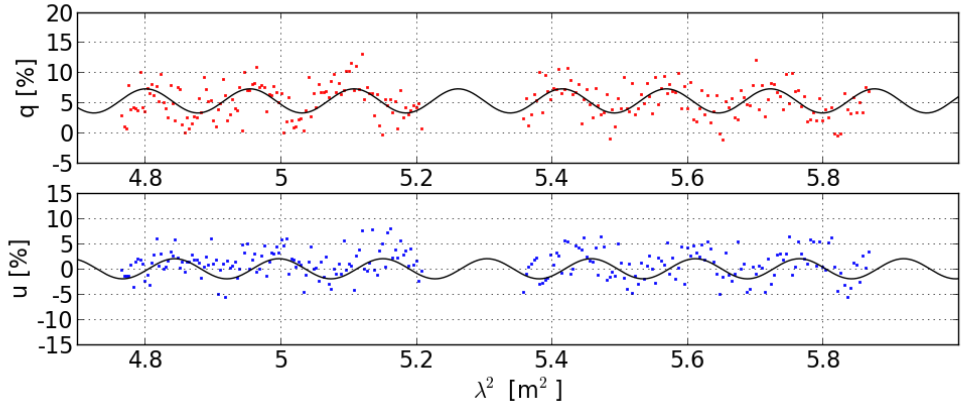


Figure 3.27: q and u plotted against λ^2, with fitted sine and cosine respectively. Taken at the red region in Figure 3.28.

Two kinds of depolarization can be seen in this source, at the two locations marked in Figure 3.28. Figure 3.29 shows the λ^2-dependence of the degree of polarization in the regions. The model for depolarization in a foreground screen from Burn (1966) (Equation 1.11) has been fitted to the data in the red region. There is clear agreement, but it should be noted that the initial polarization parameter p_i is at the imposed upper limit of 70 %, which is unrealistically high. If this limit is not imposed the best fit will be produced at even higher p_i. The fitted σ is 0.41. The Faraday depolarization model from Burn (1966) (Equation 1.12) has been fitted to the data in the blue region, showing agreement. The resulting $\Delta\phi$ is 0.23 rad m^{-2} and p_i is 17 %.

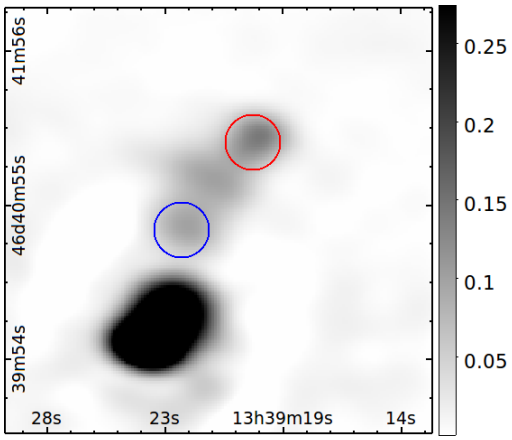


Figure 3.28: Two regions in J133920+464115 showing different kinds of depolarization.

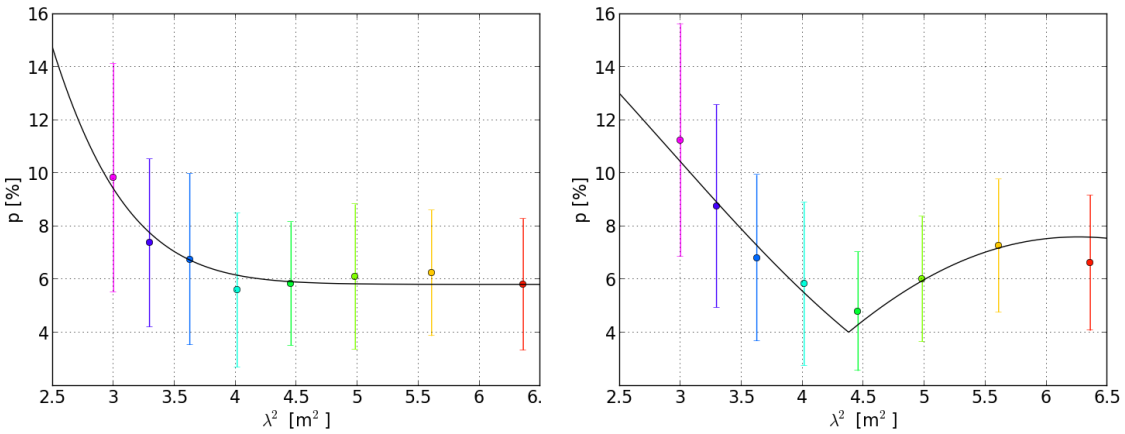


Figure 3.29: Dependence of p by λ^2 in the red (left) and blue (right) regions of Figure 3.28, consistent with depolarization from a foreground screen and Faraday depolarization respectively.

3.7 Discussion

When examining the radio sources in more detail it becomes apparent that they contain more polarization than was found by the preliminary peak detection. It is likely that examining the rest of them would reveal many others to also be polarized. Many of the sources seem to be complex in Faraday space. Because of the low maximum detectable scale in Faraday space it is not possible to tell whether the emission is localized in Faraday space, or whether the peaks represent the edges of extended emission.

Artifacts from RM synthesis are difficult to distinguish from actual polarization. Due to my subjective examination, it is not ruled out that some artifacts were interpreted as real emission and vice versa. Reducing these artifacts, as well as finding ways to distinguish them, is important to increase the chances that the detections are real and coming from the astrophysical sources. Finding a better way to distinguish between instrumental polarization and real polarized emission, or perhaps correct for the instrumental polarization, is also needed, particularly at small Faraday depths.

The small-scale variations in total intensity, I, (Figure 3.3) are on the same scale as some variations from polarized emission, and it is conceivable that similar structures appear in Q and U. However, as real polarization has more regularity and a quarter-wavelength offset between Q and U, this is unlikely to produce any false positives.

The large-scale variations in I could interfere with the degree of polarization, making the depolarization fits less certain. Another confounding factor is that the noise in Q and U is larger at low wavelengths in this dataset. As the degree of polarization involves squaring Q and U, the average P at low wavelengths will be increased. This can be confused with depolarization from a foreground screen, and will need to be corrected for. The apparent Faraday depolarization does not suffer as strongly from this effect.

The fact that the Faraday depths of Farnes, Green, et al. (2013) and Taylor et al. (2009) are different from what I found (Table 3.4) could point to errors in the data, but could also be the different wavelengths probing different Faraday depths. The different flux densities from Mulcahy et al. (2014) might have something to do with the imaging software or the primary beam correction. While I use wsclean with its beam correction, Mulcahy et al. (2014) used AWimager.

Looking back at Chapter 2, LOFAR (and the future SKA) would be a useful tool for that kind of research. SKA-Mid, in particular, will allow wideband observations at higher frequencies. In principle, LOFAR has a better resolution in Faraday space than SKA-Mid because it operates in a lower frequency range (larger wavelengths). However, depolarization effects are much stronger. This is both an advantage and a disadvantage. Stronger depolarization would increase the depolarization effects from foreground halos, but may also depolarize many sources completely. The higher frequencies of SKA-Mid would significantly increase the maximum observable scale in Faraday space.

More work is needed to investigate the polarization properties of LOFAR, understand the systematic uncertainties, and develop automated tools to extract polarization information from the very large datasets.

A

Observational theory

A.1 Radio Interferometry

Recall from Section 1.1 the resolution of a telescope:

$$\theta = 1.22\frac{\lambda}{D} \tag{A.1}$$

where D is the diameter of the telescope. This puts serious limitations on the achievable resolution with single-dish telescopes. Fortunately, there is a way to correlate the data from several telescopes to obtain up to the resolution of a hypothetical giant telescope, with a diameter as large as the largest distance between the real ones. This is called **aperture synthesis**. The information in this section is from *Tools of Radio Astronomy* (Wilson et al. 2009), which describes the subject in more depth.

Consider a pair of antennas i and j, pointing in the same direction. They will both receive signals from the same parts of the sky. Due to their different locations, however, there will be a small delay between the signals.

Correlating the signals gives us the **complex visibility** (or just visibility):

$$V_{ij} = \mathcal{F}(E_i(t))\mathcal{F}(\bar{E_j}(t)) \tag{A.2}$$

where $E(t)$ is the antenna signal and $\mathcal{F}$ denotes the Fourier transform. This can be shown to be related to the intensity distribution $I(x, y)$ of the observed source by:

$$V_{ij} = \mathcal{I}(u_{ij}, v_{ij}) \tag{A.3}$$

where $\mathcal{I}(u, v) = \mathcal{F}(I(x, y))$ is the two-dimensional Fourier transform of the source intensity distribution and u_{ij},v_{ij} are given by:

$$(u_{ij}, v_{ij}) = \frac{\mathbf{r}_i - \mathbf{r}_j}{\lambda} \tag{A.4}$$

where λ is the observational wavelength and $\mathbf{r}_{i,j}$ are the positions of the antennas projected onto a plane. This plane is perpendicular to the direction from which the signals reaching the antennas have the same phase. The point on the sky given by this direction is called the **phase center**, and defines the center of the field of view. By applying delays between the signals, or applying equivalent transformations to the visibilities, the phase center can be changed.

u and v are said to lie in the **uv-plane**, and the full collection of u_{ij} and v_{ij} is called the **uv-coverage**. These are the points at which $\mathcal{I}(u, v)$ is known. Better uv-coverage allows for a more accurate reconstruction of $I(x, y)$. Each **baseline** (distance between a pair of antennas) contributes two points to the uv-coverage at each moment. As the Earth is rotating the baselines change, increasing the uv-coverage immensely. See Figure A.1 for an illustration.

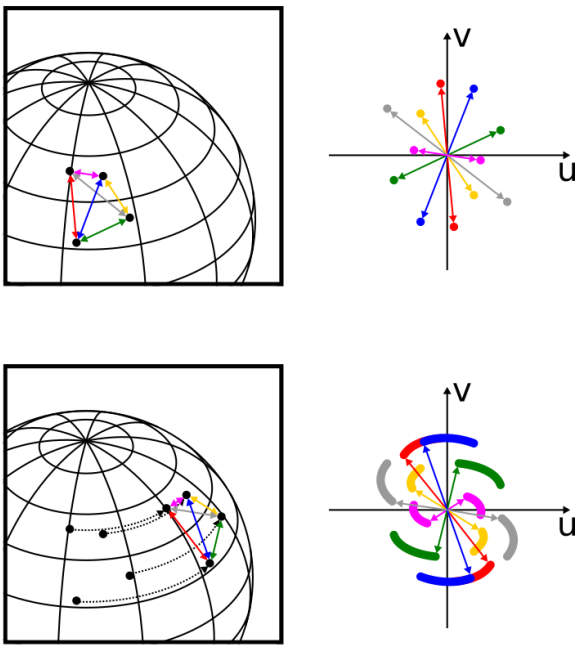


Figure A.1: Illustration of the uv-coverage. Each baseline contributes two points in the uv-plane (upper), and the baselines evolve with the rotation of the Earth (lower).

In the hypothetical case where the uv-coverage is perfect, the source intensity distribution can be reconstructed by:

$$I(x, y) = \mathcal{F}^{-1}(\mathcal{I}(u, v)) \tag{A.5}$$

With a realistic uv-coverage, it is not possible to reconstruct $I(x, y)$ unambiguously. In particular, the region around $u = v = 0$ corresponds to extended structures in the source intensity distribution. As a result, the largest observable scale is determined by the shortest baselines. Conversely, the resolution is determined by the longest baselines. Note that Equation A.4 means that the baselines, and as such the resolution and maximum scale, depend on the wavelength. A shorter wavelength leads to higher resolution and a smaller maximum scale, and vice versa.

The reconstructed intensity distribution with imperfect uv-coverage is given by:

$$\begin{aligned} I_{obs}(x,y) &= \mathcal{F}^{-1}(\sum_{i\neq j} g_{ij}\mathcal{I}(u_{ij},v_{ij})) = \\ &= \mathscr{F}^{-1}(\mathcal{I}(u,v) \times \sum_{i\neq j} g_{ij}\delta(u-u_{ij},v-v_{ij})) = \\ &= I(x,y) * PSF(x,y) \end{aligned} \tag{A.6}$$

with

$$PSF = \mathcal{F}^{-1}(\sum_{i\neq j} g_{ij}\delta(u-u_{ij},v-v_{ij})) \tag{A.7}$$

PSF is known as the **point spread function** (or the **dirty beam**) and depends on the uv-coverage as well as g_{ij}. The factor g_{ij} is arbitrary and is often chosen in one of a few different ways which depend on the **weighting scheme** (or just weighting) used. The three common weighting schemes are:

- **natural** weighting, where all baselines have equal weights.
- **uniform** weighting, where the (less numerous) longer baselines are given a higher weight, so that all scales will have the same weight in the reconstructed image. As a result, the resolution is higher but the signal-to-noise ratio is lower.
- **robust** (or **Briggs**) weighting, with characteristics in between uniform and natural weighting. A **robustness parameter** determines how close the weighting is to uniform (robustness parameter -2) or natural (robustness parameter 2).

PSFs for the three different weighting schemes are shown in Figure A.2.

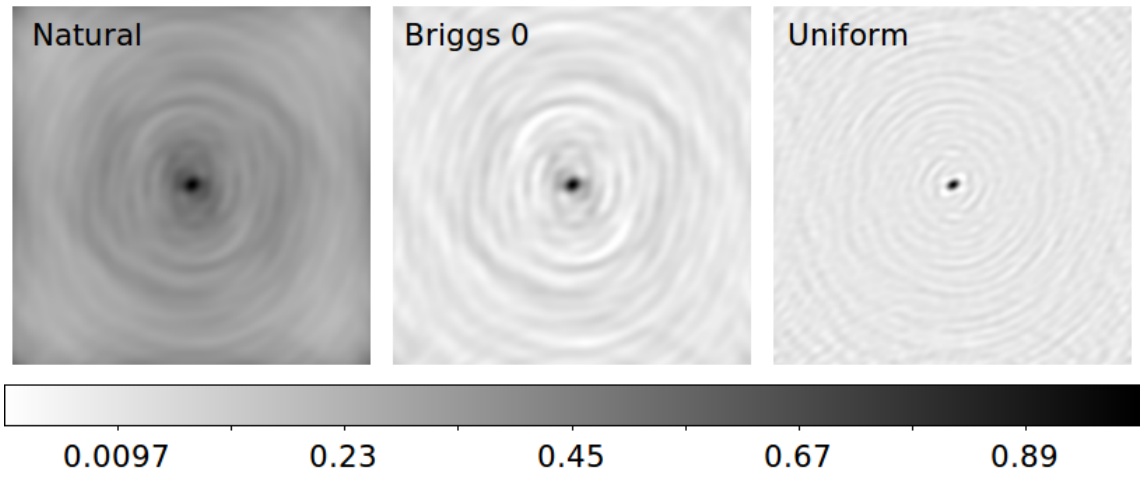


Figure A.2: PSFs for uniform, Briggs (robustness parameter 0) and natural weighting, for the LOFAR data used in this study. The uniform weighting gives a slightly higher angular resolution. The frequency is 116 MHz, the lowest in the data.

$I_{obs}(x, y)$ is known as the **dirty image**, and represents the assumption that the Fourier transform at all points not included in the uv-coverage is 0. This is, in general, not a valid assumption. The most common way to create a more realistic image that agrees with the observations is the **CLEAN algorithm** (Högbom 1974). The original CLEAN algorithm is as follows:

- Compute the dirty image and the PSF.
- Find the highest peak in the image
- Subtract the PSF from the image, centered on and scaled to that peak. The scaling factor is called the **gain**. Take note of the location and strength.
- Repeat the above two steps until either a specified **number of iterations** has been done, or the highest peak in the image is lower than a specified **cutoff**.
- Add the peaks back to the image, convolved with the **clean beam**. This is usually a Gaussian, fit to the central peak of the PSF.

The algorithm has later been refined and extended (Clark 1980; Schwab 1984).

A.2 RM synthesis

The seed of RM synthesis was planted by Burn (1966). The technique was later fully developed by Brentjens and de Bruyn (2005). The latter article is where all information in this section is from, unless stated otherwise.

Recall the Faraday dispersion function $F(\phi)$ from Section 1.2.1:

$$P(\lambda^2) = \int_{-\infty}^{\infty} F(\phi) e^{2i\phi\lambda^2} \, \mathrm{d}\phi \tag{A.8}$$

This equation is very similar to, and can be easily converted into, a Fourier transform. This makes it possible to reconstruct $F(\phi)$ from $P(\lambda^2)$, but due to the imperfect λ^2-coverage the reconstruction will not be unambiguous.

The imperfect coverage results in a "dirty" Faraday dispersion function, analogous to the dirty image in aperture synthesis. This reconstructed $F(\phi)$ is the convolution of the true Faraday dispersion function and a function called the **rotation measure spread function** (RMSF), analogous to the PSF. An example of an RMSF can be found in Figure A.3. The dirty Faraday dispersion function can be cleaned in a way similar to the CLEAN algorithm used in aperture synthesis.

The frequency coverage of the observation determines what is possible to see with RM synthesis:

- The resolution in Faraday space is given by $\delta\phi \approx \frac{2\sqrt{3}}{\Delta\lambda^2}$, where $\Delta\lambda^2$ is the difference between the maximum and minimum values of λ^2 in the observation.
- The maximum scale of observable structures (i.e. where the sensitivity drops to 50 %) in Faraday space is approximately $\frac{\pi}{\lambda^2_{\mathrm{min}}}$, where λ_{min} is the shortest wavelength.

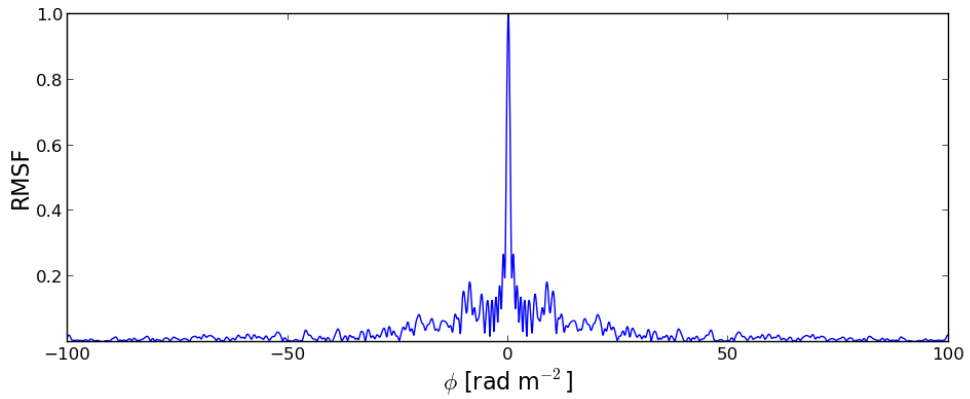


Figure A.3: The rotation measure spread function for the LOFAR data used in this study.

B

Code

B.1 Code for the analysis in Chapter 2

The code for searching for patterns in the polarization of background sources near foreground galaxies was written in MATLAB and based on code written by my supervisor Cathy Horellou. I use the 'coco' function from the Astronomy & Astrophysics package for Matlab (Ofek 2014) for coordinate conversion.

```
%#ok<*DEFNU> It's fine if functions are unused,
%            they're probably just commented out.
function exjobb
    tic
    %————————————————————Commonly used options————————————————————%
    GALAXIES = 'RC3'; % SINGS, RC3, dummy
    RANDOMIZATIONS = 0;
    SOURCE_FINDING = {...
                      'xrc', 20 ... % in rnorm
                     ,'excrc', 1 ... % in rnorm
                     ,'ellipticalRnorm', 0 ...
                       };
    DUPLICATE_DOUBLE = 1; % Else delete
    FILTER_PASS_1 = {...
                      'latitude', [20 90] ...
                     ,'ppercent', [0 20] ...
                     ...,'RM', [-100 100] ...
                     ...,'absRRM', [0 4] ...
                     ,'I', [0 1000] ...
                     ,'onlyWithZ', 0 ...
                     ,'onlyWithoutZ', 0 ...
                     ...,'z', [-inf 999999] ...
                     ,'onlySDSS', 0 ...
                     ,'onlyNED', 0 ...
                     ,'SDSSGal', 0 ...
                     ,'NEDGal', 0 ...
                     ,'SDSSQSO', 0 ...
                     ,'NEDQSO', 0 ...
                     ,'FIRSTnb', [0 inf] ...
                     ,'onlyWithExtent', 0 ...
```

```
                    ,'onlyWithoutExtent', 0 ...
                    ...,'extent', [0 50] ... % in arcsec
                       };
FILTER_PASS_2 = {...
                     'theta', [0 360] ... % in deg
                    ,'relTheta', [0 90] ... % in deg
                    ...,'L', [0 5000] ... % in pc
                       };
GALAXY_LOCATIONS = {...
                     'latitude', [20 90] ... % deg
                     ,'dec', [-40 90] ... % deg
                     };
GALAXY_FILTER = {...
                    'onlyWithT', 0 ...
                    ,'excludeT', 0 ...
                    ...,'T', [0 9] ... % Hubble type
                    ...,'distance', [-inf 20] ... % Mpc
                    ...,'z', [-inf 9999] ...
                    ...,'minMajFrac', [0 0.5] ... % minor axis/major axis
                    ,'maja', [0 180] ... % Major axis in arcmin.
                       };
% Upper limit on major axis is to filter out M31, LMC and SMC.

%----------------------------Do the calculations----------------------------%
skySources = prepareSources();
skySources = chooseExtent(skySources);
skySources = filterSourcesPass1(skySources, FILTER_PASS_1{:});
galaxies = readGalaxies(GALAXIES);
galaxies = filterGalaxies(galaxies, GALAXY_LOCATIONS{:}, ...
                                    GALAXY_FILTER{:});
%galaxies = randomizeGalaxies(galaxies, GALAXY_LOCATIONS{:});
trials = prepareTrials(galaxies, GALAXY_LOCATIONS, ...
                               'amountOfRandomizations', RANDOMIZATIONS);
for trialInd = 1:length(trials)
    if length(trials) > 1
        if trialInd == 1
            display('Doing real galaxies')
        else
            display(['Doing random trial ' num2str(trialInd-1)])
        end
    end
    trial = trials(trialInd);
    galaxies = trial.galaxySet;
    [sources_tmp, galaxies] = findsources(skySources, galaxies,...
                                  SOURCE_FINDING{:});
```

```
        galaxySources = getGalaxySources(sources_tmp,...
                        'duplicateDoubleSources', DUPLICATE_DOUBLE);
        galaxySources = filterSourcesPass2(galaxySources,...
                                           FILTER_PASS_2{:});
        galaxies = assignGalaxySourceNb(galaxies, galaxySources);
        trial = saveTrial(trial, galaxySources);
        trials(trialInd) = trial;
        if trialInd == 1
            savedSources = galaxySources;
            savedGalaxies = galaxies;
        end
    end
    galaxySources = savedSources;
    galaxies = savedGalaxies;

    %--------------------------Display the results--------------------------%
    displayResults(skySources, galaxySources, ...
                   galaxies, trials)
end

function displayResults(skySources, galaxySources, galaxies, trials)
    FIG_OFFSET = 0;
    RNORM_LIMIT = 10;

    displayOutliers(skySources, galaxies,'ppercentLimit',100,...
                                         'sourceNbLimit',500);
    if length(trials) > 1
        trials = analyzeTrials(trials,'inside',[1 RNORM_LIMIT],...
                                      'outside',[RNORM_LIMIT 20]);
        figure(20 + FIG_OFFSET); clf;
        subplot(231)
        trialModeResults(trials, 'ppercent', 'difference');
        subplot(232)
        trialModeResults(trials, 'poli', 'difference');
        subplot(233)
        trialModeResults(trials, 'absrrm', 'difference');
        subplot(234)
        trialModeResults(trials, 'totali', 'difference');
        subplot(235)
        trialModeResults(trials, 'number', 'difference');
        subplot(236)
        trialModeResults(trials, 'density', 'difference');
    end

    display('---Whole sky---')
    displayCatalogStats(skySources);
```

```
    % This will be slightly inaccurate if sources behind several galaxies
    % are duplicated.
    display('---Behind galaxies---')
    displayCatalogStats(galaxySources)

%     figure(1 + FIG_OFFSET); clf;
%     plotSkyMap(skySources, galaxySources, galaxies, ...
%                         'extraVisibleBehind',1, 'galaxyEllipseFactor',10);
%     figure(2 + FIG_OFFSET)
%     plotRRMbyLat(skySources);

    plotByFractionalRadius(3 + FIG_OFFSET, galaxySources, skySources ...
                      ,'numberOfBins',8 ...
                      ,'binByArea',0 ...
                      ,'rnormLine',RNORM_LIMIT ...
                           );

%     figure(4 + FIG_OFFSET); clf;
%     galaxyHistograms(galaxies,'rc3mode',1);
%     figure(5 + FIG_OFFSET); clf;
%     sourceHistograms(skySources);
%     suptitle('Whole sky')
%     figure(6 + FIG_OFFSET); clf;
%     sourceHistograms(galaxySources);
%     suptitle('Behind galaxies')

%     figure(7 + FIG_OFFSET); clf;
%     sourcePlot(galaxySources,'rrm','theta')
%
%     figure(8 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources,'z','ppercent')
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources,'z','ppercent')
%     title('Behind galaxies')
%
%     figure(9 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources,'extent','ppercent')
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources,'extent','ppercent')
%     title('Behind galaxies')
%
```

```
%     figure(10 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources,'extent','poli')
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources,'extent','poli')
%     title('Behind galaxies')
%
%     figure(11 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources, 'absrrm', 'ppercent','numberOfBins',10,...
%                           'leftEdge',2,'rightEdge',6);
%
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources, 'absrrm', 'ppercent','numberOfBins',10,...
%                               'leftEdge',2,'rightEdge',6);
%     title('Behind galaxies')
%
%     figure(12 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources, 'rm', 'ppercent');
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources, 'rm', 'ppercent');
%     title('Behind galaxies')
%
%     figure(13 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources, 'rrm', 'ppercent','numberOfBins',9,...
%                           'leftEdge',-5.5,'rightEdge',5.5);
%     hold on
%     plot([0 0],ylim,'k--')
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources, 'rrm', 'ppercent','numberOfBins',9,...
%                               'leftEdge',-5.5,'rightEdge',5.5);
%     hold on
%     plot([0 0],ylim,'k--')
%     title('Behind galaxies')
%
%     figure(14 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources, 'rrm', 'poli');
%     title('Whole sky')
```

```
%     subplot(122)
%     sourcePlot(galaxySources, 'rrm', 'poli');
%     title('Behind galaxies')
%
%     figure(15 + FIG_OFFSET); clf;
%     subplot(121)
%     sourcePlot(skySources, 'absrrm', 'poli');
%     title('Whole sky')
%     subplot(122)
%     sourcePlot(galaxySources, 'absrrm', 'poli');
%     title('Behind galaxies')

    % Because realizing you missed a plot is not very fun
    if toc > 5*60
       keyboard
    end
end

function sources = prepareSources(varargin)
% Loads source info from files.
% Options:
%     replaceFile
%     redoRRM
%     redoFIRSTmatching
%     redoNVSSmatching
    defaults = struct('replaceFile', 0, 'redoRRM',0, ...
                      'redoFIRSTmatching',0, 'redoNVSSmatching',0);
    option = makeOptionStruct(defaults, varargin);

    if ~exist('sources.mat', 'file') || option.replaceFile
        sources = readTaylorData();
        sources = addHammondData(sources);
        sources = removeForeground(sources);
        sources = matchToOther(sources, 'other', 'FIRST',...
                               'matchDist', 0.5*45/3600);
        sources = matchToOther(sources, 'other', 'NVSS');
        save('sources.mat', 'sources');
    else
        S = load('sources.mat');
        sources = S.sources;
        if option.redoRRM
            sources = removeForeground(sources);
        end
```

```
        if option.redoFIRSTmatching
            sources = matchToOther(sources, 'other', 'FIRST',...
                                'matchDist', 0.5*45/3600);
        end
        if option.redoNVSSmatching
            sources = matchToOther(sources, 'other', 'NVSS');
        end
        if option.redoFIRSTmatching || option.redoNVSSmatching ||...
                                       option.redoRRM
            save('sources.mat','sources');
        end
    end
end

function sources = readTaylorData(varargin)
% Reads sources from Taylor (2009).
% Options:
%     filename
    defaults = struct('filename', 'taylor.fits');
    option = makeOptionStruct(defaults, varargin);

    fitsfile = fitsread(option.filename, 'BinTable');
    ra = num2cell(fitsfile{3});
    dec = num2cell(fitsfile{5});
    totali = num2cell(fitsfile{7});
    poli = num2cell(fitsfile{9});
    ppercent = num2cell(fitsfile{11});
    rm = num2cell(fitsfile{13});
    sources = struct('ra',ra, 'dec',dec, 'totali',totali, 'poli',poli,...
        'ppercent',ppercent, 'rm',rm, 'rrm',[], 'z',[], 'FIRSTname',[],...
        'FIRSTnb',[], 'FIRSTextent',[], 'NVSSname',[], 'NVSSextent',[],...
        'NVSSextentSmallerThan',[], 'extent',[], 'SDSStype',[],...
        'NEDname',[], 'NEDtype',[], 'SIMBADname',[], 'galaxyNb',[],...
        'galaxy',[], 'theta',[], 'relTheta',[], 'r',[], 'rnorm',[]);
end

function sources = addHammondData(sources, varargin)
% Adds data from Hammond (2012) to some sources.
% Options:
%     filename
    defaults = struct('filename', 'hammondRM.fits');
    option = makeOptionStruct(defaults, varargin);

    fitsfile = fitsread(option.filename, 'BinTable');
    ra = fitsfile{4};
    dec = fitsfile{5};
```

```
    % Most of these are just to check that everything is ok.
    totali = fitsfile{8};
    poli = fitsfile{10};
    ppercent = fitsfile{12};
    z = fitsfile{76};
    rm = fitsfile{14};% We do not use the included RRM
    oldj = 1;
    ra2 = [sources.ra];
    dec2 = [sources.dec];
    for i = 1:length(ra)
        for j = oldj:length(ra2)
            % Find the source in Taylor
            if abs(ra(i)-ra2(j)) < 0.00001 && abs(dec(i)-dec2(j)) < 0.00001
                sources(j).z = z(i);
                oldj = j;
                if fitsfile{31}(i) == 1
                    sources(j).SDSStype = fitsfile{35}(i);
                end
                if fitsfile{23}(i) == 1
                    sources(j).SIMBADname = fitsfile{24}(i);
                end
                if fitsfile{16}(i) == 1
                    sources(j).NEDname = fitsfile{17}(i);
                    sources(j).NEDtype = fitsfile{18}(i);
                end
                if totali(i) ~= sources(j).totali ||...
                        poli(i) ~= sources(j).poli ||...
                        ppercent(i) ~= sources(j).ppercent ||...
                        rm(i) ~= sources(j).rm
                    display(['Some data does not match'...
                             ' between Hammond and Taylor.'])
                end
                break
            end
        end
    end

end

function sources = removeForeground(sources, varargin)
% Read RRM from the catalog provided with Oppermann (2015)
% Options:
%     filename
    defaults = struct('filename', 'oppermann_sources.fits');
    option = makeOptionStruct(defaults, varargin);
```

```
display('Taking RRM from Oppermann et al.')
[sources.rrm] = deal([]);
[sources.absrrm] = deal([]);
infofits = fitsread(option.filename,'BinTable',1);
rrmfits1 = fitsread(option.filename,'BinTable',2);
rrmfits2 = fitsread(option.filename,'BinTable',3);
avgrrmfits = fitsread(option.filename,'BinTable',4);
ch = char(infofits{1});
% We only care about the sources in Taylor
filt = (ch(:,1)=='T' & ch(:,2)=='a' & ch(:,3)=='y');
l = infofits{2}(filt);
b = infofits{3}(filt);
rm = infofits{4}(filt);
rrm = avgrrmfits{1}(filt);
% Instead of taking the absolute of the average RRM,
% take the average of the absolute RRM.
samplemat = [cell2mat(rrmfits1) cell2mat(rrmfits2)];
samplemat = samplemat(filt,:);
absrrm = median(abs(samplemat), 2);
% Convert to J2000
[coords,~] = coco([l b],'g','J2000.0','d','d');
ra = coords(:,1);
dec = coords(:,2);
% Sort both catalogs by dec and compare
if ~issorted([sources.dec])
    fields = fieldnames(sources);
    sourcesCell = struct2cell(sources);
    sortedCell = sortrows(sourcesCell',2);
    sources = cell2struct(sortedCell',fields,1);
end
[dec,ind] = sort(dec);
ra = ra(ind);
rm = rm(ind);
rrm = rrm(ind);
absrrm = absrrm(ind);

dra = abs(ra-[sources.ra]');
drm = abs(rm-[sources.rm]');
for i = 1:length(dec)
    if dra(i)<0.0005 && drm(i)==0
        sources(i).rrm = rrm(i);
        sources(i).absrrm = absrrm(i);
    end
end
```

```
% Sort by RA and compare
fields = fieldnames(sources);
sourcesCell = struct2cell(sources);
sortedCell = sortrows(sourcesCell',1);
sources = cell2struct(sortedCell',fields,1);
[ra,ind] = sort(ra);
dec = dec(ind);
rm = rm(ind);
rrm = rrm(ind);
absrrm = absrrm(ind);

ddec = abs(dec - [sources.dec]');
drm = abs(rm - [sources.rm]');
for i = 1:length(ra)
    if ddec(i)<0.0005 && drm(i)==0
        if isempty(sources(i).rrm)
            sources(i).rrm = rrm(i);
            sources(i).absrrm = absrrm(i);
        elseif sources(i).rrm ~= rrm(i)
            display('woah woah, something went wrong here')
            keyboard
        end
    end
end

% Sort by RM and compare
fields = fieldnames(sources);
sourcesCell = struct2cell(sources);
sortedCell = sortrows(sourcesCell',6);
sources = cell2struct(sortedCell',fields,1);
[rm,ind] = sort(rm);
dec = dec(ind);
ra = ra(ind);
rrm = rrm(ind);
absrrm = absrrm(ind);

ddec = abs(dec - [sources.dec]');
dra = abs(ra - [sources.ra]');
```

```
    for i = 1:length(rm)
        if ddec(i)<0.005 && dra(i)<0.005
            if isempty(sources(i).rrm)
                sources(i).rrm = rrm(i);
                sources(i).absrrm = absrrm(i);
            elseif sources(i).rrm ~= rrm(i)
                display('woah woah, something went wrong here')
                keyboard
            end
        end
    end
end

function sourcesout = matchToOther(sources, varargin)
% Match sources to NVSS or FIRST to get extents and names.
% Options:
%     other: 'FIRST' or 'NVSS'
%     matchDist: Maximum distance for a match
%     displayMatchProgress
    defaults = struct('other','FIRST', 'matchDist',0,...
                      'displayMatchProgress',1);
    option = makeOptionStruct(defaults, varargin);

    %Sort Taylor sources by dec.
    if ~issorted([sources.dec])
        fields = fieldnames(sources);
        sourcesCell = struct2cell(sources);
        sortedCell = sortrows(sourcesCell',2);
        sources = cell2struct(sortedCell',fields,1);
    end
    sourcesout = sources;
    if strcmp(option.other,'FIRST')
        filename = 'FIRST.fits';
        [sourcesout.FIRSTname] = deal([]);
        [sourcesout.FIRSTnb] = deal([]);
        [sourcesout.FIRSTextent] = deal([]);
    elseif strcmp(option.other,'NVSS')
        filename = 'nvss.fits';
        [sourcesout.NVSSname] = deal([]);
        [sourcesout.NVSSnb] = deal([]);
        [sourcesout.NVSSextent] = deal([]);
        [sourcesout.NVSSextentSmallerThan] = deal([]);
    end
    catalog = fitsread(filename,'BinTable');
```

```
if strcmp(option.other,'FIRST')
    % us for unsorted values, F for FIRST (historical reasons)
    raFus = catalog{3};
    decFus = catalog{4};
    nameFus = catalog{1};
    majaFus = catalog{9};
    sidelobepFus = catalog{5};
    typeFus = catalog{13};
    % Remove stars and probable sidelobes
    filt = true(size(raFus));
    filt = filt & sidelobepFus<0.3;
    filt = filt & char(typeFus)~='s';
    raFus = raFus(filt);
    decFus = decFus(filt);
    nameFus = nameFus(filt);
    majaFus = majaFus(filt);
elseif strcmp(option.other,'NVSS')
    raFus = catalog{2};
    decFus = catalog{4};
    nameFus = catalog{1};
    majaFus = catalog{9};
    limitFlag = char(catalog{8})=='<';
end

% Sort other sources by dec
[decF,ind] = sort(decFus);
raF = raFus(ind);
nameF = nameFus(ind);
majaF = majaFus(ind);

maxdist = option.matchDist;

oldj = 1;% To prevent looping over sources we already
         % checked to be too far away from everything.
if option.displayMatchProgress
    display(['Matching to ' option.other '.'])
end
for i = 1:length(sources)
    if option.displayMatchProgress && mod(i,1000) == 0
        display(['Taylor sources completed: ' num2str(i)])
    end
    raT = sources(i).ra;
    decT = sources(i).dec;
```

```
if decT > decF(1)-maxdist && decT < decF(end)+maxdist
    for j = oldj:length(raF)
        if option.displayMatchProgress && mod(j,10000) == 0
            display([option.other ' sources completed: ' ...
                     num2str(j)])
        end
        if abs(decT-decF(j)) <= maxdist
            oldj = j;
            upperlimit = 0;
            while j+upperlimit < length(decF)...
                     && abs(decT-decF(j+upperlimit)) <= maxdist
                upperlimit = upperlimit + 1;
            end
            % Create filter for the dec range
            filt = zeros(size(raF));
            filt(j:j+upperlimit) = 1;
            filt = logical(filt);
            %Filter out big RA differences
            filt(filt) = abs(raF(filt)-raT) <= maxdist;
            %Finally, only include sources within maxdist
            filt(filt) = ((raF(filt)-raT)*cosd(decT)).^2+...
                         (decF(filt)-decT).^2 <= maxdist^2;
            ind = find(filt);
            r = ((raF(filt)-raT)*cosd(decT)).^2+...
                        (decF(filt)-decT).^2;
            [~,minr] = min(r);
            nameout = nameF(ind(minr));
            nb = sum(filt);
            if nb == 1
                extentout = majaF(filt);
            end
            if nb == 2
                % Could be done more precisely.
                extentout = 3600*sqrt(((raF(ind(1))-raF(ind(2)))...
                                    *cosd(decT))^2+...
                                    (decF(ind(1))-decF(ind(2)))^2);
            end
            if nb > 2
                dists = zeros(nb);
                for k = 1:nb
                    dists(k,:) = sqrt(((raF(ind(k))-raF(ind)).*...
                                       cosd(decT)).^2 + ...
                                       (decF(ind(k))-decF(ind)).^2);
                end
                extentout = max(dists(:))*3600;
            end
```

```
                    if nb>1 && extentout>maxdist*2*3600
                        display('Woah there!')
                        keyboard
                    end
                    if nb > 0
                        if strcmp(option.other,'FIRST')
                            sourcesout(i).FIRSTextent = extentout;
                            sourcesout(i).FIRSTnb = nb;
                            sourcesout(i).FIRSTname = nameout;
                        elseif strcmp(option.other,'NVSS')
                            sourcesout(i).NVSSnb = nb;
                            sourcesout(i).NVSSname = nameout;
                            sourcesout(i).NVSSextent = extentout;
                            sourcesout(i).NVSSextentSmallerThan = ...
                                                        limitFlag(filt);
                        end
                    end
                    break;
                end
            end
        end
    end
end

function sourcesout = chooseExtent(sources, varargin)
% Chooses which to prioritize of NVSS extent and FIRST extent
% Options:
%     notSmallerThan: Delete NVSS extents which are upper limits
%     priorityExtent: FIRST, NVSS, FIRSTonly or NVSSonly. The latter two
%         ignores the other extent altogether.
    defaults = struct('notSmallerThan',1, 'priorityExtent','FIRSTonly');
    option = makeOptionStruct(defaults, varargin);

    if option.notSmallerThan
        filt =~ cellfun('isempty',{sources.NVSSextentSmallerThan})';
        filt(filt) = logical([sources(filt).NVSSextentSmallerThan]');
        [sources(filt).NVSSextent] = deal([]);
    end
    if strcmp(option.priorityExtent,'FIRST')
        firstfilt =~ cellfun('isempty',{sources.FIRSTextent});
        [sources(firstfilt).extent] = sources(firstfilt).FIRSTextent;
        nvssfilt =~ firstfilt & ~cellfun('isempty',{sources.NVSSextent});
        [sources(nvssfilt).extent] = sources(nvssfilt).NVSSextent;
    elseif strcmp(option.priorityExtent,'FIRSTonly')
        firstfilt =~ cellfun('isempty',{sources.FIRSTextent});
        [sources(firstfilt).extent] = sources(firstfilt).FIRSTextent;
```

```
    elseif strcmp(option.priorityExtent,'NVSS')
        nvssfilt =~ cellfun('isempty',{sources.NVSSextent});
        [sources(nvssfilt).extent] = sources(nvssfilt).NVSSextent;
        firstfilt =~ nvssfilt & ~cellfun('isempty',{sources.FIRSTextent});
        [sources(firstfilt).extent] = sources(firstfilt).FIRSTextent;
    elseif strcmp(option.priorityExtent,'NVSSonly')
        nvssfilt =~ cellfun('isempty',{sources.NVSSextent});
        [sources(nvssfilt).extent] = sources(nvssfilt).NVSSextent;
    end
    sourcesout = sources;
end

function galaxies = readGalaxies(from)
% Read galaxies from a file. from can be 'RC3' or 'SINGS'.
    if strcmp(from,'SINGS')
        catalog = fitsread('MunozMateos.fits','BinTable');
        name = catalog{2};
        ra = num2cell(catalog{4});
        dec = num2cell(catalog{5});
        maja = num2cell(catalog{6});
        mina = num2cell(catalog{7});
        pa = catalog{8};
        pa(pa > 180) = pa(pa > 180) -180; % PA between 0 and 180.
        pa = num2cell(pa);
        distance = num2cell(catalog{10});
        z = [];
        T = num2cell(catalog{11});
    elseif strcmp(from,'RC3')
        catalog = fitsread('rc3.fits','BinTable');
        name = catalog{3};
        namechars = char(name);
        % Sometimes there is no name in the first column.
        altnamefilt = isstrprop(namechars(:,1),'cntrl');
        name(altnamefilt) = catalog{4}(altnamefilt);
        ra = num2cell(catalog{1});
        dec = num2cell(catalog{2});
        maja = num2cell(10.^(catalog{8}-1));
        % Calculate minor axis from log of ratio
        mina = num2cell(10.^(-catalog{10}).*[maja{:}]');
        majafilt = isnan(catalog{8}); % Turn NaNs into empty cells
        maja(majafilt) = cell(sum(majafilt),1);
        mina(majafilt) = cell(sum(majafilt),1);
        distance = [];
        z = num2cell(catalog{14}./299792);
```

```
        T = num2cell(catalog{7});
        Tfilt = isnan(catalog{7}); % For T as well
        T(Tfilt) = cell(sum(Tfilt),1);
        pa = num2cell(catalog{11});
        PAfilt = isnan(catalog{11}); % And PA
        pa(PAfilt) = cell(sum(PAfilt),1);
    elseif strcmp(from,'dummy')
        name = {'central'};
        ra = {200};
        dec = {0};
        maja = {120};
        mina ={60};
        pa = {105};
        distance = {5};
        T = {5};
    else
        error('Unknown galaxy catalog!')
    end
    id = num2cell((1:length(ra))');
    galaxies = struct('id',id, 'name',name, 'ra',ra, 'dec',dec, ...
                      'maja',maja, 'mina',mina, 'pa',pa, 'z',z, ...
                      'distance',distance, 'T',T, 'maxRadius',[], ...
                      'ellipticalRnorm',[]);
    % Filter out any without maja.
    galaxies = galaxies(~cellfun('isempty',{galaxies.maja}));
end

function galaxiesout = randomizeGalaxies(galaxies, varargin)
% Randomize locations and PA:s of galaxies.
% Options:
%     Limits on location: [min max]
%       latitude
%       dec
    defaults = struct('latitude',[0 90], 'dec',[-90 90]);
    option = makeOptionStruct(defaults, varargin);
    % The galaxies have been initialized previously,
    % this just changes the locations.
    galaxiesout = galaxies;
    l = rand([length(galaxies) 1])*360;
    b = acosd(2*rand([length(galaxies) 1])-1)-90;
    wrongLat = abs(b)<option.latitude(1) | abs(b)>option.latitude(2);
    while sum(wrongLat) > 0
        b(wrongLat) = acosd(2*rand([sum(wrongLat) 1])-1)-90;
        wrongLat = abs(b)<option.latitude(1) | abs(b)>option.latitude(2);
    end
```

```
    radec = coco([l b],'g','J2000.0','d','d'); % convert to J2000.
    ra = radec(:,1);
    dec = radec(:,2);
    wrongDec = dec < option.dec(1) | dec > option.dec(2);
    while sum(wrongDec) > 0
        l(wrongDec) = rand([sum(wrongDec) 1])*360;
        b(wrongDec) = acosd(2*rand([sum(wrongDec) 1])-1)-90;
        wrongLat = abs(b)<option.latitude(1) | abs(b)>option.latitude(2);
        while sum(wrongLat) > 0
            b(wrongLat) = acosd(2*rand([sum(wrongLat) 1])-1)-90;
            wrongLat = abs(b)<option.latitude(1) | ...
                                                abs(b)>option.latitude(2);
        end
        radec = coco([l b],'g','J2000.0','d','d'); % convert to Galactic.
        ra = radec(:,1);
        dec = radec(:,2);
        wrongDec = dec < option.dec(1) | dec > option.dec(2);
    end

    racell = num2cell(ra);
    deccell = num2cell(dec);

    [galaxiesout(:).ra] = deal(racell{:});
    [galaxiesout(:).dec] = deal(deccell{:});
    pafilt = ~cellfun('isempty',{galaxies(:).pa});
    pas = num2cell(rand(1,sum(pafilt))*180);
    [galaxiesout(pafilt).pa] = deal(pas{:});
end

function galaxies = assignGalaxySourceNb(galaxies, sources)
% Count the number of sources behind each galaxy
    sourceGals = [sources.galaxy];
    sourceGalsID = [sourceGals.id];
    [ids, inds] = sort([galaxies.id]);
    out = num2cell(histc(sourceGalsID(:), ids));
    [galaxies(inds).sourceNb] = deal(out{:});
    ntot = sum([galaxies.sourceNb]);
    display(['Total number of background sources = ' num2str(ntot)]);
end
```

```
function sources = filterSourcesPass1(sources, varargin)
% Filter the sources before matching.
% Options:
%     1 or 0:
%       displayFilterProcess
%       onlyWithZ
%       onlyWithoutZ
%       onlySDSS
%       onlyNED
%       SDSSGal: Galaxy in SDSS
%       NEDGal: Galaxy in NED
%       SDSSQSO: QSO in SDSS
%       NEDQSO: QSO in NED
%       onlyWithExtent
%       onlyWithoutExtent
%     [min max]:
%       latitude
%       ppercent
%       RM
%       absRRM
%       I
%       z: Implies onlyWithZ
%       FIRSTnb: Number of component sources in FIRST
%       extent: Implies onlyWithExtent
    defaults = struct('displayFilterProcess',1,...
                      'latitude', [0 90],...
                      'ppercent', [0 inf],...
                      'RM', [-inf inf],...
                      'absRRM', [0 inf],...
                      'I', [0 inf],...
                      'onlyWithZ', 0,...
                      'onlyWithoutZ', 0,...
                      'z', [-inf inf],...
                      'onlySDSS', 0,...
                      'onlyNED', 0,...
                      'SDSSGal', 0,...
                      'NEDGal', 0,...
                      'SDSSQSO', 0,...
                      'NEDQSO', 0,...
                      'FIRSTnb', [0 inf],...
                      'onlyWithExtent', 0,...
                      'onlyWithoutExtent', 0,...
                      'extent', [0 inf]);
    option = makeOptionStruct(defaults, varargin);
```

```
filt = true(length(sources), 1);
dispopt = option.displayFilterProcess;
if dispopt
    display('---Filtering sources---');
    display(['Starting with ' num2str(sum(filt)) ' sources.']);
end

% Filter sources by galactic latitude
if option.latitude(1) > 0 || option.latitude(2) < 90
    galCo = coco([[sources.ra]' [sources.dec]'],'J2000.0','g','d','d');
    filt = filt & abs(galCo(:,2))<option.latitude(2);
    filt = filt & abs(galCo(:,2))>option.latitude(1);
    if dispopt
        display(['Only looking at sources with '...
                 num2str(option.latitude(1)) '<|b|<' ...
                 num2str(option.latitude(2)) '. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end

% Filter sources by ppercent
if option.ppercent(1) > 0 || ~isinf(option.ppercent(2))
    filt = filt & [sources.ppercent]' < option.ppercent(2);
    filt = filt & [sources.ppercent]' > option.ppercent(1);
    if dispopt
        display(['Only looking at sources with '...
                 num2str(option.ppercent(1)) '<ppercent<'...
                 num2str(option.ppercent(2)) '. '...
                 num2str(sum(filt)) ' left.']);
    end
end
% Filter sources by RM
if ~isinf(option.RM(1)) || ~isinf(option.RM(2))
    filt = filt & [sources.rm]'<option.RM(2);
    filt = filt & [sources.rm]'>option.RM(1);
    if dispopt
        display(['Only looking at sources with ' ...
                 num2str(option.RM(1)) '<RM<' ...
                 num2str(option.RM(2)) '. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
```

```
% Filter sources by absolute RRM
if option.absRRM(1) > 0 || ~isinf(option.absRRM(2))
    filt = filt & [sources.absrrm]'<option.absRRM(2);
    filt = filt & [sources.absrrm]'>option.absRRM(1);
    if dispopt
        display(['Only looking at sources with ' ...
                 num2str(option.absRRM(1)) '<|RRM|<' ...
                 num2str(option.absRRM(2)) '. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end

% Filter sources by I
if option.I(1) > 0 || ~isinf(option.I(2))
    filt = filt & [sources.totali]'<option.I(2);
    filt = filt & [sources.totali]'>option.I(1);
    if dispopt
        display(['Only looking at sources with '...
            num2str(option.I(1)) '<I<' num2str(option.I(2)) '. '...
            num2str(sum(filt)) ' left.']);
    end
end

% Remove sources without z, or outside the z range
if option.onlyWithZ || ~isinf(option.z(1)) || ~isinf(option.z(2))
    filt = filt & (~cellfun('isempty',{sources.z}))';
    filt(filt) = filt(filt) & ([sources(filt).z]' < option.z(2)) & ...
        ([sources(filt).z]' > option.z(1));
    if dispopt
        display(['Including sources with ' ...
            num2str(option.z(1)) '<z<' num2str(option.z(2)) '. ' ...
            num2str(sum(filt)) ' left.']);
    end
end
%Remove sources with z
if option.onlyWithoutZ
    filt  =filt & (cellfun('isempty',{sources.z}))';
    if dispopt
        display(['Removing sources with z. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
```

```
% Filter by catalog/survey
if option.onlySDSS
    filt = filt & ~cellfun('isempty',{sources.SDSStype})';
    if dispopt
        display(['Only looking at sources in SDSS. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
if option.onlyNED
    filt = filt & ~cellfun('isempty',{sources.NEDname})';
    if dispopt
        display(['Only looking at sources in NED. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
% Look at galaxies
if option.SDSSGal
    filt = filt & ~cellfun('isempty',{sources.SDSStype})';
    type = char([sources(filt).SDSStype]');
    filt(filt) = filt(filt) & type(:,1)=='G';
    if dispopt
        display(['Only looking at SDSS galaxies. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end

if option.NEDGal
    filt = filt & ~cellfun('isempty',{sources.NEDtype})';
    type = char([sources(filt).NEDtype]');
    filt(filt) = filt(filt) & (type(:,1)=='G') & (type(:,2)~='P') & ...
                              (type(:,2)~='G') & (type(:,2)~='T');
    if dispopt
        display(['Only looking at NED galaxies. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
% Look at QSOs
if option.SDSSQSO
    filt = filt & ~cellfun('isempty',{sources.SDSStype})';
    type = char([sources(filt).SDSStype]');
    filt(filt) = filt(filt) & type(:,1)=='Q';
    if dispopt
        display(['Only looking at SDSS QSOs. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
```

```
if option.NEDQSO
    filt = filt & ~cellfun('isempty',{sources.NEDtype})';
    type = char([sources(filt).NEDtype]');
    filt(filt) = filt(filt) & type(:,1)=='Q' & type(:,2)=='S';
    if dispopt
        display(['Only looking at NED QSOs. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
% Only include sources that correspond to
% a certain number of FIRST sources.
if option.FIRSTnb(1) > 0 || ~isinf(option.FIRSTnb(2))
    filt = filt & ~cellfun('isempty',{sources.FIRSTnb})';
    filt(filt) = filt(filt) & ...
                    [sources(filt).FIRSTnb]' >= option.FIRSTnb(1);
    filt(filt) = filt(filt) & ...
                    [sources(filt).FIRSTnb]' <= option.FIRSTnb(2);
    if dispopt
        display(['Only looking at sources with number of parts ' ...
                 num2str(option.FIRSTnb(1)) ' < nb < ' ...
                 num2str(option.FIRSTnb(2)) '. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
%Only include sources with known angular size, and maybe some limits
if option.onlyWithExtent || option.extent(1) > 0 || ...
                                            ~isinf(option.extent(2))
    filt = filt & ~cellfun('isempty',{sources.extent})';
    filt(filt) = filt(filt) & [sources(filt).extent]'<option.extent(2);
    filt(filt) = filt(filt) & [sources(filt).extent]'>option.extent(1);
    if dispopt
        display(['Only looking at sources with angular size ' ...
                 num2str(option.extent(1)) ' < extent < ' ...
                 num2str(option.extent(2)) '. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
%Only include sources without known angular size
if option.onlyWithoutExtent
    filt = filt & cellfun('isempty',{sources.extent})';
    if dispopt
        display(['Only looking at sources without angular size. ' ...
                 num2str(sum(filt)) ' left.']);
    end
end
```

```
    if dispopt
        display(' ');
    end

    sources = sources(filt);
end

function sources = filterSourcesPass2(sources, varargin)
% Filter the sources after galaxies have been assigned.
% Options:
%     displayFilterProcess
%     L: Extent of the region that the source radiation passes through,
%         at the distance of the galaxy. In pc.
%     theta: Position angle relative to the galaxy.
%     relTheta: Angle between galaxy angle and line between galaxy and
%         source.
    defaults = struct('displayFilterProcess',1,...
                      'L', [0 inf],...
                      'theta', [0 360],...
                      'relTheta', [0 90]);
    option = makeOptionStruct(defaults, varargin);

    filt = true(1, length(sources));
    dispopt = option.displayFilterProcess;
    if dispopt
        display(['Number of sources behind galaxies: ' ...
                 num2str(length(sources))])
    end
    if option.L(1) > 0 || ~isinf(option.L(2))
        filt = filt & ~cellfun('isempty',{sources.extent});
        gals = [sources.galaxy];
        filt(filt) = [gals(filt).distance].*...
            [sources(filt).extent]*pi*10^6/180/3600 >= option.L(1);
        filt(filt) = [gals(filt).distance].*...
            [sources(filt).extent]*pi*10^6/180/3600 <= option.L(2);
        if dispopt
            display(['Filtering so that ' num2str(option.L(1)) ...
                     '<= L <= ' num2str(option.L(2)) ...
                     '. ' num2str(sum(filt)) ' left.'])
        end
    end
```

```
    if option.theta(1) > 0 || option.theta(2) < 360
        filt = filt & ~cellfun('isempty',{sources.theta});
        filt(filt) = [sources(filt).theta] >= option.theta(1) & ...
            [sources(filt).theta] <= option.theta(2);
        if dispopt
            display(['Filtering so that ' num2str(option.theta(1)) ...
                '<=theta<=' num2str(option.theta(2)) '. ' ...
                num2str(sum(filt)) ' left.'])
        end
    end
    if option.relTheta(1) > 0 || option.relTheta(2) < 90
        filt = filt & ~cellfun('isempty',{sources.relTheta});
        filt(filt) = [sources(filt).relTheta] >= option.relThet(1) & ...
            [sources(filt).relTheta] <= option.relTheta(2);
        if dispopt
            display(['Filtering so that ' num2str(option.relTheta(1)) ...
                '<=relative angle<=' num2str(option.relTheta(2)) '. ' ...
                num2str(sum(filt)) ' left.'])
        end
    end
    sources = sources(filt);
end

function galaxiesout = filterGalaxies(galaxies, varargin)
% Filter the galaxies.
% Options:
%     1 or 0:
%       displayFilterProcess
%       onlyWithT
%       excludeT: Excludes galaxies in the T range. Implies onlyWithT.
%     [min max]:
%       latitude
%       dec
%       T: Implies onlyWithT
%       distance
%       z
%       minMajFrac: minor axis / major axis
%       maja: Major axis
    defaults = struct('displayFilterProcess', 1,...
                      'latitude', [0 90],...
                      'dec', [-90 90],...
                      'onlyWithT', 0,...
                      'excludeT', 0,...
                      'T', [-inf inf],...
                      'distance', [-inf inf],...
                      'z', [-inf inf],...
```

```
                'minMajFrac', [0 1],...
                'maja', [0 inf]);
option = makeOptionStruct(defaults, varargin);

filt = true(length(galaxies),1);

dispopt = option.displayFilterProcess;
if dispopt
    display('---Filtering galaxies---');
    display(['Starting with ' num2str(sum(filt)) ' galaxies.']);
end
if option.latitude(1) > 0 || option.latitude(2) < 90 ||...
                  option.dec(1) > -90 || option.dec(2) < 90
    filt = filt & [galaxies.dec]' > option.dec(1);
    filt = filt & [galaxies.dec]' < option.dec(2);
    galCo = coco([[galaxies.ra]' [galaxies.dec]'], 'J2000.0', ...
                                                    'g', 'd', 'd');
    filt = filt & abs(galCo(:,2))>option.latitude(1);
    filt = filt & abs(galCo(:,2))<option.latitude(2);
    if dispopt
        display(['Filtering galaxies by location. '...
                 num2str(sum(filt)) ' galaxies.'])
    end
end
% Filter by Hubble T
if option.onlyWithT || ~isinf(option.T(1)) || ~isinf(option.T(2))
    filt = filt & ~cellfun('isempty',{galaxies.T})';
    tmpfilt = filt;
    tmpfilt(tmpfilt) = tmpfilt(tmpfilt) & ...
                       [galaxies(tmpfilt).T]' <= option.T(2);
    tmpfilt(tmpfilt) = tmpfilt(tmpfilt) & ...
                       [galaxies(tmpfilt).T]' >= option.T(1);
    if option.excludeT
        filt = filt & ~tmpfilt;
        if dispopt
            display(['Not looking at galaxies with ' ...
                     num2str(option.T(1)) '<=T<=' ...
                     num2str(option.T(2)) '. ' ...
                     num2str(sum(filt)) ' galaxies.'])
        end
```

```
    else
        filt = filt & tmpfilt;
        if dispopt
            display(['Only looking at galaxies with ' ...
                     num2str(option.T(1)) '<=T<=' ...
                     num2str(option.T(2)) '. ' ...
                     num2str(sum(filt)) ' galaxies.'])
        end
    end
end
% Filter by distance
if ~isinf(option.distance(1)) || ~isinf(option.distance(2))
    filt = filt & [galaxies.distance]' >= option.distance(1);
    filt = filt & [galaxies.distance]' <= option.distance(2);
    if dispopt
        display(['Only include galaxies between ' ...
            num2str(option.distance(1)) ' and ' ...
            num2str(option.distance(2)) ' Mpc away. ' ...
            num2str(sum(filt)) ' galaxies.'])
    end
end
if ~isinf(option.z(1)) || ~isinf(option.z(2))
    filt = filt & [galaxies.z]' >= option.z(1);
    filt = filt & [galaxies.z]' <= option.z(2);
    if dispopt
        display(['Only include galaxies  with z between ' ...
            num2str(option.z(1)) ' and ' ...
            num2str(option.z(2)) '. ' ...
            num2str(sum(filt)) ' galaxies.'])
    end
end
% mina/maja
if option.minMajFrac(1) > 0 || option.minMajFrac(2) < 1
    frac = [galaxies.mina]'./[galaxies.maja]';
    filt = filt & frac >= option.minMajFrac(1);
    filt = filt & frac <= option.minMajFrac(2);
    if dispopt
        display(['Only include galaxies with mina/maja between ' ...
                 num2str(option.minMajFrac(1)) ' and ' ...
                 num2str(option.minMajFrac(2)) '. ' ...
                 num2str(sum(filt)) ' galaxies.'])
    end
end
```

```
    if option.maja(1) > 0 || ~isinf(option.maja(2))
        filt = filt & [galaxies.maja]' >= option.maja(1);
        filt = filt & [galaxies.maja]' <= option.maja(2);
        if dispopt
            display(['Only include galaxies between ' ...
                num2str(option.maja(1)) ' and ' num2str(option.maja(2)) ...
                ' arcmin on the sky. ' num2str(sum(filt)) ' galaxies.'])
        end
    end

    if dispopt
        display(' ')
    end
    galaxiesout = galaxies(filt);
end

function [sources, galaxies] = findsources(sources, galaxies, varargin)
% Assigns galaxies to sources.
% Options:
%     xrc: Maximum distance from galaxy, in galaxy major axes.
%     excrc: Minimum distance from galaxy, in galaxy major axes.
%     averageMajMin: Instead of galaxy major axes as units, average major
%         and minor.
%     ellipseApprox: Assume that an RA difference means the same thing in
%         terms of distance on the sky for all positions near one galaxy.
%     ellipticalRnorm
    defaults = struct('xrc',0, 'excrc',0, 'averageMajMin',0, ...
                      'ellipseApprox',0, 'ellipticalRnorm',0);
    option = makeOptionStruct(defaults, varargin);

    xrc = option.xrc;
    excrc = option.excrc;

    % Sort sources by dec
    if ~issorted([sources.dec])
        fields = fieldnames(sources);
        sourcesCell = struct2cell(sources);
        sortedCell = sortrows(sourcesCell', 2);
        sources = cell2struct(sortedCell', fields, 1);
    end
    % Sort galaxies by lowest dec of searched region
    lowerbound = [galaxies(:).dec] - xrc*0.5*[galaxies(:).maja]/60;
```

```
if ~issorted(lowerbound)
    lowerbound = num2cell(lowerbound);
    [galaxies(:).lowerbound] = deal(lowerbound{:});
    fields = fieldnames(galaxies);
    galaxiesCell = struct2cell(galaxies);
    % lowerbound is the last row.
    sz = size(galaxiesCell);
    sortedCell = sortrows(galaxiesCell',sz(1));
    galaxies = cell2struct(sortedCell',fields,1);
    galaxies = rmfield(galaxies,'lowerbound');
end
[sources.galaxyNb] = deal(0);
oldj = 1;
display('Finding sources:')
for i = 1:length(galaxies)
    if mod(i,1000) == 0
        display(['Galaxy ' num2str(i)])
    end
    galaxy = galaxies(i);

    racluster = galaxy.ra;
    deccluster = galaxy.dec;
    rmaj = galaxy.maja/2;
    rmin = galaxy.mina/2;
    pa = galaxy.pa;
    if option.averageMajMin
        rc = (rmaj + rmin)/2/60; % average of the 2, in deg
    else
        rc = rmaj/60;
    end

    outRadius = xrc*rc; % in deg

    galaxy.maxRadius = outRadius;
    galaxy.ellipticalRnorm = option.ellipticalRnorm;
    galaxies(i) = galaxy;

    % No sense in searching already searched dec, since we sorted
    for j = oldj:length(sources)
        if deccluster - sources(j).dec < outRadius
            oldj = j; % We will break out of the loop this iteration.
            upperlimit = 0;
```

```
% Find the other end of the dec range to search.
while j+upperlimit < length(sources) && ...
      sources(j+upperlimit).dec - deccluster < outRadius
    upperlimit = upperlimit + 1;
end
decfilt = false(length(sources),1);
decfilt(j:j+upperlimit) = true;
% Compute the distance
if option.ellipseApprox
    deltara1 = [sources(decfilt).ra]' - racluster;
    for k = 1:length(deltara1)
        if deltara1(k) > 180
            deltara1(k) = deltara1(k) - 360;
        elseif deltara1(k) < -180
            deltara1(k) = deltara1(k) + 360;
        end
    end
    deltara = deltara1.*cos(deccluster/180*pi);
    deltadec = [sources(decfilt).dec]' - deccluster;
    deltar = sqrt(deltara.^2 + deltadec.^2);
else
    b1 = deccluster;
    b2 = [sources(decfilt).dec]';
    l1 = racluster;
    l2 = [sources(decfilt).ra]';
    % Spherical law of cosines:
    deltar = acosd(sind(b1).*sind(b2) ...
                   + cosd(b1).*cosd(b2).*cosd(l1-l2));
end
smallfilt = (deltar < outRadius);
filt = decfilt;
filt(decfilt) = smallfilt;
% Compute theta
if sum(smallfilt) ~= 0
    if option.ellipseApprox
        num = deltara(smallfilt);
        denom = deltadec(smallfilt);
        theta = atan(num./denom)/pi*180 + (denom<0)*180;
        theta(theta<0) = theta(theta<0)+360;
```

```
else
    b1 = deccluster;
    b2 = [sources(filt).dec]';
    num = sind(b2)-cosd(deltar(smallfilt)).*sind(b1);
    denom = sind(deltar(smallfilt)).*cosd(b1);
    theta = real(acosd(num./denom));
    negativeTheta = [sources(filt).ra]'-racluster < 0;
    theta(negativeTheta) = 360 - theta(negativeTheta);
end
thetacell = num2cell(theta);
relTheta = abs(theta-pa); % Angle from galactic disk
relTheta(relTheta>180) = relTheta(relTheta>180)-180;
relTheta(relTheta>90) = 180-relTheta(relTheta>90);
relThetaCell = num2cell(relTheta);
rcell = num2cell(deltar(smallfilt));
if option.ellipticalRnorm
    rma = rmaj/60;
    rmi = rmin/60;
    rc = 1./sqrt((cosd(pa-theta)./rma).^2 ...
                + (sind(pa-theta)./rmi).^2);
    rnorm = num2cell(deltar(smallfilt)./rc);
else
    rnorm = num2cell(deltar(smallfilt)/rc);
end
ind = find(filt);
for k = 1:sum(smallfilt)
    % To not include too many sources if
    % using elliptical rnorm, and also excluding
    % sources that are too close.
    if rnorm{k} < xrc && rnorm{k} > excrc
        % Add galaxy
        if isempty(sources(ind(k)).galaxy)
            sources(ind(k)).galaxy = galaxy;
        else
            sources(ind(k)).galaxy(end+1) = galaxy;
        end
        %Add theta
        sources(ind(k)).theta(end+1) = thetacell{k};
        %Add relTheta
        sources(ind(k)).relTheta(end+1) = ...
                                        relThetaCell{k};
        %Add r
        sources(ind(k)).r(end+1) = rcell{k};
        %Add rnorm
        sources(ind(k)).rnorm(end+1) = rnorm{k};
```

```
                        %Increase galaxyNb
                        sources(ind(k)).galaxyNb = ...
                                        sources(ind(k)).galaxyNb+1;
                    end
                end
            end
            break;
        end
    end
end

function galaxySources = getGalaxySources(sources, varargin)
% Filters down to the sources behind galaxies
% Options:
%     duplicateDoubleSources: If not, double sources are removed.
    defaults = struct('duplicateDoubleSources',0);
    option = makeOptionStruct(defaults, varargin);

    filt = [sources.galaxyNb]'>0;
    if ~option.duplicateDoubleSources
        filt2 = filt & [sources.galaxyNb]'==1;
        display(['Removing ' num2str(sum(filt) - sum(filt2)) ...
                 ' double sources'])
        filt = filt2;
        galaxySources = sources(filt);
    else
        galaxySources = sources(filt);
        indexMap = find([galaxySources.galaxyNb]'>1);
        display(['Duplicating ' num2str(length(indexMap)) ...
                 ' double sources'])
        for i = 1:length(indexMap)
            if mod(i,1000) == 0
                display(['Source ' num2str(i)])
            end
            source = galaxySources(indexMap(i));
            for j = 1:source.galaxyNb
                newsource = source;
                newsource.galaxyNb = 1;
                newsource.galaxy = source.galaxy(j);
                newsource.theta = source.theta(j);
                newsource.relTheta = source.relTheta(j);
                newsource.r = source.r(j);
                newsource.rnorm = source.rnorm(j);
```

```
                if j == 1
                    galaxySources(indexMap(i)) = newsource;
                else
                    galaxySources(end+1) = newsource; %#ok<AGROW>
                    % No way of preallocating
                end
            end
        end
    end

end

function trials = prepareTrials(galaxies, galaxyLocations, varargin)
% Initialize trials. The first trial is always the real one.
% Options:
%     amountOfRandomizations: Number of additional trials with random
%         galaxies.
    defaults = struct('amountOfRandomizations',0);
    option = makeOptionStruct(defaults, varargin);
    quantities = struct('totali',[], 'poli',[], 'ppercent',[], 'rm',[], ...
                        'rrm',[], 'absrrm',[], 'number',[], 'density',[]);
    trials = struct('galaxySet',[], 'rnorm',[], 'totali',[], ...
                    'poli',[], 'ppercent',[], 'rm',[], 'rrm',[], ...
                    'absrrm',[], 'inside',quantities,'outside',quantities);
    trials(1).galaxySet = galaxies;
    if option.amountOfRandomizations > 0
        display('Randomizing galaxies')
        for i = 1:option.amountOfRandomizations
            if mod(i,10) == 0
                display(['trial ' num2str(i)])
            end
            trials(i+1).galaxySet = randomizeGalaxies(galaxies, ...
                                                      galaxyLocations{:});
            trials(i+1).inside = quantities;
            trials(i+1).outside = quantities;
        end
    end
end
```

```
function trial = saveTrial(trial, sources)
% Saves the data from a trial.
    fields = fieldnames(trial);

    for i=1:length(fields)
        field = fields{i};
        if ~strcmp(field,'galaxySet') && ~strcmp(field,'inside') ...
                                      && ~strcmp(field,'outside');
            trial.(field) = [sources.(field)];
        end
    end
end

function trials = analyzeTrials(trials, varargin)
% Calculate the mean quantities (as well as source number and
% number/rnorm^2) in the two rnorm regions given by the options 'inside'
% and 'outside'.
    defaults = struct('inside',[1 10],'outside',[10 20]);
    option = makeOptionStruct(defaults, varargin);

    for i = 1:length(trials)
        trial = trials(i);
        insidefilt = trial.rnorm <= option.inside(2);
        insidefilt = insidefilt & trial.rnorm >= option.inside(1);
        outsidefilt = trial.rnorm >= option.outside(1);
        outsidefilt = outsidefilt & trial.rnorm <= option.outside(2);
        fields = fieldnames(trial.inside);
        for j=1:length(fields)
            field = fields{j};
            if ~strcmp(field,'number') && ~strcmp(field,'density')
                quantity = trial.(field);
                trial.inside.(field) = median(quantity(insidefilt));
                trial.outside.(field) = median(quantity(outsidefilt));
            end
        end
        inNum = sum(insidefilt);
        outNum = sum(outsidefilt);
        trial.inside.number = inNum;
        trial.outside.number = outNum;
        inArea = pi*(option.inside(2)^2-option.inside(1)^2);
        outArea = pi*(option.outside(2)^2-option.outside(1)^2);
        trial.inside.density = inNum/inArea;
        trial.outside.density = outNum/outArea;
        trials(i) = trial;
    end
end
```

```
function higherFrac = trialModeResults(trials, quantity, metric)
% Display trial mode statistics and plot Monte Carlo.
% quantity is the quantity to analyze: 'totali', 'poli', 'ppercent', 'rm',
%     'rrm', 'absrrm', 'z', 'extent', 'number', 'density'.
% metric is the metric to use: 'difference', 'fraction', 'inside'.
% Returns higherFrac: Fraction of random trials where metric is higher than
%     in real trial.
    hold on
    inside = [trials.inside];
    outside = [trials.outside];
    if strcmp(metric,'difference')
        display([quantity ' inside - outside:'])
        %real = inside(1).(quantity) - outside(1).(quantity);
        real = inside(1).(quantity) - outside(1).(quantity);
        random = [inside(2:end).(quantity)] ...
                 - [outside(2:end).(quantity)];
    elseif strcmp(metric,'inside')
        display([quantity ' inside:'])
        real = inside(1).(quantity);
        random = [inside(2:end).(quantity)];
    elseif strcmp(metric,'outside')
        display([quantity ' outside:'])
        real = outside(1).(quantity);
        random = [outside(2:end).(quantity)];
    elseif strcmp(metric,'fraction')
        display([quantity ' inside / outside'])
        real = inside(1).(quantity)./outside(1).(quantity);
        random = [inside(2:end).(quantity)] ...
                 ./[outside(2:end).(quantity)];
    end
    display(['Real: ' num2str(real)])
    display(['Mean random: ' num2str(mean(random))])
    display(['Std of random: ' num2str(std(random))])
    higherRandom = sum(random>real);
    higherFrac = higherRandom/(length(trials)-1);
    display(['Real is lower than in ' num2str(higherRandom) ...
        ' trials. That is ' num2str(higherFrac*100) '%.'])
    histogram(random)
    plot([real,real],ylim,'color','r')
    xlabel([quantity ' ' metric])
end
```

```
function displayCatalogStats(sources)
% Displays statistics of the sources
    numsources = length(sources);
    meanp = mean([sources.ppercent]);
    stdp = std([sources.ppercent]);
    medianp = median([sources.ppercent]);
    meanrm = mean([sources.rm]);
    stdrm = std([sources.rm]);
    medianrm = median([sources.rm]);
    meanrrm = mean([sources.rrm]);
    stdrrm = std([sources.rrm]);
    medianrrm = median([sources.rrm]);
    meanAbsrrm = mean([sources.absrrm]);
    medianAbsrrm = median([sources.absrrm]);
    stdAbsrrm = std([sources.absrrm]);
    extentFilt = ~cellfun('isempty',{sources.extent});
    meanExtent = mean([sources(extentFilt).extent]);
    medianExtent = median([sources(extentFilt).extent]);
    stdExtent = std([sources(extentFilt).extent]);
    display('---Sample statistics---');
    display(['Number of sources = ' num2str(numsources)]);
    display(['Mean p = ' num2str(meanp)]);
    display(['Median p = ' num2str(medianp)]);
    display(['Std of p = ' num2str(stdp)]);
    display(['Mean RM = ' num2str(meanrm)]);
    display(['Median RM = ' num2str(medianrm)]);
    display(['Std of RM = ' num2str(stdrm)]);
    display(['Mean RRM = ' num2str(meanrrm)]);
    display(['Median RRM = ' num2str(medianrrm)]);
    display(['Std of RRM = ' num2str(stdrrm)]);
    display(['Mean absolute RRM = ' num2str(meanAbsrrm)]);
    display(['Median absolute RRM = ' num2str(medianAbsrrm)]);
    display(['Std of absolute RRM = ' num2str(stdAbsrrm)]);
    display(['Mean extent = ' num2str(meanExtent)]);
    display(['Median extent = ' num2str(medianExtent)]);
    display(['Std of extent = ' num2str(stdExtent)]);
    display(' ');
end
```

```
function displayOutliers(sources, galaxies, varargin)
% Displays any sources or galaxies with quantities above the specified
% limits
% Options:
%     ppercentLimit: For sources.
%     sourceNbLimit: For galaxies.
    defaults = struct('ppercentLimit',inf, 'sourceNbLimit',inf);
    option = makeOptionStruct(defaults, varargin);

    filt = [sources.ppercent] >= option.ppercentLimit;
    src = sources(filt);
    for i = 1:sum(filt)
        if ~isempty(src(i).SIMBADname)
            display(['p=' num2str(src(i).ppercent) ' ' ...
                     char(src(i).SIMBADname)])
        else
            display(['p=' num2str(src(i).ppercent)...
                ' Unknown, RA: ' num2str(src(i).ra) ...
                ' dec: ' num2str(src(i).dec)])
        end
    end

    filt = [galaxies.sourceNb] >= option.sourceNbLimit;
    gal = galaxies(filt);
    for i = 1:sum(filt)
        display([gal(i).name ' maja=' num2str(gal(i).maja) ...
                             ' nb=' num2str(gal(i).sourceNb)])
    end
end

function plotSkyMap(sources, galaxySources, galaxies, varargin)
% Plots the sources and galaxies with examined regions
% Options:
%     extraVisibleBehind: Make the sources behind galaxies larger
%     galaxyEllipseFactor: How much larger than the galaxy to draw the
%         ellipse
    hold on
    defaults = struct('extraVisibleBehind',0,'galaxyEllipseFactor',10);
    option = makeOptionStruct(defaults, varargin);
    plot([sources.ra],[sources.dec],'k.','MarkerSize',2);
    plot([galaxies.ra],[galaxies.dec],'r.','MarkerSize',8);
    set(gca,'XDir','reverse');
    axis equal;
    axis([0 360 -40 90]);

    xlabel('RA [deg]'); ylabel('Dec [deg]')
```

```
    for i = 1:length(galaxies)  % 1 foreground galaxy at a time
        galaxy = galaxies(i);
        % Display the location of the galaxies on the sky
        rma = option.galaxyEllipseFactor*galaxy.maja/120;
        rmi = option.galaxyEllipseFactor*galaxy.mina/120;
        f = 1/cosd(galaxy.dec);
        pa = galaxy.pa;
        r1 = rma*sqrt(cosd(pa)^2*f^2+sind(pa)^2);
        r2 = rmi*sqrt(sind(pa)^2*f^2+cosd(pa)^2);
        ang = pi/2 - acos(rma*cosd(pa)/r1);
        ellipse(r1,r2,ang,galaxy.ra,galaxy.dec,'r--',300);
        if galaxy.ra+rma > 360
            ellipse(r1,r2,ang,galaxy.ra-360,galaxy.dec,'r--',300);
        end
        if galaxy.ra-rma < 0
            ellipse(r1,r2,ang,galaxy.ra+360,galaxy.dec,'r--',300);
        end

    end
    if option.extraVisibleBehind
        plot([galaxySources.ra],[galaxySources.dec],'b.','MarkerSize',3);
    else
        plot([galaxySources.ra],[galaxySources.dec],'b.','MarkerSize',2);
    end
end

function sourcePlot(sources, X, Y, varargin)
% Plots arbitrary Y against arbitrary X.
% X and Y are field names in sources.
% Options:
%     numberOfBins
%     transform: Transform to apply for the statistics when binning.
    hold on
    defaults = struct('numberOfBins',10, 'transform','none', ...
                      'leftEdge',nan, 'rightEdge',nan);
    option = makeOptionStruct(defaults, varargin);

    xFilt = ~cellfun('isempty',{sources.(X)});
    yFilt = ~cellfun('isempty',{sources.(Y)});
    filt = xFilt & yFilt;
    x = [sources(filt).(X)];
    y = [sources(filt).(Y)];
```

```
    plot(x, y,'.k')
    [cy, ly, uy, cx] = binData(x, y, option.numberOfBins, ...
                                    'transform', option.transform,...
                                    'leftEdge', option.leftEdge,...
                                    'rightEdge', option.rightEdge);
    errorbar(cx,cy,ly,uy,'r.', 'MarkerSize',10);
    ylabel(Y)
    xlabel(X)
end

function [binValues, lowerLimits, upperLimits, centers, nums, edges] = ...
                                            binData(x, y, nbin, varargin)
% Helper function to bin data.
% Options:
%     constantBinWidthSquared: Useful for e.g. having bins with constant
%         area in plots vs r.
%     leftEdge
%     rightEdge
%     transform: Transform to apply to y before calculating statistics and
%         transforming back. 'sqrt', 'log' or 'none'
    defaults = struct('constantBinWidthSquared',0, 'leftEdge',nan, ...
                      'rightEdge',nan, 'transform','none');
    option = makeOptionStruct(defaults, varargin);
    if isnan(option.leftEdge)
        start = min(x);
    else
        start = option.leftEdge;
    end
    if isnan(option.rightEdge)
        stop = max(x);
    else
        stop = option.rightEdge;
    end
    if option.constantBinWidthSquared
        edges = sqrt(linspace(start.^2,stop.^2,nbin+1));
    else
        edges = linspace(start,stop,nbin+1);
    end
    bin = discretize(x, edges);
    binValues = zeros(1,nbin);
    lowerLimits = zeros(1,nbin);
    upperLimits = zeros(1,nbin);
    nums = zeros(1,nbin);
    centers = edges(1:length(edges)-1) + diff(edges)/2;
```

```
    for i=1:nbin
        z = y(bin==i);
        binValues(i) = mean(z);
        SEM = std(z)/sqrt(length(z));
        lowerLimits(i) = SEM;
        upperLimits(i) = SEM;
        nums(i) = length(z);
    end
end

function plotRRMbyLat(sources)
% Plots RRM by galactic latitude, to see if the reconstruction is sound.
    hold on
    galCo = coco([[sources.ra]' [sources.dec]'],'J2000.0','g','d','d');
    plot(abs(galCo(:,2)), [sources.rrm]','.')
    p = polyfit(abs(galCo(:,2)), [sources.rrm]',1);
    plot(abs(galCo(:,2)),p(1).*abs(galCo(:,2))+p(2),'color','r')
    display('Linear fit of rrm by galactic latitude:')
    display(['k=' num2str(p(1)) ' m=' num2str(p(2)) ])
end

function plotByFractionalRadius(fig, sources, allSources, varargin)
% Plots ppercent, rrm, absrrm and totali agains rnorm, along with the
% number of sources in each bin.
% Options:
%     numberOfBins
%     binByArea: Bins have constant area instead of constant rnorm diff.
%     rnormLine: Plot a vertical line at the specified rnorm.
    defaults = struct('numberOfBins',10,'binByArea',0,'rnormLine',nan);
    option = makeOptionStruct(defaults, varargin);

    figure(fig)
    clf

    rnorms = [sources.rnorm];
    ppercents = [sources.ppercent];
    polis = [sources.poli];
    absrrms = [sources.absrrm];
    totalis = [sources.totali];
```

```
subplot(231) % ppercent
%plot(rnorms,ppercents,'r.', 'MarkerSize',2)
hold on
[cy, ly, uy, cx] = binData(rnorms, ppercents, option.numberOfBins,...
         'constantBinWidthSquared',option.binByArea);
errorbar(cx,cy,ly,uy,'b.', 'MarkerSize',10);
commonx = xlim;
commony = [mean([allSources.ppercent]),...
           mean([allSources.ppercent])];
plot(commonx, commony, 'k:') % dotted line
xlabel('R/Rc'); ylabel('p [%]'); axis tight
if ~isnan(option.rnormLine)
    plot([option.rnormLine option.rnormLine],ylim,'k--')
end

subplot(232) % poli
%plot(rnorms,polis,'r.', 'MarkerSize',2)
hold on
[cy, ly, uy, cx] = binData(rnorms, polis, option.numberOfBins,...
         'constantBinWidthSquared',option.binByArea);
errorbar(cx,cy,ly,uy,'b.', 'MarkerSize',10);
commonx = xlim;
commony = [mean([allSources.poli]), mean([allSources.poli])];
plot(commonx, commony, 'k:') % dotted line
xlabel('R/Rc'); ylabel('PI [mJy]'); axis tight
if ~isnan(option.rnormLine)
    plot([option.rnormLine option.rnormLine],ylim,'k--')
end

subplot(233) % absrrm
%plot(rnorms,absrrms,'r.', 'MarkerSize',2)
hold on
[cy, ly, uy, cx] = binData(rnorms, absrrms, option.numberOfBins,...
         'constantBinWidthSquared',option.binByArea);
errorbar(cx,cy,ly,uy,'b.', 'MarkerSize',10);
commonx = xlim;
commony = [mean([allSources.absrrm]), mean([allSources.absrrm])];
plot(commonx, commony, 'k:') % dotted line
xlabel('R/Rc'); ylabel('|RRM| [rad m^{-2}]'); axis tight
if ~isnan(option.rnormLine)
    plot([option.rnormLine option.rnormLine],ylim,'k--')
end
```

```
    subplot(234) % totali
    %plot(rnorms,totalis,'r.', 'MarkerSize',2)
    hold on
    [cy, ly, uy, cx] = binData(rnorms, totalis, option.numberOfBins,...
             'constantBinWidthSquared',option.binByArea);
    errorbar(cx,cy,ly,uy,'b.', 'MarkerSize',10);
    commonx = xlim;
    commony = [mean([allSources.totali]), ...
               mean([allSources.totali])];
    plot(commonx, commony, 'k:') % dotted line
    xlabel('R/Rc'); ylabel('I [mJy]'); axis tight
    if ~isnan(option.rnormLine)
        plot([option.rnormLine option.rnormLine],ylim,'k--')
    end

    % Number in each bin and source density
    subplot(235)
    hold on
    [~, ~, ~, cx, num, edges] = binData(rnorms, rnorms,...
          option.numberOfBins, 'constantBinWidthSquared',option.binByArea);
    bar(cx, num);
    axis tight;
    xlim(commonx);
    xlabel('R/R_c')
    ylabel('Amount of sources')
    if ~isnan(option.rnormLine)
        plot([option.rnormLine option.rnormLine],ylim,'k--')
    end

    subplot(236)
    hold on
    circleAreas = pi*edges.^2;
    regionAreas = diff(circleAreas);
    dens = num./regionAreas;
    bar(cx, dens);
    axis tight;
    xlim(commonx);
    xlabel('R/R_c')
    ylabel('Sources/r_{norm}^2')
    if ~isnan(option.rnormLine)
        plot([option.rnormLine option.rnormLine],ylim,'k--')
    end
end
```

```
function sourceHistograms(sources)
% Plots distributions of ppercent, RRM, absolute RRM and extent.
    subplot(231)
    histogram([sources.ppercent])
    xlabel('Degree of polarization (%)')
    subplot(232)
    histogram([sources.absrrm]);
    xlabel('|RRM| (rad/m^2)')
    subplot(233)
    histogram([sources.rrm]);
    xlabel('RRM (rad/m^2)')
    subplot(234)
    histogram([sources(~cellfun('isempty',{sources.extent})).extent]);
    xlabel('Extent (as)')
    subplot(235)
    histogram([sources.totali]);
    xlabel('I (mJy)')
    subplot(236)
    histogram([sources.poli]);
    xlabel('PI (mJy)')
end

function galaxyHistograms(galaxies,varargin)
% Plots distributions of galaxy maja and distance or z
% Options:
%     rc3mode: To use with RC3 galaxies, makes the maja histogram clearer.
    defaults = struct('rc3mode',0);
    option = makeOptionStruct(defaults, varargin);
    subplot(211)
    distfilt = ~cellfun('isempty',{galaxies.distance});
    zfilt = ~cellfun('isempty',{galaxies.z});
    if any(distfilt)
        histogram([galaxies.distance])
        xlabel('Galaxy distance [Mpc]')
    elseif any(zfilt)
        histogram([galaxies.z])
        xlabel('Galaxy redshift z')
    end
```

```
    subplot(212)
    if option.rc3mode
        minim = min([galaxies.maja]);
        maxim = max([galaxies.maja]);
        x = linspace(log10(minim),log10(maxim),50); % change numbers to your needs
        Xexp = 10.^x; % 10 to 1 thousand
        %min([galaxies.maja])
        %max([galaxies.maja])
        histogram([galaxies.maja], Xexp);
        set(gca, 'Xscale', 'log');
    else
        histogram([galaxies.maja]);
    end

    xlabel('Galaxy major axes [arcmin]')
end

function options = makeOptionStruct(defaults, varargs)
% Helper function to make name/value pairs into a struct, for options.
    optionNames = fieldnames(defaults);
    options = defaults;

    % Needs to be a whole number of name/value pairs.
    nArgs = length(varargs);
    if mod(nArgs, 2) ~= 0
        error('Malformed name/value pairs.')
    end

    for pair = reshape(varargs,2,[]) % pair is {name; value}
        inputName = pair{1};
        if any(strcmp(inputName, optionNames))
            options.(inputName) = pair{2};
        else
            error('Unknown option: %s', inputName)
        end
    end
end
```

B.2 Code for imaging and analyzing LOFAR data

The code for the analysis in Chapter 3 was written in Python. In addition to the programs mentioned in Section 3.2, I have also included support for Faraday synthesis (Bell and Enßlin 2012) using fsclean (Bell and Ensslin 2015).

Imaging program (rmsynth.py)

```
import time
import os
import shutil
import sys
import subprocess
import inspect
import math

import numpy
from astropy.io import fits
import pyregion

import statusfile
from h5_to_fits import h5_to_fits



# The list of visibility files:
MS_LIST = ["M51.BLOCKA.MS.final",
           "M51.BLOCKB.MS.final",
           "M51.BLOCKC.MS.final",
           "M51.BLOCKD.MS.final",
           "M51.BLOCKE.MS.final",
           "M51.BLOCKF.MS.final",
           "M51.BLOCKG.MS.final",
           "M51.BLOCKH.MS.final"]
# Where these files are located:
MS_DIR = "/mnt/bure_1/anton"

# Set this to True to use the "data" column
# in the MS instead of "corrected data":
USE_CORRECTED_DATA = True

# Removes the averaged visibilities when they are no longer needed.
# This greatly reduces disk usage, but increases the time consumption
# if you want to redo the wsclean step.
REMOVE_AVG_DATA = False
```

```
# The default directory and file names.
# There is probably no need to change these.
AVG_DIR = "averaged_data"
WSCLEAN_NAME = "wscleaned"
FSCLEAN_NAME = "fscleaned"
RMSYNTH_DIR = "RM_synthesis"
FSCLEAN_DIR = "Faraday_synthesis"
FSCLEAN_PARAMS = "fsclean.parset"
FILTER_NAME = "rm_filter.reg"
PYRMSYNTH_PARAMS = "params.par"
PYRMSYNTH_NAME = "out"
PHASE_AVG_LOG = "phase_avg.log"
FSCLEAN_LOG = "fsclean.log"
WSCLEAN_LOG = "wsclean.log"
PYRMSYNTH_LOG = "pyrmsynth.log"


def main(directory, phase_avg_arguments=None, wsclean_arguments=None,
         pyrmsynth_arguments=None, fsclean_arguments=None,
         noreplace=True, include_fsclean=False, redo=None,
         last_step=None, standalone_mode=False):
    """Create images and Faraday cubes of a source.

    This program will image a location in Faraday space, using the visibilities
    given by MS_LIST and MS_DIR. By default fsclean is not done, but support for
    it is included (as long as the visibilities are in a single MeasurementSet).
    The steps are (in order):

    Phase shifting and averaging ("avg")
    Faraday synthesis if included ("fsclean")
    Creating I, Q, and U images ("wsclean")
    Creating Faraday cubes ("pyrmsynth")

    If the program stops before finishing, it can be resumed from the latest
    completed step by running it again using the same directory.

    The completed steps, along with their given parameters, are written to
    "status.txt".

    For documentation on the parameters for the individual steps, see the
    methods 'do_phase_shift_and_average', 'do_wsclean', 'do_pyrmsynth'
    and 'do_fsclean'.

    NOTE: During the wsclean step many files will be opened simultaneously,
    so make sure your system allows it. (e.g. by running ulimit -n 10000)
```

```
Args:
    directory: The name of the directory in which the data of the source
        will be stored. If the directory already exists and contains
        products of this program, it continues where it left off.
    phase_avg_arguments: A dictionary of arguments to
        send to the 'do_phase_shift_and_average' method.
    wsclean_arguments: A dictionary of arguments to
        send to the 'do_wsclean' method.
    pyrmsynth_arguments: A dictionary of arguments to
        send to the 'do_pyrmsynth' method.
    fsclean_arguments: A dictionary of arguments to
        send to the 'do_fsclean' method.
    noreplace: If the program is not run from the command line, the default
        behaviour is to abort if files from previous runs are found. Change
        this argument to False to instead replace them.
    include_fsclean: This will include fsclean in the process. Note that
        the program will currently halt while fsclean is being run, and this
        may take a very long time. Note that the program will still run
        wsclean and pyrmsynth. If you only want fsclean, you can set
        'last_step' to "fsclean" in combination with this option.
    redo: Redo this step and those after (unless prevented by 'last_step').
        Valid values are None, "avg", "wsclean", "pyrmsynth" and "fsclean".
        If "fsclean" is chosen, it will not mark the later steps for redoing
        as they do not depend on fsclean.
    last_step: Skip all steps after this one. Valid values are None, "avg",
        "fsclean" and "wsclean" ("pyrmsynth" is already the last step).
    standalone_mode: Determines whether to ask for user input (when removing
        files) and print output to the screen. If the program is run from
        the command line this will be set to True. Note that a progress
        indicator from the averaging will be printed to the screen anyway.
"""
if phase_avg_arguments == None:
    phase_avg_arguments = dict()
if wsclean_arguments == None:
    wsclean_arguments = dict()
if pyrmsynth_arguments == None:
    pyrmsynth_arguments = dict()
if fsclean_arguments == None:
    fsclean_arguments = dict()

# Create and/or enter the directory
if not os.path.isdir("./" + directory):
    os.mkdir("./" + directory)
os.chdir("./" + directory)
```

```
# Redirect all output to logs if not run directly through command line.
if not standalone_mode:
    sys.stdout = open("output.log", "a", buffering=0)
    sys.stderr = open("errors.log", "a", buffering=0)

# Create a status file (if none exists).
statusfile.initialize()

# Allow for redoing averaging without writing the coordinates again.
if "ra" in phase_avg_arguments and phase_avg_arguments["ra"] == "same":
    phase_avg_arguments["ra"] = statusfile.read_shift_and_avg("ra")
if "dec" in phase_avg_arguments and phase_avg_arguments["dec"] == "same":
    phase_avg_arguments["dec"] = statusfile.read_shift_and_avg("dec")

#-------------------------------------------------------------------#
#------------------Decide which steps to do-------------------------#
#-------------------------------------------------------------------#

will_shift_and_avg = False
will_wsclean = False
will_pyrmsynth = False
will_fsclean = False
if not statusfile.check_shift_and_avg() or redo == "avg":
    will_shift_and_avg = True
    will_wsclean = True
    will_pyrmsynth = True
if include_fsclean and (not statusfile.check_fsclean()
                        or redo == "fsclean"):
    will_fsclean = True
if not statusfile.check_wsclean() or redo == "wsclean":
    will_wsclean = True
    will_pyrmsynth = True
if not statusfile.check_pyrmsynth() or redo == "pyrmsynth":
    will_pyrmsynth = True
#Stop early?
if last_step == "avg":
    will_fsclean = False
    will_wsclean = False
    will_pyrmsynth = False
elif last_step == "fsclean":
    will_wsclean = False
    will_pyrmsynth = False
elif last_step == "wsclean":
    will_pyrmsynth = False
```

```
#-------------------------------------------------------------#
#-----------Remove old files and reset statusfile-------------#
#-------------------------------------------------------------#
iqu_dirs = ["stokes_i", "stokes_q", "stokes_u", "MFS"]
# Determines which files to remove, making sure they exist.
remove_vis = os.path.isdir("./" + AVG_DIR) and will_shift_and_avg

remove_iqu = ( any([d in os.listdir(".") for d in iqu_dirs])
            and (will_wsclean or will_shift_and_avg) )

remove_rmsynthed = ( os.path.isdir("./" + RMSYNTH_DIR)
              and (will_pyrmsynth or will_wsclean or will_shift_and_avg) )

remove_fscleaned = ( os.path.isdir("./" + FSCLEAN_DIR)
                    and (will_fsclean or will_shift_and_avg) )
# Ask for confirmation before removing anything.
if remove_vis or remove_iqu or remove_rmsynthed or remove_fscleaned:
    if standalone_mode:
        print("This will remove:")
        if remove_vis:
            print("*the averaged visibility data")
        if remove_iqu:
            print("*the I, Q and U images")
        if remove_rmsynthed:
            print("*the results of RM synthesis")
        if remove_fscleaned:
            print("*the results of Faraday synthesis")
        answer = raw_input("Are you sure about this? [y/n]:")
        if answer != "y":
            sys.exit("Exiting.")
    elif noreplace:
        sys.exit("Files exist. Exiting.")

# Remove the files
if remove_vis:
    if os.path.isdir("./" + AVG_DIR):
        shutil.rmtree("./" + AVG_DIR)
    if os.path.isfile(PHASE_AVG_LOG):
        os.remove(PHASE_AVG_LOG)
    statusfile.clear_shift_and_avg()
```

```
if remove_iqu:
    for d in iqu_dirs:
        if os.path.isdir("./" + d):
            shutil.rmtree("./" + d)
    if os.path.isfile(WSCLEAN_LOG):
        os.remove(WSCLEAN_LOG)
    statusfile.clear_wsclean()
if remove_rmsynthed:
    if os.path.isdir("./" + RMSYNTH_DIR):
        shutil.rmtree("./" + RMSYNTH_DIR)
    statusfile.clear_pyrmsynth()
if remove_fscleaned:
    if os.path.isdir("./" + FSCLEAN_DIR):
        shutil.rmtree("./" + FSCLEAN_DIR)
    if os.path.isfile(FSCLEAN_LOG):
        os.remove(FSCLEAN_LOG)
    if os.path.isfile("./" + FSCLEAN_PARAMS):
        os.remove("./" + FSCLEAN_PARAMS)
    statusfile.clear_fsclean()

#-------------------------------------------------------------------#
#--------------------The real action begins-------------------------#
#-------------------------------------------------------------------#

# Phase shift and average
if will_shift_and_avg:
    channelsout = do_phaseshift_and_average(**phase_avg_arguments)
    ra = phase_avg_arguments["ra"]
    dec = phase_avg_arguments["dec"]
else:
    print("Skipping phase shifting and averaging.")
    if will_wsclean or will_fsclean:
        print("Reading number of channels and coords from status file.")
        channelsout = statusfile.read_shift_and_avg("number of channels")
        channelsout = int(channelsout)
        ra = statusfile.read_shift_and_avg("ra")
        dec = statusfile.read_shift_and_avg("dec")

# Run Faraday synthesis
if will_fsclean:
    fsclean_arguments["channels"] = channelsout
    fsclean_arguments["radec"] = ra + " " + dec
    do_fsclean(**fsclean_arguments)
else:
    print("Skipping fsclean")
```

```
    # Make I,Q,U images
    if will_wsclean:
        wsclean_arguments["channels"] = channelsout
        do_wsclean(**wsclean_arguments)
    else:
        print("Skipping wsclean.")

    # Remove averaged visibilites if no longer needed.
    if REMOVE_AVG_DATA and (will_wsclean or will_fsclean):
        if os.path.isdir("./" + AVG_DIR):
            shutil.rmtree("./" + AVG_DIR)

    # Do RM synthesis
    if will_pyrmsynth:
        do_pyrmsynth(**pyrmsynth_arguments)
    else:
        print("Skipping pyrmsynth.")

    # Done!

#-------------------------------------------------------------------#
#--------------------Phase shift and average------------------------#
#-------------------------------------------------------------------#
def do_phaseshift_and_average(ra=None, dec=None, avg_freq=2, avg_time=170):
    """Use NDPPP to phase shift and average the visibilities.

    Args:
        ra: Right ascension. None means do not phase shift.
        dec: Declination. None means do not phase shift.
        avg_freq: Average in frequency by this factor.
        avg_time: Average in time by this factor.
    """

    time_before = time.time()

    steplist = [] # NDPPP steps to do
    parlist = [] # NDPPP parameters

    if USE_CORRECTED_DATA:
        parlist.append("msin.datacolumn=CORRECTED_DATA")
    else:
        parlist.append("msin.datacolumn=DATA")
    parlist.append("msout.datacolumn=DATA")
```

```
# Add phase shifting parameters
if ra == None or dec == None:
    print("Not phase shifting.")
else:
    print("Phase shifting: " + ra + " " + dec)
    steplist.append("phaseshifter")
    parlist.append("phaseshifter.phasecenter=[" + ra + "," + dec + "]")
# Add averaging parameters
if avg_freq == 1 and avg_time == 1:
    print("Not averaging")
else:
    print("Averaging: " + str(avg_time) + " timesteps, "
          + str(avg_freq) + " channels.")
    steplist.append("average")
    parlist.append("average.timestep=" + str(avg_time))
    parlist.append("average.freqstep=" + str(avg_freq))
# Format the steps like NDPPP wants them.
steps_string = "steps=["
for step in steplist:
    steps_string = steps_string + step + ","
steps_string = steps_string[:-1] + "]" # [:-1] is to remove the last comma.

os.mkdir("./" + AVG_DIR)
#And here the magic happens.
for ms in MS_LIST:
    ndppp_command = ("NDPPP msin=" + MS_DIR + "/" + ms + " msout="
                     + AVG_DIR + "/" + ms + " " + steps_string)
    for par in parlist:
        ndppp_command = ndppp_command + " " + par
    print(ndppp_command)
    # Run NDPPP and send its output to the log
    subprocess.check_call(ndppp_command + " >> " + PHASE_AVG_LOG,
                          shell=True)

# Use the log to get the number of channels. This works for the dataset I
# have, and probably for any dataset that is either a single MS or several
# MS split by frequency. But I'm not proud of it.
nchanlist = []
with open(PHASE_AVG_LOG, "r") as logfile:
    for line in logfile:
        if "nchan" in line and not "(0)" in line:
            param, value = line.split(":",1)
            nchans = int(value.strip())
            nchanlist.append(nchans)
    channelsout = sum(nchanlist)
```

```
    time_after = time.time()
    statusfile.write_shift_and_avg(
                    ra=ra, dec=dec, avg_freq=avg_freq, avg_time=avg_time,
                    channelsout=channelsout, command=ndppp_command,
                    time=time_after - time_before)
    #Finally, return the number of channels to pass to wsclean
    return channelsout

#------------------------------------------------------------------#
#-----------------------------wsclean------------------------------#
#------------------------------------------------------------------#
def do_wsclean(channels, wsclean_avg=1, imagesize=256, pixelsize=1,
               taper=None, weighting="briggs 0", niter=25000, mgain=0.85,
               cutoff=0.0003, beamshape=None, samebeam=True, joinchannels=True,
               lofar_correction=True):
    """Use wsclean to create images of each channel, as well as MFS images.

    Args:
        channels: Number of channels in the data.
        wsclean_avg: Additional averaging in frequency in this step. This can
            be useful if you have saved the averaged data and want to increase
            averaging.
        imagesize: The length of the sides of the field, in pixels.
        pixelsize: The size of each pixel, in arcseconds.
        taper: The target beamsize (in arcsec) of the Gaussian taper applied to
            the weights. If None, no taper is applied.
            More information: https://sourceforge.net/p/wsclean/wiki/Tapering
        weighting, niter, mgain, cutoff: These are passed directly to wsclean,
            cutoff being the threshold.
        beamshape: If not None, gives a manual shape for the restoring beam.
            Should be a tuple of (bmaj, bmin, pa), with bmaj and bmin in arcsec
            and pa in degrees.
        samebeam: If True (and no beamshape is given), make the psf of the
            longest wavelength and get the restoring beam for all channels
            from that.
        joinchannels: Clean the channels jointly. More information:
            https://sourceforge.net/p/wsclean/wiki/WidebandDeconvolution
        lofar_correction: Uses the LOFAR primary beam correction in wsclean.
    """
    time_before = time.time()
    channels = int(channels/wsclean_avg)
    if not beamshape == None:
        samebeam = False
    inputstr = ""
    for ms in MS_LIST:
        inputstr += " ./" + AVG_DIR + "/" + ms
```

```
if lofar_correction:
    pols = "iquv"
    lofarstr = "--apply-primary-beam "
else:
    pols = "iqu"
    lofarstr = ""
if joinchannels:
    joinstr = "-joinchannels "
else:
    joinstr = ""
if taper == None:
    taperstr = ""
else:
    taperstr = "-taper-gaussian " + str(taper) + "asec "
if beamshape == None:
    if samebeam:
        # Make PSF of the longest wavelength and use that beam size.
        wsclean_psf_command = ("wsclean -v -make-psf-only -size "
                               + str(imagesize) + " " + str(imagesize)
                               + " -scale " + str(pixelsize) + "asec -weight "
                               + weighting + " -channelrange 0 1 " + taperstr
                               + joinstr + "-name beamfit" + inputstr)
        print("Getting psf:")
        print(wsclean_psf_command)
        p = subprocess.check_call(wsclean_psf_command + " >> "
                                  + WSCLEAN_LOG, shell=True)
        hdr = fits.getheader("beamfit-psf.fits")
        bmaj = hdr["bmaj"]*3600
        bmin = hdr["bmin"]*3600
        pa = hdr["bpa"]
        beamshape = (bmaj, bmin, pa)
        bs_str = (" -beamshape " + str(bmaj) + " "
                  + str(bmin) + " " + str(pa))
    else:
        bs_str = ""
else:
    bmaj, bmin, pa = beamshape
    bs_str = " -beamshape " + str(bmaj) + " " + str(bmin) + " " + str(pa)

os.mkdir("./MFS")
```

```
# Make the MFS images with beam correction,
# as wsclean 1.11 does not do it when imaging multiple channels
if lofar_correction:
    wsclean_mfs_command = ("wsclean " + lofarstr + "-size " + str(imagesize)
                           + " "+str(imagesize) + " -scale "
                           + str(pixelsize) + "asec -weight " + weighting
                           + " -pol " + pols + " -niter " + str(niter)
                           + " -mgain " + str(mgain)
                           + " -no-update-model-required " + taperstr
                           + bs_str + " -threshold " + str(cutoff)
                           + " -name " + WSCLEAN_NAME + inputstr)
    p = subprocess.check_call(wsclean_mfs_command + " >> " + WSCLEAN_LOG,
                              shell=True)

    # Move the beam corrected MFS output files into
    # their directory and remove the rest
    for pol in ["I", "Q", "U"]:
        shutil.move("./" + WSCLEAN_NAME + "-" + pol + "-image-pb.fits",
            "./MFS/" + WSCLEAN_NAME + "-MFS-" + pol + "-image-pb.fits")
    os.remove("./" + WSCLEAN_NAME + "-V-image-pb.fits")
    for pol in ["I", "Q", "U", "V"]:
        os.remove("./" + WSCLEAN_NAME + "-" + pol + "-dirty.fits")
        os.remove("./" + WSCLEAN_NAME + "-" + pol + "-first-residual.fits")
        os.remove("./" + WSCLEAN_NAME + "-" + pol + "-image.fits")
        os.remove("./" + WSCLEAN_NAME + "-" + pol + "-model.fits")
        os.remove("./" + WSCLEAN_NAME + "-" + pol + "-residual.fits")
    for jonescomp in ["XX", "XY", "YX", "YY"]:
        os.remove("./" + WSCLEAN_NAME + "-beam-" + jonescomp + ".fits")
        os.remove("./" + WSCLEAN_NAME + "-beam-" + jonescomp + "i.fits")
    os.remove("./" + WSCLEAN_NAME + "-psf.fits")
# Make the channel images
wsclean_command = ("wsclean " + lofarstr + "-size " + str(imagesize) + " "
                   + str(imagesize) + " -scale " + str(pixelsize)
                   + "asec -weight " + weighting + " -pol " + pols
                   + " -niter " + str(niter) + " -mgain " + str(mgain)
                   + " -no-update-model-required" + " -channelsout "
                   + str(channels) + " " + joinstr + taperstr + bs_str
                   + " -threshold " + str(cutoff) + " -name "
                   + WSCLEAN_NAME + inputstr)
print("Running wsclean:")
print(wsclean_command)
print("(This will probably take a while.)")
p = subprocess.check_call(wsclean_command + " >> " + WSCLEAN_LOG,
                          shell=True)
```

```
os.mkdir("./stokes_i")
os.mkdir("./stokes_q")
os.mkdir("./stokes_u")
for i in range(0, channels):
    # Move image files to the appropriate directories
    if channels == 1:
        firstpart = WSCLEAN_NAME
    else:
        firstpart = WSCLEAN_NAME + "-" + str(i).zfill(4)
    for pol in ["I", "Q", "U"]:
        if lofar_correction:
            filename = firstpart + "-" + pol + "-image-pb.fits"
        else:
            filename = firstpart + "-" + pol + "-image.fits"
        shutil.move("./" + filename,
                    "./stokes_" + pol.lower() + "/" + filename)
    # Remove everything else
    os.remove("./" + firstpart + "-psf.fits")
    for pol in ["I", "Q", "U"]:
        os.remove("./" + firstpart + "-" + pol + "-dirty.fits")
        os.remove("./" + firstpart + "-" + pol + "-first-residual.fits")
        os.remove("./" + firstpart + "-" + pol + "-model.fits")
        os.remove("./" + firstpart + "-" + pol + "-residual.fits")
        if lofar_correction:
            os.remove("./" + firstpart + "-" + pol + "-image.fits")
    if lofar_correction:
        os.remove("./" + firstpart + "-V-dirty.fits")
        os.remove("./" + firstpart + "-V-first-residual.fits")
        os.remove("./" + firstpart + "-V-model.fits")
        os.remove("./" + firstpart + "-V-residual.fits")
        os.remove("./" + firstpart + "-V-image.fits")
        os.remove("./" + firstpart + "-V-image-pb.fits")
        for jonescomp in ["XX","XY","YX","YY"]:
            os.remove("./" + firstpart + "-beam-" + jonescomp + ".fits")
            os.remove("./" + firstpart + "-beam-" + jonescomp + "i.fits")
if channels != 1:
    # Move the averaged images
    firstpart = WSCLEAN_NAME + "-MFS"
    for pol in ["I", "Q", "U"]:
        shutil.move("./" + firstpart + "-" + pol + "-image.fits",
                    "./MFS/" + firstpart + "-" + pol + "-image.fits")
```

```
        # Also remove the other averaged files
        os.remove("./" + firstpart + "-psf.fits")
        for pol in ["I", "Q", "U"]:
            os.remove("./" + firstpart + "-" + pol + "-dirty.fits")
            os.remove("./" + firstpart + "-" + pol + "-model.fits")
            os.remove("./" + firstpart + "-" + pol + "-residual.fits")
        if lofar_correction:
            os.remove("./" + firstpart + "-V-dirty.fits")
            os.remove("./" + firstpart + "-V-model.fits")
            os.remove("./" + firstpart + "-V-residual.fits")
            os.remove("./" + firstpart + "-V-image.fits")


    time_after = time.time()
    statusfile.write_wsclean(imagesize=imagesize, pixelsize=pixelsize,
                             weighting=weighting, niter=niter, mgain=mgain,
                             cutoff=cutoff, channelsout=channels, taper=taper,
                             beamshape=beamshape, samebeam=samebeam,
                             command=wsclean_command,
                             time=time_after - time_before)


#------------------------------------------------------------------#
#-------------------------pyrmsynth--------------------------------#
#------------------------------------------------------------------#
def do_pyrmsynth(min_phi=-100, max_phi=100, phi_res=0.25,
                 niter=5000, cutoff=0.0003, specind=None):
    """Uses pyrmsynth to do RM synthesis on the Q and U images.

    If a ds9 region file with the name FILTER_NAME or a FITS file with the name
    'mask.fits' exists in the directory of the source it will be used as a
    mask for RM synthesis.

    If a file with the name 'spectral-index.fits' exists the spectral index of
    each line of sight will be taken from the file.

    Args:
        min_phi: The minimum Faraday depth.
        max_phi: The maximum Faraday depth.
        phi_res: The distance between adjacent sampled points in Faraday space.
        niter: Maximum number of iterations in RMCLEAN.
        cutoff: The maximum value of the residual whel RMCLEAN will stop.
        specind: The spectral index, if no spectral index file exists.
            If None, none is supplied to pyrmsynth.
    """
```

```
time_before = time.time()
if os.path.isfile("./" + FILTER_NAME):
    use_mask = True
    # Make the ds9 region file into a filter
    # First, open an image file and massacre it to make pyregion happy
    fitsfile = fits.open("./MFS/" + WSCLEAN_NAME + "-MFS-I-image.fits")
    hdr = fitsfile[0].header
    for nb in [3, 4]:
        for word in ["ctype", "crpix", "crval", "cdelt", "cunit", "naxis"]:
            del hdr[word + str(nb)]
    hdr["naxis"] = 2
    # Create the mask
    image_size = hdr["naxis1"]
    region = pyregion.open("./" + FILTER_NAME)
    filt = region.as_imagecoord(hdr).get_filter()
    mask = filt.mask((image_size, image_size))
    int_mask = mask.astype(int)
    hdu = fits.PrimaryHDU(int_mask)
    if os.path.isfile("./mask.fits"):
        os.remove("./mask.fits")
    hdu.writeto("mask.fits")
elif os.path.isfile("./mask.fits"):
    use_mask = True
else:
    use_mask = False
# Write a parameter file using default_params.par as base
dirname = os.path.dirname(os.path.abspath(inspect.stack()[0][1]))
default_params = open(dirname + "/default_params.par", "r")
new_params = open("./" + PYRMSYNTH_PARAMS, "w")
for line in default_params:
    if line[0:7] == "phi_min":
        new_params.write("phi_min " + str(min_phi) + "\n")
    elif line[0:4] == "nphi":
        new_params.write("nphi " + str(int((max_phi - min_phi)/phi_res))
                         + "\n")
    elif line[0:4] == "dphi":
        new_params.write("dphi " + str(phi_res) + "\n")
    elif line[0:8] == "outputfn":
        new_params.write("outputfn " + "./" + RMSYNTH_DIR + "/"
                         + PYRMSYNTH_NAME + "\n")
    elif line[0:5] == "niter":
        new_params.write("niter " + str(niter) + "\n")
    elif line[0:6]=="cutoff":
        new_params.write("cutoff " + str(cutoff) + "\n")
```

```
        elif line[0:10] == "%imagemask":
            if use_mask:
                new_params.write("imagemask mask.fits \n")
            else:
                new_params.write(line)
        elif line[0:7] == "% alpha":
            if os.path.isfile("spectral-index.fits"):
                new_params.write("alpha spectral-index.fits \n")
            elif specind != None:
                new_params.write("alpha " + str(specind) + " \n")
            else:
                new_params.write(line)
        else:
            new_params.write(line)
    default_params.close()
    new_params.close()
    os.mkdir("./" + RMSYNTH_DIR)
    # I use a slightly modified pyrmsynth due to a bug with
    # creating the 2D maps (on my system at least).
    pyrmsynth_command = ("python " + dirname
                         + "/pyrmsynth_modified/rmsynthesis.py -s "
                         + "./" + PYRMSYNTH_PARAMS)
    print(pyrmsynth_command)
    subprocess.check_call(pyrmsynth_command + " > " + PYRMSYNTH_LOG, shell=True)
    time_after = time.time()
    statusfile.write_pyrmsynth(min_phi=min_phi, max_phi=max_phi,
                               phi_res=phi_res,specind=specind,
                               command=pyrmsynth_command, niter=niter,
                               time=time_after - time_before)


#-------------------------------------------------------------------------#
#-----------------------------fsclean-------------------------------------#
#-------------------------------------------------------------------------#
def do_fsclean(channels, size=256, scale=1, max_phi=100, phi_res=0.25,
               beamshape=None, default_bphi=1, niter=1000, gain=0.1, cutoff=0,
               hogbom_clean=True, radec=None):
    """Uses fsclean to run Faraday synthesis on the visibilities.

    This method creates Faraday cubes in both hdf5 and fits format.

    Unless an explicit beamshape is supplied, wsclean will be used to provide
    bmaj, bmin and pa for the shortest wavelength.

    NOTE: default_bphi will be set to 1 rad m^-2 by default, which works for
    the dataset I have. You should calculate your theoretical resolution.
```

```
NOTE: For this to work the visibilities need to be contained in a single
MeasurementSet.

Args:
    channels: The number of channels in the data.
    size: The length of the sides of the field, in pixels.
    scale: The size of each pixel, in arcseconds.
    max_phi: The maximum Faraday depth. The minimum is equal to -max_phi.
    phi_res: The distance between adjacent sampled points in Faraday space.
    beamshape: The shape of the restoring beam,
        as a tuple (bmaj, bmin, pa, bphi). If None, use wsclean and
        default_bphi to get the beam.
    default_bphi: The value of bphi when getting the beamshape from wsclean.
    niter, gain, cutoff: These are passed directly to fsclean.
    hogbom_clean: If True, use Hogbom CLEAN, else use Clark CLEAN.
    radec: A string on the form "RA DEC" with the position of the source.
        This is to give the fits files the correct coordinates, as the
        position does not seem to be saved in the hdf5 files.

"""
time_before = time.time()
if len(MS_LIST) > 1:
    print("WARNING! Only one MS can be used for fsclean, skipping fsclean.")
    return
infilename = "./" + AVG_DIR + "/" + MS_LIST[0]

#Write a parameter file for fsclean.
with open("./" + FSCLEAN_PARAMS,"w") as params:
    params.write("nra = " + str(size) + "\n")
    params.write("ndec = " + str(size) + "\n")
    params.write("cellsize = " + str(scale) + "\n")
    nphi = int(2 * max_phi / phi_res)
    params.write("nphi = " + str(nphi) + "\n")
    params.write("dphi = " + str(phi_res) + "\n")
    params.write("niter = " + str(niter) + "\n")
    params.write("gain = " + str(gain) + "\n")
    params.write("cutoff = " + str(cutoff) + "\n")
    if beamshape == None:
        # As fsclean beam fitting doesn't work as of this code, use
        # wsclean to get the beam size.
        wsclean_psf_command = ("wsclean -v -make-psf-only -size "
                               + str(size) + " " + str(size) + " -scale "
                               + str(scale) + "asec -channelrange "
                               + str(channels - 2) + " " + str(channels - 1)
                               + " -weight natural "
                               + "-name beamfit-fsclean " + infilename)
```

```
        print("Getting psf:")
        print(wsclean_psf_command)
        p = subprocess.check_call(wsclean_psf_command + " >> "
                                  + FSCLEAN_LOG, shell=True)
        hdr = fits.getheader("beamfit_fsclean_psf.fits")
        bmaj = hdr["bmaj"]*3600
        bmin = hdr["bmin"]*3600
        pa = hdr["bpa"]
        beamshape = (bmaj, bmin, pa, default_bphi)
        fitbeam = True
    else:
        fitbeam = False
    bmaj = beamshape[0]
    bmin = beamshape[1]
    bpa = beamshape[2]
    bphi = beamshape[3]
    params.write("bmaj = " + str(bmaj) + "\n")
    params.write("bmin = " + str(bmin) + "\n")
    params.write("bpa = " + str(bpa) + "\n")
    params.write("bphi = " + str(bphi) + "\n")
    params.write("verbosity = 1\n")
    if hogbom_clean:
        params.write("clean_type = 1\n")
    else:
        params.write("clean_type = 0\n")
outfilename = FSCLEAN_DIR + "/" + FSCLEAN_NAME
os.mkdir("./" + FSCLEAN_DIR)
# Assume fsclean is in the same directory as this script
script_dir = os.path.dirname(os.path.abspath(inspect.stack()[0][1]))
fsclean_command = ("python " + script_dir + "/fsclean/fsclean.py ./"
                   + FSCLEAN_PARAMS + " " + infilename + " " + outfilename)
print(fsclean_command)
subprocess.check_call(fsclean_command, shell=True)
# Append the fsclean logfile to the one I use
with open(FSCLEAN_DIR + "/" + FSCLEAN_NAME + ".log") as inlog:
    with open(FSCLEAN_LOG,"a") as outlog:
        outlog.write(inlog.read())
os.remove(FSCLEAN_DIR + "/" + FSCLEAN_NAME + ".log")
# The residual should also be in the Faraday synthesis directory
shutil.move(FSCLEAN_NAME + "_resim.hdf5",
            FSCLEAN_DIR + "/" + FSCLEAN_NAME + "_resim.hdf5")
# Convert to FITS
for fn in os.listdir(FSCLEAN_DIR):
    fn = FSCLEAN_DIR + "/" + fn
    if fn[-4:] == "hdf5":
        h5_to_fits(fn, fn[:-4] + "fits", beamshape=beamshape, radec=radec)
```

```
    time_after = time.time()
    statusfile.write_fsclean(max_phi=max_phi, phi_res=phi_res, gain=gain,
                             cutoff=cutoff, hogbom_clean=hogbom_clean,
                             size=size, scale=scale, command=fsclean_command,
                             niter=niter, fitbeam=fitbeam, beamshape=beamshape,
                             time=time_after - time_before)

#-------------------------------------------------------------------#
#----------------------From command line----------------------------#
#-------------------------------------------------------------------#
if __name__ == "__main__":
    # Create the list of command line arguments
    cl_arguments = sys.argv[1:]
    if len(cl_arguments) == 0 or "-h" in cl_arguments:
        helpstr = """
      Command line arguments:
      -coords <ra> <dec> (ex. -coords 13h39m23s +46d40m05s)
      -freqavg <number of channels> (Default: 2)
      -timeavg <number of timesteps> (Default: 170)
      -size <image size> (Default: 256)
      -scale <pixel size> (Default: 1)
      -niter <number of CLEAN iterations> (Default: 25000)
      -cutoff <CLEAN cutoff> (Default: 0.0003)
      -taper <beam size> (Gaussian taper, beam size in arcsec.)
      -nojoinchannels (Do not give the joinchannels parameter to wsclean)
      -no-correct-beam (Don't do LOFAR primary beam correction in wsclean)
      -wscleanavg <number of channels> (Averaging in wsclean step. Default: 1)
      -beamshape <bmaj> <bmin> <pa> (Set the restoring beam for wsclean)
      -no-fit-same-beam (Let wsclean fit the beam for each channel)
      -rmiter <number of iterations> (RM clean iterations. Default: 5000)
      -rmcutoff <cutoff> (RM clean cutoff. Default: 0.0001)
      -maxphi <phi> (RM clean and fsclean maximum absolute Faraday depth)
      -phi-sampling <dphi> (RM clean and fsclean sampling in Faraday depth)
      -specind <alpha> (Set a spectral index for pyrmsynth)
      -fsiter <number of iterations> (fsclean iterations. Default: 1000)
      -fscutoff <cutoff> (fsclean cutoff. Default: 0)
      -bphi <fwhm> (Restoring beam FWHM in Faraday depth when using fsclean)
      -fsclean-only (do fsclean instead of wsclean+pyrmsynth)
      -fsclean-too (do fsclean as well as wsclean+pyrmsynth)
      -redo <part> (avg, mscorpol, wsclean, pyrmsynth, Redo from this step.)
      -last <part> (avg, mscorpol, wsclean. Stop after this step.)
      """
        sys.exit(helpstr)
```

```
# Create the dicts of input arguments
general_arguments = dict()
phase_avg_arguments = dict()
wsclean_arguments = dict()
pyrmsynth_arguments = dict()
fsclean_arguments = dict()
# Parse the command line arguments
if "--coords" in cl_arguments:
    ind = cl_arguments.index("--coords")
    phase_avg_arguments["ra"] = cl_arguments.pop(ind + 1)
    if not phase_avg_arguments["ra"] == "same":
        phase_avg_arguments["dec"] = cl_arguments.pop(ind + 1)
    else:
        phase_avg_arguments["dec"] = "same"
    del cl_arguments[ind]
if "--freqavg" in cl_arguments:
    ind = cl_arguments.index("--freqavg")
    phase_avg_arguments["avg_freq"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--timeavg" in cl_arguments:
    ind = cl_arguments.index("--timeavg")
    phase_avg_arguments["avg_time"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--size" in cl_arguments:
    ind = cl_arguments.index("--size")
    size = int(cl_arguments.pop(ind + 1))
    wsclean_arguments["imagesize"] = size
    fsclean_arguments["size"] = size
    del cl_arguments[ind]
if "--scale" in cl_arguments:
    ind = cl_arguments.index("--scale")
    scale = float(cl_arguments.pop(ind + 1))
    wsclean_arguments["pixelsize"] = scale
    fsclean_arguments["scale"] = scale
    del cl_arguments[ind]
if "--niter" in cl_arguments:
    ind = cl_arguments.index("--niter")
    wsclean_arguments["niter"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--cutoff" in cl_arguments:
    ind = cl_arguments.index("--cutoff")
    wsclean_arguments["cutoff"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
```

```
if "--taper" in cl_arguments:
    ind = cl_arguments.index("--taper")
    wsclean_arguments["taper"] = float(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--nojoinchannels" in cl_arguments:
    ind = cl_arguments.index("--nojoinchannels")
    wsclean_arguments["joinchannels"] = False
    del cl_arguments[ind]
if "--no-correct-beam" in cl_arguments:
    ind = cl_arguments.index("--no-correct-beam")
    wsclean_arguments["lofar_correction"] = False
    del cl_arguments[ind]
if "--wscleanavg" in cl_arguments:
    ind = cl_arguments.index("--wscleanavg")
    wsclean_arguments["wsclean_avg"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--beamshape" in cl_arguments:
    ind = cl_arguments.index("--beamshape")
    bmaj = float(cl_arguments.pop(ind + 1))
    bmin = float(cl_arguments.pop(ind + 1))
    pa = float(cl_arguments.pop(ind + 1))
    wsclean_arguments["beamshape"] = (bmaj, bmin, pa)
    del cl_arguments[ind]
if "--no-fit-same-beam" in cl_arguments:
    ind = cl_arguments.index("--no-fit-same-beam")
    wsclean_arguments["samebeam"] = False
    del cl_arguments[ind]
if "--rmiter" in cl_arguments:
    ind = cl_arguments.index("--rmiter")
    pyrmsynth_arguments["niter"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--rmcutoff" in cl_arguments:
    ind = cl_arguments.index("--rmcutoff")
    pyrmsynth_arguments["cutoff"] = float(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--maxphi" in cl_arguments:
    ind = cl_arguments.index("--maxphi")
    max_phi = float(cl_arguments.pop(ind + 1))
    pyrmsynth_arguments["min_phi"] = -max_phi
    pyrmsynth_arguments["max_phi"] = max_phi
    fsclean_arguments["max_phi"] = max_phi
    del cl_arguments[ind]
```

```
if "--phi-sampling" in cl_arguments:
    ind = cl_arguments.index("--phi-sampling")
    dphi = float(cl_arguments.pop(ind + 1))
    pyrmsynth_arguments["phi_res"] = dphi
    fsclean_arguments["phi_res"] = dphi
    del cl_arguments[ind]
if "--specind" in cl_arguments:
    ind = cl_arguments.index("--specind")
    pyrmsynth_arguments["specind"] = float(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--fsiter" in cl_arguments:
    ind = cl_arguments.index("--fsiter")
    fsclean_arguments["niter"] = int(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--fscutoff" in cl_arguments:
    ind = cl_arguments.index("--fscutoff")
    fsclean_arguments["cutoff"] = float(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--bphi" in cl_arguments:
    ind = cl_arguments.index("--bphi")
    fsclean_arguments["default_bphi"] = float(cl_arguments.pop(ind + 1))
    del cl_arguments[ind]
if "--fsclean-only" in cl_arguments:
    ind = cl_arguments.index("--fsclean-only")
    general_arguments["include_fsclean"] = True
    general_arguments["last_step"] = "fsclean"
    del cl_arguments[ind]
if "--fsclean-too" in cl_arguments:
    ind = cl_arguments.index("--fsclean-too")
    general_arguments["include_fsclean"] = True
    del cl_arguments[ind]
if "--redo" in cl_arguments:
    ind = cl_arguments.index("--redo")
    redo_what = cl_arguments.pop(ind + 1)
    if redo_what in ["avg", "wsclean", "pyrmsynth", "fsclean"]:
        general_arguments["redo"] = redo_what
    else:
        sys.exit("I don't understand what you want to redo.")
    del cl_arguments[ind]
```

```
    if "--last" in cl_arguments:
        ind = cl_arguments.index("--last")
        last_what = cl_arguments.pop(ind + 1)
        if last_what in ["avg", "wsclean", "pyrmsynth", "fsclean"]:
            general_arguments["last_step"] = last_what
        else:
            sys.exit("I don't understand when you want to stop.")
        del cl_arguments[ind]
    # As this is from the command line:
    general_arguments["standalone_mode"] = True
    # Use what remains as output directory
    if len(cl_arguments) == 1:
        directory = cl_arguments[0]
    elif len(cl_arguments) == 0:
        sys.exit("Did you remember to specify the directory?")
    else:
        sys.exit("There was at least one unrecognized argument.")
    main(directory, phase_avg_arguments, wsclean_arguments,
         pyrmsynth_arguments, fsclean_arguments, **general_arguments)
```

The status for each source (statusfile.py)

```
import os
import csv
import datetime
from collections import OrderedDict
STATUS_FILENAME = "status.txt"

def initialize(force=False):
    """Create an empty status file.

    Args:
        force: If this is true, the status file is replaced if it exists.

    """
    # Do nothing if the file already exists, unless force is applied
    if os.path.isfile(STATUS_FILENAME) and not force:
        return
    with open(STATUS_FILENAME,"w") as thefile:
        thefile.write("#This file contains the progress of rmsynth.py"
                      + " in this directory.\n")
    # These will create new instances of the steps if none are present, which
    # we want here.
    clear_shift_and_avg()
    clear_wsclean()
    clear_pyrmsynth()
    clear_fsclean()
```

```
def _read_from_step(beginning, toread):
    # Reads a value from an arbitrary step. 'beginning' should be the actual
    # text of the step in the status file, as in "Done RM synthesis". 'toread'
    # should be the name of the parameter to read, as in "maximum phi"
    with open(STATUS_FILENAME, "rb") as thefile:
        csvreader = csv.reader(thefile, delimiter=":", skipinitialspace=False)
        # Goes through the file until it finds the right step.
        for row in csvreader:
            if row[0] == beginning:
                break
        # Continues from there until it reaches the end of the step or finds the
        # parameter to read.
        for row in csvreader:
            if not row[0][0:4] == "    ":
                break
            if row[0].strip() == toread:
                thefile.close()
                #Because the time format uses colons, which I use as separators.
                return ":".join(row[1:]).strip()

def _write_step(beginning, arg_dict, time, havedone=True):
    # Writes an arbitrary step. 'beginning' should be the actual text of the
    # step in the status file, as in "Done RM synthesis". 'arg_dict' is an
    # ordered dictionary of all the parameters to write and their values.
    # 'time' is the time the step took, and 'havedone'=False is used when
    # clearing the step.
    with open(STATUS_FILENAME,"rb") as thefile:
        contents = thefile.readlines()
        thefile.seek(0)
        csvreader = csv.reader(thefile, delimiter=":", skipinitialspace=True)

        if time == None:
            timestr = ""
        else:
            timestr = str(datetime.timedelta(seconds=time))
            arg_dict["time taken"] = timestr

        have_written = False
        for ind, row in enumerate(csvreader):
            # Finds the step
            if row[0] == beginning:
                if havedone:
                    contents[ind] = beginning + ": Yes\n"
                else:
                    contents[ind] = beginning + ": No\n"
```

```
                # Delete all the current parameters of the step
                while (ind + 1 < len(contents)
                       and contents[ind + 1][0:4] == "    "):
                    del contents[ind + 1]
                # Make a list of all the new parameters
                to_add = []
                for arg in arg_dict:
                    to_add.append("    " + arg + ": "
                                  + str(arg_dict[arg]) + "\n")
                contents[ind + 1:ind + 1] = to_add
                have_written = True
        # If the step has not been found, append it at the end.
        if not have_written:
            if havedone:
                contents.append(beginning + ": Yes\n")
            else:
                contents.append(beginning + ": No\n")
            for arg in arg_dict:
                contents.append("    " + arg + ": " + str(arg_dict[arg]) + "\n")
    with open(STATUS_FILENAME, "w") as thefile:
        thefile.writelines(contents)

def _check_step(beginning):
    # Checks if an arbitrary step has been done. 'beginning' should be the
    # actual text of the step in the status file, as in "Done RM synthesis".
    thefile = open(STATUS_FILENAME, "rb")
    csvreader = csv.reader(thefile, delimiter=":", skipinitialspace=True)
    for row in csvreader:
        if row[0] == beginning:
            thefile.close()
            return row[1] == "Yes"
    return False

def write_shift_and_avg(ra="", dec="", avg_freq="", avg_time="",
                        channelsout="", command="", time=None):
    """Writes the phase shifting and averaging step to the status file."""
    beginning = "Shifted and averaged"
    arg_dict = OrderedDict()
    arg_dict["ra"] = ra
    arg_dict["dec"] = dec
    arg_dict["frequency averaging"] = avg_freq
    arg_dict["time averaging"] = avg_time
    arg_dict["number of channels"] = channelsout
    arg_dict["command"] = command
    _write_step(beginning, arg_dict, time)
```

```
def clear_shift_and_avg():
    """Clears the phase shifting and averaging step."""
    beginning = "Shifted and averaged"
    arg_dict = OrderedDict()
    arg_dict["ra"] = ""
    arg_dict["dec"] = ""
    arg_dict["frequency averaging"] = ""
    arg_dict["time averaging"] = ""
    arg_dict["number of channels"] = ""
    arg_dict["command"] = ""
    _write_step(beginning, arg_dict, time=None, havedone=False)

def check_shift_and_avg():
    """Checks if the phase shifting and averaging step has been done."""
    return _check_step("Shifted and averaged")

def read_shift_and_avg(toread):
    """Reads a parameter value from the phase shifting and averaging step."""
    return _read_from_step("Shifted and averaged", toread)

def write_wsclean(imagesize="", pixelsize="", weighting="", niter="",
                  mgain="", channelsout="", taper="", beamshape=None,
                  samebeam="", cutoff="", command="", time=None):
    """Writes the imaging step to the status file."""
    beginning = "Created I,Q,U images"
    arg_dict = OrderedDict()
    arg_dict["image size"] = imagesize
    arg_dict["pixel size"] = pixelsize
    arg_dict["weighting"] = weighting
    arg_dict["wsclean niter"] = niter
    arg_dict["wsclean mgain"] = mgain
    arg_dict["wsclean cutoff"] = cutoff
    arg_dict["channels after wsclean"] = channelsout
    arg_dict["taper"] = taper
    arg_dict["restoring beam fit to longest wavelength psf"] = samebeam
    if beamshape == None:
        beamshape = ("","","")
    arg_dict["restoring beam major axis"] = beamshape[0]
    arg_dict["restoring beam minor axis"] = beamshape[1]
    arg_dict["restoring beam position angle"] = beamshape[2]
    arg_dict["command"] = command
    _write_step(beginning, arg_dict, time)
```

```
def clear_wsclean():
    """Clears the imaging step."""
    beginning = "Created I,Q,U images"
    arg_dict = OrderedDict()
    arg_dict["image size"] = ""
    arg_dict["pixel size"]=""
    arg_dict["weighting"] = ""
    arg_dict["wsclean niter"] = ""
    arg_dict["wsclean mgain"] = ""
    arg_dict["wsclean cutoff"] = ""
    arg_dict["channels after wsclean"] = ""
    arg_dict["taper"] = ""
    arg_dict["restoring beam fit to longest wavelength psf"] = ""
    arg_dict["restoring beam major axis"] = ""
    arg_dict["restoring beam minor axis"] = ""
    arg_dict["restoring beam position angle"] = ""
    arg_dict["command"] = ""
    _write_step(beginning, arg_dict, time=None, havedone=False)

def check_wsclean():
    """Checks if the imaging step has been done."""
    return _check_step("Created I,Q,U images")

def read_wsclean(toread):
    """Reads a parameter value from the imaging step."""
    return _read_from_step("Created I,Q,U images", toread)

def write_pyrmsynth(min_phi="", max_phi="", phi_res="", niter="", cutoff="",
                    specind="", command="", time=None):
    """Writes the RM synthesis step to the status file."""
    beginning = "Done RM synthesis"
    arg_dict = OrderedDict()
    arg_dict["minimum phi"] = min_phi
    arg_dict["maximum phi"] = max_phi
    arg_dict["phi resolution"] = phi_res
    arg_dict["RMCLEAN iterations"] = niter
    arg_dict["RMCLEAN cutoff"] = cutoff
    arg_dict["spectral index"] = specind
    arg_dict["command"] = command
    _write_step(beginning, arg_dict, time)
```

```
def clear_pyrmsynth():
    """Clears the RM synthesis step."""
    beginning = "Done RM synthesis"
    arg_dict = OrderedDict()
    arg_dict["minimum phi"] = ""
    arg_dict["maximum phi"] = ""
    arg_dict["phi resolution"] = ""
    arg_dict["RMCLEAN iterations"] = ""
    arg_dict["RMCLEAN cutoff"] = ""
    arg_dict["spectral index"] = ""
    arg_dict["command"] = ""
    _write_step(beginning, arg_dict, time=None, havedone=False)

def check_pyrmsynth():
    """Checks if the RM synthesis step has been done."""
    return _check_step("Done RM synthesis")

def read_pyrmsynth(toread):
    """Reads a parameter value from the RM synthesis step."""
    return _read_from_step("Done RM synthesis",toread)

def write_fsclean(max_phi="", phi_res="", gain="", cutoff="", hogbom_clean="",
                  size="", niter="", scale="", fitbeam="", beamshape=None,
                  command="", time=None):
    """Writes the Faraday synthesis step to the status file."""
    beginning = "Done fsclean"
    arg_dict = OrderedDict()
    arg_dict["image size"] = size
    arg_dict["pixel size"] = scale
    if max_phi == "":
        arg_dict["minimum phi"] = ""
    else:
        arg_dict["minimum phi"] = -max_phi
    arg_dict["maximum phi"] = max_phi
    arg_dict["phi resolution"] = phi_res
    arg_dict["niter"] = niter
    arg_dict["cutoff"] = cutoff
    if hogbom_clean == "":
        cleantype = ""
    elif hogbom_clean:
        cleantype = "Hogbom"
    else:
        cleantype = "Clark"
    arg_dict["CLEAN type"] = cleantype
    arg_dict["restoring beam fit to shortest wavelength psf"] = fitbeam
```

```
    if beamshape == None:
        beamshape = ("","","","")
    arg_dict["restoring beam major axis"] = beamshape[0]
    arg_dict["restoring beam minor axis"] = beamshape[1]
    arg_dict["restoring beam position angle"] = beamshape[2]
    arg_dict["restoring beam phi depth"] = beamshape[3]
    arg_dict["command"] = command
    _write_step(beginning, arg_dict, time)

def clear_fsclean():
    """Clears the Faraday synthesis step."""
    beginning = "Done fsclean"
    arg_dict = OrderedDict()
    arg_dict["image size"] = ""
    arg_dict["pixel size"] = ""
    arg_dict["minimum phi"] = ""
    arg_dict["maximum phi"] = ""
    arg_dict["phi resolution"] = ""
    arg_dict["niter"] = ""
    arg_dict["cutoff"] = ""
    arg_dict["CLEAN type"] = ""
    arg_dict["restoring beam fit to shortest wavelength psf"] = ""
    arg_dict["restoring beam major axis"] = ""
    arg_dict["restoring beam minor axis"] = ""
    arg_dict["restoring beam position angle"] = ""
    arg_dict["restoring beam phi depth"] = ""
    arg_dict["command"] = ""
    _write_step(beginning, arg_dict, time=None, havedone=False)

def check_fsclean():
    """Checks if the Faraday synthesis step has been done."""
    return _check_step("Done fsclean")

def read_fsclean(toread):
    """Reads a parameter value from the Faraday synthesis step."""
    return _read_from_step("Done fsclean", toread)

def write_table_info(source_id=None, philist="", valuelist="", I="", F_noise="",
                     F_bg="", I_noise="", I_bg="", reg_ra="", reg_dec="",
                     F_r="", I_r="", sigma="", time=None):
    """Writes an examined source from rmsynth_plot.

    Note that there may be several instances of this "step", differentiated
    by 'source_id'.
    """
```

```
    if source_id == None:
        beginning = "Examined"
    else:
        beginning = "Examined " + str(source_id)
    arg_dict = OrderedDict()
    arg_dict["central RA"] = reg_ra
    arg_dict["central dec"] = reg_dec
    arg_dict["F radius"] = F_r
    arg_dict["I radius"] = I_r
    phistr = ""
    for phi in philist:
        phistr += str(phi) + ","
    phistr = phistr[:-1]
    arg_dict["phi"] = phistr
    valstr = ""
    for val in valuelist:
        valstr += str(val) + ","
    valstr = valstr[:-1]
    arg_dict["F"] = valstr
    arg_dict["I"] = I
    arg_dict["F spectrum noise"] = F_noise
    arg_dict["F spectrum background"] = F_bg
    arg_dict["Threshold"] = sigma
    arg_dict["I image noise"] = I_noise
    arg_dict["I image background"] = I_bg
    _write_step(beginning, arg_dict, time)

def find_table_info():
    """Returns the ID of all examined sources in the status file."""
    thefile = open(STATUS_FILENAME, "rb")
    contents = thefile.readlines()
    ids = []
    for line in contents:
        truncline = line.split(":", 1)[0]
        splitline = truncline.split(" ", 1)
        if splitline[0] == "Examined":
            if len(splitline) == 2:
                ids.append(splitline[1])
            else:
                ids.append(None)
    return ids
```

```
def check_table_info(source_id=None):
    """Checks if a source with the given 'source_id' has been examined."""
    if source_id == None:
        id_str = ""
    else:
        id_str = " " + str(source_id)
    return _check_step("Examined" + id_str)

def read_table_info(toread, source_id=None):
    """Reads a parameter/result from the examined source 'source_id'."""
    if source_id == None:
        id_str = ""
    else:
        id_str = " " + str(source_id)
    return _read_from_step("Examined" + id_str, toread)

def clear_table_info():
    """Removes all the examined sources from the status file.

    Note that this currently deletes all steps after the first 'Examined', so
    take care if adding new steps.
    """
    beginning = "Examined"
    with open(STATUS_FILENAME, "rb") as thefile:
        contents = thefile.readlines()
        delind = None
        for ind, row in enumerate(contents):
            if row.split()[0] == beginning:
                delind = ind
                break
        if not delind == None:
            contents = contents[:delind]
    with open(STATUS_FILENAME,"w") as thefile:
        thefile.writelines(contents)
```

Converting fsclean output to FITS (h5_to_fits.py)

```
import sys

import h5py
from astropy.io import fits
from astropy.coordinates import SkyCoord
import numpy

def h5_to_fits(inname, outname, beamshape=None, radec=None):
    """Convert a hdf5 file from fsclean to a fits file readable by ds9.

    Args:
        inname: Name of the hdf5 file.
        outname: Name of the FITS file to output.
        beamshape: Parameters of the restoring beam.
        radec: Right ascension and declination on the form "RA DEC"
    """
    infile = h5py.File(inname,"r")
    dataset = infile.get("data")
    complex_data = dataset[:,:,:]
    p_data_before_flip = numpy.absolute(complex_data)
    # The axes are defined differently:
    numpy.swapaxes(p_data_before_flip, 0, 2)
    p_data = numpy.fliplr(p_data_before_flip)
    hdr = fits.Header()
    hdr.append(("simple", "T"))
    hdr.append(("bitpix", -64))
    hdr.append(("naxis", 3))
    hdr.append(("naxis1", p_data.shape[0]))
    hdr.append(("naxis2", p_data.shape[1]))
    hdr.append(("naxis3", p_data.shape[2]))
    hdr.append(("extend", "T"))
    hdr.append(("bunit", infile.attrs["image_units"]))
    hdr.append(("equinox", 2000))
    hdr.append(("btype", "intensity"))
    hdr.append(("origin", infile.attrs["origin"]))
    if not beamshape == None:
        hdr.append("bmaj", beamshape[0])
        hdr.append("bmin", beamshape[1])
        hdr.append("bpa", beamshape[2])
        hdr.append("bphi", beamshape[3])
    for i, j in enumerate([2, 1, 0]): # Mapping the different axes to each other
        hdr.append(("ctype"+str(i + 1),infile.attrs["axis_desc"][j]))
        hdr.append(("crpix"+str(i + 1),infile.attrs["crpix"][j]))
        hdr.append(("crval"+str(i + 1),infile.attrs["crval"][j]))
```

```
            hdr.append(("cdelt"+str(i + 1),infile.attrs["cdelt"][j]))
            hdr.append(("cunit"+str(i + 1),infile.attrs["axis_units"][j]))
        if not radec == None:
            coords = SkyCoord(radec)
            hdr["ctype1"] = "RA---SIN"
            hdr["ctype2"] = "DEC--SIN"
            hdr["cdelt1"] = -hdr["cdelt1"]*360*0.5/numpy.pi
            hdr["cdelt2"] = hdr["cdelt2"]*360*0.5/numpy.pi
            hdr["crval1"] = coords.ra.degree
            hdr["crval2"] = coords.dec.degree
        hdr.append(("date", infile.attrs["date"]))

        hdu = fits.PrimaryHDU(data=p_data, header=hdr)
        hdu.writeto(outname)
        infile.close()

if __name__ == "__main__":
    # Running from the command line will make a viewable image, but the file
    # will not include the beam information or a working coordinate system.
    arguments = sys.argv[1:]
    if not len(arguments) >= 2:
        sys.exit("First arg=in, second arg=out")
    inname = arguments[0]
    outname = arguments[1]
    h5_to_fits(inname, outname)
```

Image sources in parallel (queue_sources.py)

```
import multiprocessing
from time import sleep
from collections import OrderedDict

from anton_rm_synth import run as rmsynth

MAX_PROCESSES = 3

sourcelist = []
def add(dirname, **kwargs):
    if "phase_avg_arguments" in kwargs:
        if not "ra" in kwargs["phase_avg_arguments"]:
            kwargs["phase_avg_arguments"]["ra"] = "same"
        if not "dec" in kwargs["phase_avg_arguments"]:
            kwargs["phase_avg_arguments"]["dec"] = "same"
    else:
        kwargs["phase_avg_arguments"] = {"ra":"same","dec":"same"}
    kwargs["output_dir"] = dirname
    sourcelist.append(kwargs)
```

```
dirlist = OrderedDict()
dirlist["mulcahy5–6"] = ("13h39m23s", "46d40m08s")
dirlist["mulcahy7"] = ("13h41m45s", "46d57m17s")
dirlist["mulcahy4"] = ("13h37m08s", "48d58m04s")
dirlist["mulcahy3"] = ("13h32m59s", "45d42m03s")
dirlist["mulcahy2"] = ("13h31m33s", "45d39m30s")
dirlist["mulcahy1"] = ("13h26m29s", "47d37m42s")
dirlist["farnes15"] = ("13h33m35s", "46d58m08s")
dirlist["farnes14"] = ("13h33m24s", "47d29m35s")
dirlist["farnes13"] = ("13h32m45s", "47d22m23s")
dirlist["farnes11–12"] = ("13h31m25s", "47d13m01s")
dirlist["farnes9–10"] = ("13h30m45s", "47d03m17s")
dirlist["farnes8"] = ("13h30m40s", "46d47m05s")
dirlist["farnes7"] = ("13h30m32s", "47d30m53s")
dirlist["farnes5"] = ("13h29m37s", "46d59m06s")
dirlist["farnes4"] = ("13h28m18s", "46d46m17s")
dirlist["farnes3"] = ("13h27m57s", "47d42m51s")
dirlist["farnes2"] = ("13h27m03s", "47d05m43s")
dirlist["farnes1"] = ("13h25m48s", "47d26m05s")
dirlist["taylor4"] = ("13h43m06s", "45d32m48s")
dirlist["taylor5"] = ("13h21m28s", "45d47m33s")
dirlist["taylor6"] = ("13h37m39s", "47d41m48s")
dirlist["taylor8"] = ("13h24m54s", "48d22m17s")
dirlist["taylor9"] = ("13h25m32s", "45d16m32s")
dirlist["taylor10"] = ("13h17m31s", "46d18m33s")
dirlist["taylor11–22"] = ("13h30m49s", "48d21m07s")
dirlist["taylor13"] = ("13h39m13s", "45d28m10s")
dirlist["taylor14"] = ("13h32m22s", "45d23m40s")
dirlist["taylor15"] = ("13h42m08s", "46d42m39s")
dirlist["taylor16"] = ("13h20m05s", "46d39m38s")
dirlist["taylor17"] = ("13h22m14s", "49d38m42s")
dirlist["taylor18"] = ("13h37m29s", "48d18m08s")
dirlist["taylor20"] = ("13h28m20s", "49d44m37s")
dirlist["taylor21"] = ("13h24m29s", "47d43m23s")
dirlist["taylor23–36"] = ("13h36m16s", "49d00m10s")
dirlist["taylor24"] = ("13h37m31s", "46d21m18s")
dirlist["taylor25"] = ("13h19m23s", "48d44m54s")
dirlist["taylor26"] = ("13h37m56s", "45d30m46s")
dirlist["taylor27"] = ("13h44m03s", "48d35m42s")
dirlist["taylor28"] = ("13h33m24s", "48d56m14s")
dirlist["taylor29"] = ("13h22m22s", "45d57m56s")
dirlist["taylor30"] = ("13h24m15s", "44d41m10s")
dirlist["taylor32"] = ("13h20m52s", "45d41m11s")
dirlist["taylor33"] = ("13h39m48s", "47d41m16s")
dirlist["taylor35"] = ("13h27m47s", "48d42m06s")
```

```
dirlist["taylor37"] = ("13h28m54s", "48d17m48s")
dirlist["taylor38"] = ("13h30m48s", "48d00m06s")
dirlist["taylor39"] = ("13h34m35s", "45d11m58s")
dirlist["taylor40"] = ("13h24m44s", "46d39m37s")
dirlist["taylor41"] = ("13h34m51s", "46d27m44s")
dirlist["CHMAX1"] = None
dirlist["taylor2"] = ("13h26m23s", "49d34m43s")
dirlist["taylor42"] = ("13h36m12s", "46d06m22s")
dirlist["farnes6"] = ("13h30m09s", "47d11m31s")
dirlist["mao9"] = ("13h29m44s", "47d17m09s")
dirlist["mao11"] = ("13h29m35s", "47d06m47s")
dirlist["m51"] = ("13h29m51s", "47d12m18s")

specialargs = {"ALL":{"noreplace":False, "redo_pyrmsynth":True,
                      "pyrmsynth_arguments":{"min_phi":-200, "max_phi":200,
                                             "phi_res":0.1,"niter":50000}}}
sizes = {"taylor2":512, "taylor42":512, "m51":1024, "farnes6":512, "mao9":512,
         "mao11":512}

for d in dirlist:
    kwargs = {}
    if not dirlist[d] == None:
        kwargs.update({"phase_avg_arguments":{"ra":dirlist[d][0],
                                              "dec":dirlist[d][1]}})
    if d in sizes:
        size = sizes[d]
        niter = 25000*(size/256)**2
        kwargs.update({"wsclean_arguments":{"imagesize":size, "niter":niter}})
    if d in specialargs:
        kwargs.update(specialargs[d])
    if "ALL" in specialargs:
        kwargs.update(specialargs["ALL"])
    add(d, **kwargs)

processlist = []
oldactive = []
while len(sourcelist) > 0 or len(multiprocessing.active_children()) > 0:
    active = multiprocessing.active_children()
    act_names = []
    for a in active:
        act_names.append(a.name)
    if not act_names == oldactive:
        oldactive = act_names
        print("Active processes:")
        print(act_names)
        print(str(len(sourcelist)) + " pending.")
```

```
    if len(active)<MAX_PROCESSES and len(sourcelist)>0:
        kwargs = sourcelist.pop(0)
        if kwargs["output_dir"][0:5]=="CHMAX":
            MAX_PROCESSES = int(kwargs["output_dir"][5:])
            print("Changed maximum number of processes to "+str(MAX_PROCESSES))
        else:
            p = multiprocessing.Process(target=rmsynth,
                                        name=kwargs["output_dir"], kwargs=kwargs)
            processlist.append(p)
            p.start()
            print("started "+str(kwargs))
    else:
        sleep(3)
print("Done!")
```

Examining sources (rmsynth_plot.py)

```
import os
import shutil
import sys
import itertools
import copy
import math
from collections import Iterable

from astropy.io import fits
from astropy import constants as const
from astropy import wcs
from astropy.modeling import models, fitting
import numpy
import scipy
from scipy import ndimage
from scipy import integrate
import h5py
from matplotlib import pyplot,animation,patches,ticker
from matplotlib import cm as colormap
import matplotlib
from IPython import embed

import statusfile

#General options
RMSYNTH_DIR = "RM_synthesis"
RMSYNTH_PREFIX = "out"

TEXT_SIZE = 16
LEGEND = False
```

```
ANIM_DISP_LENGTH = 10 # The length of the animated line.
ANIM_INTERVAL = 1000 # Time between animation frames.

# Save the data in a .npz file. Somewhat improves loading times, but increases
# disk usage. Will load all data.
SAVE_DATA = False
DATA_FILENAME="rmplot_data"

LOAD_ALL = True # Loads all data instead of an area around the region.
LOAD_FACTOR = 5 # How much bigger than the region the loaded area is.


# This is here to make the program easier to extend with multiple SkyMaps,
# if that is wanted.
current_sky_map = None

def main(input_dir):
    """Create a SkyMap and a measurement, and starts embedded IPython."""
    global current_sky_map
    if not os.path.isdir("./" + input_dir):
        sys.exit("The directory " + input_dir + " isn't here!")
    os.chdir("./" + input_dir)
    sky_map = SkyMap()
    current_sky_map = sky_map
    meas = Measurement(sky_map)
    embed()

def model_q_and_u(reg,depol_model=None):
    """Fit sine and cosine to q and u, with depolarization, in mulcahy5."""
    reg.add_plot("l2_q", color="red")
    reg.add_plot("l2_u", color="blue")
    if not depol_model == None:
        depol = make_model(depol_model)
        if depol_model == "depol_tribble_qsf":
            kwargs = {"st":0.5, "sigma":0.5, "offset":0.05}
        elif depol_model == "depol_tribble_long":
            kwargs = {"amp":0.5, "offset":0.05}
        elif depol_model == "depol_burn":
            kwargs = {"amp":0.5, "sigma":0.5, "offset":0.05}
        reg.fit_model(depol, "l2", "p", **kwargs)
        # I did not manage to fit them jointly
        q_model = make_model("sine_depol", depol_model=depol, amplitude=1,
                             rm=20.75, phase=0.5, offset=0.05)
        u_model = make_model("cosine_depol", depol_model=depol, amplitude=1,
                             rm=20.75, phase=0, offset=0.05)
```

```
    else:
        depol = None
        q_model = make_model("sine", amplitude=1, rm=20.75,
                  phase=0.5, offset=0.05)
        u_model = make_model("cosine", amplitude=1,
                  rm=20.75, phase=0, offset=0.05)
    reg.fit_model(q_model, "l2", "q")
    reg.fit_model(u_model, "l2", "u")
    reg.plot_model(q_model, 0, color="black")
    reg.plot_model(u_model, 1, color="black")
    return q_model, u_model, depol

def noise_plots(reg, start_r=1, stop_r=21):
    """Plot MFS I noise, and spectrum and image noises for F,
    as a function of region radius.
    """
    noises = {"MFS_I":[], "image_F":[], "spectrum_F":[]}
    rlist = range(start_r, stop_r)
    for r in rlist:
        reg.change_size(r, absolute=True)
        reg.skymap.fig.canvas.draw()
        reg.update_data("MFS_I_noise")
        noises["MFS_I"].append(reg.MFS_I_noise)
        noises["image_F"].append(reg.im_noise("F"))
        noises["spectrum_F"].append(reg.spec_noise("F")[0])
    for quantity in noises:
        noises[quantity] = numpy.array(noises[quantity])
    fig, (iax, fax, sfax) = pyplot.subplots(1, 3)
    iax.plot(rlist, 1000000*noises["MFS_I"])
    fax.plot(rlist, 1000000*noises["image_F"])
    sfax.plot(rlist, 1000000*noises["spectrum_F"])
    for ax in (iax, fax, sfax):
        ax.get_yaxis().get_major_formatter().set_powerlimits((-3, 4))
        ax.set_xlabel("r ['']", fontsize=TEXT_SIZE)
        ax.tick_params(axis="both", which="major", labelsize=TEXT_SIZE)
        yticks = ax.yaxis.get_major_ticks()
        yticks[0].label1.set_visible(False)
    iax.set_ylabel("std(I) [$\mu$Jy/pix]", fontsize=TEXT_SIZE)
    fax.set_ylabel("Image std(F) [$\mu$Jy/pix/rmsf]", fontsize=TEXT_SIZE)
    sfax.set_ylabel("Spectrum std(F) [$\mu$Jy/rmsf]", fontsize=TEXT_SIZE)
    pyplot.show(block=False)

def list_regions():
    """Return a string with an overview of all regions."""
    return current_sky_map.list_regions()
```

```
def add_region(x, y, r=10):
    """Add a region at (x, y)."""
    return current_sky_map.add_region(x,y,r)

def get_regions():
    """Return a list of the regions."""
    return current_sky_map.get_regions()

def make_model(model_type, name=None, **kwargs):
    """Create a model of the specified model_type.

    Returns a custom model in the astropy modeling framework:
    http://docs.astropy.org/en/stable/api/astropy.modeling.custom_model.html

    If name=None, model_type is used as the name. Some models take extra kwargs,
    see below. Remaining kwargs are used to specify the parameter values.

    Valid models:
        "depol_tribble_qsf": Depolarization with quadratic structure function.
            See Tribble (1991). Parameters: st=s_0/t, sigma, amp, offset
        "depol_long": Long wavelength approximation as in Tribble (1991).
            Amplitude includes s_0/t and sigma. Parameters: amp, offset
        "depol_sinc": Depolarization by a sinc function from a slab,
            as used in Burn (1966). Parameters: Dphi, amp, offset
        "specind_power_law": The spectral dependence as a power law.
            Parameters: x_0, alpha, amp
        "gaussian_slab": Model a peak as a Gaussian.
            Parameters: amplitude, mean, stddev
        "sine": Parameters: amplitude, rm, phase, offset
        "cosine": Parameters: amplitude, rm, phase, offset
        "sine_depol": Sine multiplied by a depolarization model.
            Need to specify a kwarg: depol_model. The model must contain
            parameters amp and offset. Parameters: as "sine"
        "cosine_depol": Cosine multiplied by a depolarization model.
            Need to specify a kwarg: depol_model. The model must contain
            parameters amp and offset. Parameters: as "cosine"
    """
    if name == None:
        name = model_type
```

```
if model_type == "depol_tribble_qsf":
    @models.custom_model
    def depol_tribble_qsf(l2, st=0, sigma=0, amp=0, offset=0):
        num = 1 - numpy.exp(-0.5*st**2 - 4*(sigma*l2)**2)
        denom = 1 + 8*(sigma*l2/st)**2
        term = numpy.exp(-0.5*st**2 - 4*(sigma*l2)**2)
        return amp*numpy.sqrt(num/denom + term) + offset
    bounds = {"st":[0, None], "sigma":[None, None], "amp":[0, 0.7],
              "offset":[None, None]}
    model = depol_tribble_qsf(bounds=bounds, name=name, **kwargs)
elif model_type == "depol_long":
    @models.custom_model
    def depol_long(l2, amp=0, offset=0):
        # amp = st/(sigma*2*sqrt(2))
        return amp/l2 + offset
    bounds = {"amp":[0, None], "offset":[None, None]}
    model = depol_long(bounds=bounds, name=name, **kwargs)
elif model_type == "depol_burn":
    @models.custom_model
    def depol_burn(l2, amp=0, sigma=0, offset=0):
        return amp*numpy.exp(-2 * sigma**2 * l2**2) + offset
    bounds = {"amp":[0, 0.7], "sigma":[None, None], "offset":[None, None]}
    model = depol_burn(bounds=bounds, name=name, **kwargs)
elif model_type == "depol_sinc":
    @models.custom_model
    def depol_sinc(l2, amp=0, Dphi=0, offset=0):
        return numpy.absolute(amp*numpy.sinc(Dphi*l2)) + offset
    bounds = {"amp":[0.1, None], "Dphi":[None, None], "offset":[None, None]}
    model = depol_sinc(bounds=bounds, name=name, **kwargs)
elif model_type == "specind_power_law":
    @models.custom_model
    def specind_power_law(nu, amp=0, nu_0=0, alpha=0):
        return amp * (nu/nu_0)**alpha
    bounds = {"amp":[0, None], "x_0":[None, None], "alpha":[None, None]}
    model = specind_power_law(bounds=bounds, name=name, **kwargs)
elif model_type == "gaussian_slab":
    bounds = {"amplitude":[None, None],
              "mean":[None, None], "stddev":[0.25, 3]}
    model = models.Gaussian1D(bounds=bounds, name=name, **kwargs)
elif model_type == "sine":
    @models.custom_model
    def sine(l2, amplitude=0, rm=0, phase=0, offset=0):
        return amplitude*numpy.sin(2*rm*l2 + 2*numpy.pi*phase) + offset
    bounds = {"amplitude":[0.02, None], "rm":[0, None], "phase":[0, 1],
              "offset":[None, None]}
    model = sine(bounds=bounds, name=name, **kwargs)
```

```
    elif model_type == "cosine":
        @models.custom_model
        def cosine(l2, amplitude=0, rm=0, phase=0, offset=0):
            return amplitude*numpy.cos(2*rm*l2 + 2*numpy.pi*phase) + offset
        bounds = {"amplitude":[0.02, None], "rm":[0, None], "phase":[0, 1],
                  "offset":[None, None]}
        model = cosine(bounds=bounds, name=name, **kwargs)
    elif model_type == "sine_depol":
        depol_model = kwargs["depol_model"]
        depol_amp = depol_model.amp.value
        depol_offset = depol_model.offset.value
        del kwargs["depol_model"]
        @models.custom_model
        def sine_depol(l2, amplitude=0, rm=0, phase=0, offset=0):
            return (amplitude*numpy.sin(2*rm*l2 + 2*numpy.pi*phase)
                    *((depol_model(l2) - depol_offset)/depol_amp) + offset)
        bounds = {"amplitude":[0, None], "rm":[None, None], "phase":[-1, 1],
                  "offset":[None, None]}
        model = sine_depol(bounds=bounds, name=name, **kwargs)
    elif model_type == "cosine_depol":
        depol_model = kwargs["depol_model"]
        depol_amp = depol_model.amp.value
        depol_offset = depol_model.offset.value
        del kwargs["depol_model"]
        @models.custom_model
        def cosine_depol(l2, amplitude=0, rm=0, phase=0, offset=0):
            return (amplitude*numpy.cos(2*rm*l2 + 2*numpy.pi*phase)
                    *((depol_model(l2) - depol_offset)/depol_amp) + offset)
        bounds = {"amplitude":[0, None], "rm":[None, None], "phase":[-1, 1],
                  "offset":[None, None]}
        model = cosine_depol(bounds=bounds, name=name, **kwargs)
    return model

def make_and_fit(region, model_type, xaxis, yaxis, name=None,
                 fitter=None, **kwargs):
    """Makes a model and fits it to the data in the specified region.

    See make_model and Region.fit_model for information about the arguments.
    """
    model = make_model(model_type, name)
    region.fit_model(model, xaxis, yaxis, fitter=fitter, **kwargs)
    return model
```

```
class SkyMap(object):
    """The entity controlling the raw data and the regions.

    Middle clicking on the source image will create a region. Regions can be
    moved around by dragging them with the mouse, and deleted by right clicking
    on them. Scrolling the mouse wheel over a region will resize it.

    After creating a region, it is most easily accessed by using get_regions.
    """

    def __init__(self):
        """Creates a new SkyMap instance from the MFS I file."""
        #Open the MFS I image to show.
        print("Loading total intensity image")
        if os.path.isfile("./MFS/wscleaned-MFS-I-image-pb.fits"):
            MFS_filename = "./MFS/wscleaned-MFS-I-image-pb.fits"
        # These are for compability with previous versions of rmsynth.
        elif os.path.isfile("./MFS/wscleaned-MFS-I-image.fits"):
            MFS_filename = "./MFS/wscleaned-MFS-I-image.fits"
            print("WARNING: The MFS file appears to not be beam corrected!")
        elif os.path.isfile("./stokes_i/wscleaned-MFS-I-image.fits"):
            MFS_filename="./stokes_i/wscleaned-MFS-I-image.fits"
            print("WARNING: This is very old, better not use it for any results!")
        else:
            sys.exit("No MFS file found!")
        self.image_data = fits.getdata(MFS_filename)[0, 0, :, :]
        self.image_header = fits.getheader(MFS_filename)
        self.image_size = self.image_data.shape[0]
        # To use for converting from Jy/beam to Jy
        self.pixel_size = abs(self.image_header["cdelt1"]*3600)
        self.bmaj = self.image_header["bmaj"]*3600
        self.bmin = self.image_header["bmin"]*3600
        self.beamfactor = self.pixel_size**2 / (self.bmaj*self.bmin*1.13)
        self.image_data = self.image_data*self.beamfactor
        # Find the number of channels
        self.channels = int(statusfile.read_wsclean("channels after wsclean"))
        # Initialize the data variables
        self.nu = None
        self.block_inds = None
        self.lambda2 = None
        # Calculate phi axis.
        hdr = fits.getheader("./" + RMSYNTH_DIR + "/" + RMSYNTH_PREFIX
                             + "_clean_p.fits")
        crval = hdr["crval3"]
        naxis = hdr["naxis3"]
        cdelt = hdr["cdelt3"]
```

```
self.phi = numpy.linspace(crval, crval + (naxis-1)*cdelt, naxis)
self.F = numpy.empty((len(self.phi), self.image_size, self.image_size),
                     dtype=complex)
self.P = numpy.empty((self.channels, self.image_size, self.image_size),
                     dtype=complex)
self.I = numpy.empty((self.channels, self.image_size, self.image_size),
                     dtype=float)
self.loaded = numpy.zeros((self.image_size, self.image_size),
                          dtype=bool)
self.loaded_drawer = None # The matplotlib Artist drawing loaded area
if LOAD_ALL:
    print("Loading entire image")
    self.load_data(0, self.image_size, 0, self.image_size)
if SAVE_DATA:
    if os.path.isfile(DATA_FILENAME + ".npz"):
        print("Loading data from file.")
        datafile = numpy.load(DATA_FILENAME + ".npz")
        self.F = datafile["F"]
        self.P = datafile["P"]
        self.I = datafile["I"]
        self.nu = datafile["nu"]
        self.block_inds = datafile["block_inds"]
        self.lambda2 = datafile["lambda2"]
        self.loaded[:, :] = 1
        print("Done")
    else:
        if not LOAD_ALL:
            self.load_data(0, self.image_size, 0, self.image_size)
        print("Saving data in file")
        numpy.savez(DATA_FILENAME, F=self.F, P=self.P, I=self.I,
                    nu=self.nu, block_inds=self.block_inds,
                    lambda2=self.lambda2)
        print("Done")
self.fig = pyplot.figure()
self.ax = self.fig.add_subplot(111)
self.__draw_map()
self.region_list = []
self.moving_list = [] # List containing all currently moving regions.
#Connecting events
self.click_id = self.fig.canvas.mpl_connect('button_press_event',
                                            self.__onclick)
self.scroll_id = self.fig.canvas.mpl_connect('scroll_event',
                                             self.__onscroll)
self.release_id = self.fig.canvas.mpl_connect('button_release_event',
                                              self.__onrelease)
```

```
        self.move_id = self.fig.canvas.mpl_connect('motion_notify_event',
                                                   self.__onmove)

    def __draw_map(self):
        # Draws the I image of the source.
        ax = self.ax
        image = ax.imshow(self.image_data, cmap="hot", interpolation="nearest",
                          origin="lower", vmax=numpy.amax(self.image_data)
                          + 0.00001) # This term prevents wraparound
        # To make the image coordinates output in ra,dec instead of pixels
        def format_coord(x,y):
            w = wcs.WCS(self.image_header)
            sky = wcs.utils.pixel_to_skycoord(x, y, w)
            return sky.to_string("hmsdms", precision=1)
        ax.format_coord = format_coord
        self.fig.colorbar(mappable=image)
        pyplot.show(block=False)

    def __draw_loaded(self):
        # Draws the edges of the loaded area in white.
        print("Displaying loaded region")
        struct = ndimage.generate_binary_structure(2, 2)
        erode = ndimage.binary_erosion(self.loaded, struct)
        edges = self.loaded^erode
        edges = numpy.ma.masked_where(edges==0, edges)
        if self.loaded_drawer == None:
            self.loaded_drawer = self.ax.imshow(edges, interpolation="nearest",
                                                origin="lower", cmap="binary")
        else:
            self.loaded_drawer.set_data(edges)
        self.fig.canvas.draw()

    def add_region(self, x, y, r=10, **kwargs):
        """Add a new region to the SkyMap.

        x, y: The position in pixels.
        r: The radius in pixels
        **kwargs will be passed on to the constructor of Region.
        """
        possible_colors = []
        possible_colors.append("green")
        possible_colors.append("yellow")
        possible_colors.append("cyan")
        possible_colors.append("red")
```

```
    for reg in self.region_list:
        other_color = reg.color
        if other_color in possible_colors:
            possible_colors.remove(other_color)
    if not possible_colors:
        colors = ["red"]
    newreg = Region(self, x, y, color=possible_colors[0],
                    radius=r, **kwargs)
    self.region_list.append(newreg)
    return newreg

def list_regions(self):
    """Return a string with an overview of the regions in the SkyMap."""
    return_str = ""
    for i, reg in enumerate(self.region_list):
        return_str += "---Region " + str(i) + ": " + reg.color + "----\n"
        plots = reg.list_plots()
        plots_list = plots.split("\n")
        plots_list = ["    " + line for line in plots_list]
        plots = "\n".join(plots_list)
        return_str += plots + "\n"
    return return_str

def get_regions(self):
    """Return a copy of the list of regions in the map."""
    return copy.copy(self.region_list)

def __onclick(self, event):
    # When clicking in the map.
    if event.xdata != None and event.ydata != None:
        if event.button == 1:
            # Start moving any regions clicked in.
            for reg in self.region_list:
                if reg.contains(event.xdata, event.ydata):
                    self.moving_list.append(reg)
        if event.button == 2:
            # If clicking in a region: Redraw the plots.
            # If region belongs to a Measurement: Sample the noise.
            # If not clicking in a region: Create a new region.
            create_new = True
            for reg in self.region_list:
                if reg.contains(event.xdata, event.ydata):
                    reg.redraw_plots()
                    create_new = False
                    if not reg.measurement == None:
                        reg.measurement.add_noise(reg.measuring)
```

```
            if create_new:
                self.add_region(event.xdata, event.ydata)
        elif event.button == 3:
            # Remove any regions clicked in.
            for reg in self.region_list:
                if reg.contains(event.xdata, event.ydata):
                    reg.remove()

def __onmove(self, event):
    # Move all dragged regions.
    for reg in self.moving_list:
        reg.move(event.xdata, event.ydata)
        self.fig.canvas.draw()

def __onrelease(self, event):
    # Stop moving regions.
    if event.button == 1:
        self.moving_list = []

def __onscroll(self, event):
    # Change region size.
    if event.xdata != None and event.ydata != None:
        for reg in self.region_list:
            if reg.contains(event.xdata, event.ydata):
                if event.button == "up":
                    reg.change_size(0.5)
                elif event.button == "down":
                    reg.change_size(-0.5)
                self.fig.canvas.draw()

def load_data(self, xleft=0, xright=256, ydown=0, yup=256):
    """Load data in a rectangle."""
    if xleft < 0:
        xleft = 0
    if xright > self.image_size:
        xright = self.image_size
    if ydown < 0:
        ydown = 0
    if yup > self.image_size:
        yup = self.image_size
    # This assumes that the files in alphabetical order are also in
    # frequency order, which they will be from rmsynth.
    i_filelist = os.listdir("./stokes_i")
    i_filelist.sort()
    q_filelist = os.listdir("./stokes_q")
    q_filelist.sort()
```

```
u_filelist = os.listdir("./stokes_u")
u_filelist.sort()
# Only load frequencies if they haven't been loaded.
if self.nu == None:
    self.nu = numpy.empty(self.channels)
    do_freqs=True
else:
    do_freqs=False
# Load Q, U data
for ind, i_name in enumerate(i_filelist):
    q_name = q_filelist[ind]
    u_name = u_filelist[ind]
    if ind % 100 == 0:
        print("channel " + str(ind) + " done.")
    i_path = "./stokes_i/" + i_name
    i_data = fits.getdata(i_path)[0, 0, xleft:xright, ydown:yup]
    i_data = i_data * self.beamfactor # Jy/pix instead of Jy/beam
    self.I[ind, xleft:xright, ydown:yup] = i_data
    q_path = "./stokes_q/"+q_name
    u_path = "./stokes_u/"+u_name
    re = fits.getdata(q_path)[0, 0, xleft:xright, ydown:yup]
    im = fits.getdata(u_path)[0, 0, xleft:xright, ydown:yup]
    self.P[ind, xleft:xright, ydown:yup] = (re + 1j*im)*self.beamfactor
    if do_freqs:
        freqs = fits.getheader("./stokes_i/" + i_name)["crval3"]
        self.nu[ind] = freqs
# Load frequencies and separate into blocks
if do_freqs:
    # Find where the different blocks are by looking for unusually
    # large jumps.
    jump_inds = [i for i, delt in enumerate(numpy.diff(self.nu))
                 if delt > numpy.average(numpy.diff(self.nu))]
    self.block_inds = []
    for i, ind in enumerate(jump_inds):
        if i == 0:
            self.block_inds.append((0, ind))
        else:
            self.block_inds.append((jump_inds[i - 1] + 1, ind))
        if i == len(jump_inds) - 1:
            self.block_inds.append((ind + 1, len(self.nu) - 1))
    self.lambda2 = (const.c.value/self.nu)**2
print("Loading Faraday Q and U.")
Fq_path = "./" + RMSYNTH_DIR + "/" + RMSYNTH_PREFIX + "_clean_q.fits"
Fu_path = "./" + RMSYNTH_DIR + "/" + RMSYNTH_PREFIX + "_clean_u.fits"
re = fits.getdata(Fq_path)[:, xleft:xright, ydown:yup]
im = fits.getdata(Fu_path)[:, xleft:xright, ydown:yup]
```

```
        self.F[:, xleft:xright, ydown:yup] = (re + 1j*im)*self.beamfactor
        self.loaded[xleft:xright, ydown:yup] = True
        print("Data loaded.")
        if not (LOAD_ALL or SAVE_DATA):
            self.__draw_loaded()

class Region(object):
    """A region to plot quantities at different locations in the source."""

    def __init__(self, skymap, center_x, center_y, radius=10, color="green",
                 blockcolors="rainbow", silent=False):
        """Creates a new region.

        Args:
            skymap: The SkyMap object the region belongs to.
            center_x, center_y: The position of the region.
            radius: The radius of the region.
            color: The color of the region when drawn on the SkyMap.
            blockcolors: Colors of the different blocks.
                Values are "rainbow" and "black".
            silent: If True, no plots will be made from this region.
        """
        self.skymap = skymap
        self.color = color
        self.circle = patches.Circle((center_x, center_y), radius, color=color,
                                     fill=False)
        self.locked_r = False
        self.silent = silent
        self.connected_to = set() # Other regions to move along with.
        self.measurement = None
        self.measuring = None
        self.skymap.ax.add_artist(self.circle)
        self.skymap.fig.canvas.draw()
        # To be initialized in update_data, declared here for readability only
        self.F = None
        self.F_noise = None
        self.I = None
        self.I_noise = None
        self.P = None
        self.P_noise = None
        self.npixels = None
        self.MFS_I = None
        self.MFS_I_noise = None
        self.up_to_date = {"I":False, "I_noise":False, "P":False,
                           "P_noise":False, "F":False, "F_noise":False,
                           "MFS_I":False, "MFS_I_noise":False}
```

```
    for quant in self.up_to_date:
        self.update_data(quant)
    self.paused = False
    self.l2 = skymap.lambda2
    self.nu = skymap.nu
    self.phi = skymap.phi
    self.block_inds = skymap.block_inds
    cm = colormap.get_cmap("gist_rainbow")
    self.block_colors = []
    for block in self.skymap.block_inds:
        if blockcolors == "rainbow":
            centfreq = (self.nu[block[1]] + self.nu[block[0]])/2
            freqrange = self.nu[-1] - self.nu[0]
            proportion = (centfreq - self.nu[0])/freqrange
            self.block_colors.append(cm(proportion))
        elif blockcolors == "black":
            self.block_colors.append([0, 0, 0])
    self.plotlist = []
    self.model_lines = []
    self.updatable_plots = []
    self.anims = []
    if not self.silent:
        self.fig = pyplot.figure()

def redraw_plots(self):
    """Redraws the plots."""
    self.__make_figure()

def pause(self):
    """The region will not update when moved, until resume() is called."""
    self.paused = True

def resume(self):
    """Undoes the effects of pause()."""
    self.paused = False
    self.__hide_models()
    self.__update_plots()

def connect(self, reg):
    """Connect to another region to keep the same center."""
    self.connected_to.add(reg)
    reg.connected_to.add(self)
```

```
def add_plot(self, plotstring="custom", index=None, redraw=True, **kwargs):
    """Add a plot to the region figure.

    Args:
        plotstring: A shorthand to use common parameters. These can
            still be modified by explicitly giving them as kwargs.
            Implemented plotstrings are "l2_p", "l2_q", "l2_u", "phi_F",
            "phi_Fq", "phi_Fu", "nu_I". The first part of each plotstring
            is the x-axis and the second is the y-axis, with names defined
            as in get_quantity.
        index: At which position to add the plot. If there is a plot there,
            draw in the same plot.
        redraw: If False, will not redraw the figure of the region.
    Valid kwargs:
        xaxis, yaxis: Which quantities to plot against each other. For a
            list of available quantities, see documentation for
            get_quantity.
        loglog: If true, uses logarithmic axes.
        square: Use square axes.
        avg_block: Whether to average each block into a single point.
        errorbar_x, errorbar_y: Whether to have error bars when averaging.
        x_prefix, y_prefix: Use larger or smaller units for the respective
            quantities. Valid prefixes are "m", "k" and "M".
        animate: Whether to animate the plot.
        anim_increasing_nu: If True, the animation goes from low frequency
            to high, otherwise the reverse.
        Other kwargs are passed to 'plot' in matplotlib.

    """
    if plotstring == "l2_p":
        args = {"xaxis":"l2", "yaxis":"p", "avg_block":True, "fmt":"o"}
    elif plotstring == "l2_q":
        args = {"xaxis":"l2", "yaxis":"q", "linestyle":"None", "marker":".",
                "markersize":4}
    elif plotstring == "l2_u":
        args = {"xaxis":"l2", "yaxis":"u", "linestyle":"None", "marker":".",
                "markersize":4}
    elif plotstring == "phi_F":
        args = {"xaxis":"phi", "yaxis":"F", "color":"k", "y_prefix":"m"}
    elif plotstring == "phi_Fq":
        args = {"xaxis":"phi", "yaxis":"Fq", "color":"r", "y_prefix":"m"}
    elif plotstring == "phi_Fu":
        args = {"xaxis":"phi", "yaxis":"Fu", "color":"b", "y_prefix":"m"}
```

```
    elif plotstring == "nu_I":
        args = {"xaxis":"nu", "yaxis":"I", "loglog":True,
                "linestyle":"None", "marker":".", "markersize":4,
                "x_prefix":"M", "y_prefix":"m"}
    else:
        args = {}
    args.update(kwargs)
    if index == None or index == len(self.plotlist):
        self.plotlist.append([args])
    else:
        self.plotlist[index].append(args)
    if redraw:
        self.redraw_plots()

def remove_plot(self, index=-1, redraw=True):
    """Remove the plot at the specified index. Default: last plot."""
    del self.plotlist[index]
    if redraw:
        self.redraw_plots()

def list_plots(self):
    """Return a string with an overview of the plots."""
    returnstr = ""
    for i, plot in enumerate(self.plotlist):
        returnstr += "plot " + str(i) + ": " + str(plot) + "\n"
    return(returnstr)

def contains(self, x, y):
    """Evaluate if the coordinates (in pixels) lie inside the region."""
    circ_x, circ_y = self.circle.center
    return (x - circ_x)**2 + (y - circ_y)**2 <= self.circle.radius**2

def move(self, x, y):
    """Moves the region to x, y."""
    self.circle.center = (x, y)
    # Mark the quantities in the region as outdated, to update the used
    # ones when needed.
    for quant in self.up_to_date:
        self.up_to_date[quant] = False
    # Move any connected regions along.
    for reg in self.connected_to:
        if not reg.circle.center == (x, y):
            reg.move(x, y)
    if not self.paused:
        self.__hide_models() # Any fitted models are no longer valid.
        self.__update_plots()
```

```
def change_size(self, delta_r, absolute=False):
    """Change the radius of the region.

    Args:
        delta_r: How much to increase or decrease the radius.
        absolute: If True, instead set the radius to delta_r.
    """
    if not self.locked_r:
        # Mark the quantities in the region as outdated, to update the used
        # ones when needed.
        for quant in self.up_to_date:
            self.up_to_date[quant] = False
        if absolute:
            self.circle.radius = delta_r
        else:
            self.circle.radius += delta_r
        # Remove the region if it is too small.
        if self.circle.radius < 0.5:
            self.remove()
        elif not self.paused:
            self.__hide_models() # Any fitted models are no longer valid.
            self.__update_plots()

def remove(self):
    """Remove the region from the SkyMap."""
    if not self.silent:
        pyplot.close(self.fig)
    self.circle.remove()
    self.skymap.fig.canvas.draw()
    self.skymap.region_list.remove(self)

def plot_model(self,model,index=0,**kwargs):
    """Plot the given model in plot at the given index.

    kwargs are passed to matplotlib's plot. The default color is black.
    """
    if not self.silent:
        # Make the default color black
        if not "color" in kwargs:
            kwargs["color"] = "black"
        ax = self.plotlist[index][0]["ax"]
        xlims = ax.get_xlim()
        x = numpy.linspace(xlims[0], xlims[1], 2000)
        y = model(x)
        line, = ax.plot(x, y, label=model.name, **kwargs)
```

```
        if LEGEND:
            ax.legend()
        self.fig.canvas.draw()
        self.model_lines.append(line)

def fit_model(self, model, xaxis, yaxis, fitter=None,
              blocks=None, **kwargs):
    """Fit the given model to the data in xaxis and yaxis.

    Args:
        fitter: The astropy fitter to use. Default is LevMarLSQFitter()
        blocks: Fit only to specific frequency blocks.

    kwargs are starting values for the model parameters. They can be lists
    of values. In that case, the method fits the model to the data using
    all combinations of the starting values, and chooses the best fit."""
    if fitter == None:
        fitter = fitting.LevMarLSQFitter()
    x = self.get_quantity(xaxis)["data"]
    y = self.get_quantity(yaxis)["data"]
    if not blocks == None:
        new_x = numpy.array([], dtype="f")
        new_y = numpy.array([], dtype="f")
        for i, block in enumerate([self.block_inds[j] for j in blocks]):
            new_x = numpy.append(new_x, x[slice(*block)])
            new_y = numpy.append(new_y, y[slice(*block)])
        x = new_x
        y = new_y
    runlist = [] # List of fits with different parameter combinations
    parameters = {}
    for param, model_val in zip(model.param_names, model.parameters):
        if param in kwargs:
            if isinstance(kwargs[param], Iterable):
                parameters[param] = kwargs[param]
            else:
                parameters[param] = [kwargs[param]]
        else:
            parameters[param] = [model_val]
    paramnames = parameters.keys()
    paramvalues = parameters.values()
    starting_values = itertools.product(*paramvalues)
```

```
    # Run all fits
    for parvals in starting_values:
        param_dict = dict(zip(paramnames, parvals))
        modelin = model.copy()
        for i, name in enumerate(modelin.param_names):
            modelin.parameters[i] = param_dict[name]
        modelout = fitter(modelin, x=x, y=y, maxiter=1000)
        msg = fitter.fit_info["message"]
        lsq = numpy.sum((model(x) - y)**2)
        runlist.append({"model":modelout, "msg":msg, "lsq":lsq})
    minrun = min(runlist, key=lambda a:a["lsq"]) # Find the best fit
    model.parameters = minrun["model"].parameters
    print("------Model: " + model.name + "------")
    print(zip(model.param_names, model.parameters))
    print("lsq=" + str(minrun["lsq"]))
    print(minrun["msg"])

def __hide_models(self):
    # Remove models from the plots
    if not self.silent:
        for line in self.model_lines:
            ax = line.axes
            line.remove()
            if LEGEND:
                ax.legend() # Bug: The legend is not always removed.
        self.fig.canvas.draw()
        self.model_lines = []

def subtract_model(self, model, xaxis, yaxis):
    """Subtract the given model from the data of the region.

    After moving or otherwise updating the region this subtraction will
    no longer apply.

    When subtracting P or F models, the subtracted real and imaginary
    components are chosen to preserve the complex angle.

    xaxis and yaxis are the quantities used in the model.
    """
    x = self.get_quantity(xaxis)["data"]
    to_remove = model(x)
    if yaxis.lower() in ["p", "q", "u", "i"]:
        if yaxis.islower():
            to_remove = self.I * to_remove
```

```
        if yaxis.lower() == "p":
            # Preserves the complex angle. Kind of arbitrary.
            self.P = self.P - to_remove*self.P/numpy.absolute(self.P)
        elif yaxis.lower() == "q":
            self.P = self.P - to_remove # Remember that P is complex.
        elif yaxis.lower() == "u":
            self.P = self.P - 1j*to_remove
        elif yaxis.lower() == "i":
            self.I = self.I - to_remove
    elif yaxis.lower() == "F":
    # Preserves the complex angle. Kind of arbitrary.
        self.F = self.F - to_remove*self.F/numpy.absolute(self.F)
    elif yaxis.lower() == "Fq":
        self.F = self.F - to_remove # F is also complex.
    elif yaxis.lower() == "Fu":
        self.F = self.F - 1j*to_remove
    else:
        print("Subtracting a model from " + yaxis + " is not implemented.")
    self.__update_plots()


def __make_figure(self):
    # Clears the region figure and adds all plots in the list
    if not self.silent:
        self.anims = []
        self.__hide_models()
        self.fig.clf()
        # This creates one column of plots,  but changing that should not
        # create any problems.
        subplotx = 1
        subploty = len(self.plotlist)
        for i, plots in enumerate(self.plotlist):
            new_ax = self.fig.add_subplot(subploty, subplotx, i + 1)
            for args in plots:
                try:
                    args["ax"] = new_ax
                    self.__plot_updatably(**args)
                except TypeError as e:
                    print("In subplot " + str(i) + ":")
                    print(e)
                    print("You might want to remove the plot and try again.")
        pyplot.show(block=False)
```

```
def update_data(self, quantity):
    """Update the given quantity to match the location of the region."""
    if not self.up_to_date[quantity]:
        x, y = self.circle.center
        r = self.circle.radius
        # Find the slice of the image to bother with.
        xleft = max(int(x) - int(r+1), 0)
        xright = min(int(x) + int(r+1), self.skymap.image_size)
        ydown = max(int(y) - int(r+1), 0)
        yup = min(int(y) + int(r+1), self.skymap.image_size)
        # Create a circular mask to use in that slice.
        y_grid, x_grid = numpy.ogrid[ydown:yup, xleft:xright]
        mask = (x_grid - x)**2 + (y_grid - y)**2 <= r**2
        self.npixels = numpy.sum(mask)
        # Is everything loaded?
        load_list = self.skymap.loaded[ydown:yup, xleft:xright][mask]
        if not len(load_list) == numpy.sum(load_list):
            self.skymap.load_data(ydown - int(LOAD_FACTOR*r),
                                  yup + int(LOAD_FACTOR*r),
                                  xleft - int(LOAD_FACTOR*r),
                                  xright + int(LOAD_FACTOR*r))
        # It is now
        # Quantities
        if quantity == "F":
            self.F = numpy.sum(self.skymap.F[:, ydown:yup,
                                             xleft:xright][:, mask], axis=1)
            self.up_to_date["F"] = True
        elif quantity == "I":
            self.I = numpy.sum(self.skymap.I[:, ydown:yup,
                                             xleft:xright][:, mask], axis=1)
            self.up_to_date["I"] = True
        elif quantity == "P":
            self.P = numpy.sum(self.skymap.P[:, ydown:yup,
                                             xleft:xright][:, mask], axis=1)
            self.up_to_date["P"] = True
        # Noise
        elif quantity == "F_noise":
            re = numpy.real(self.skymap.F[:, ydown:yup,
                                          xleft:xright][:, mask])
            im = numpy.imag(self.skymap.F[:, ydown:yup,
                                          xleft:xright][:, mask])
            self.F_noise = numpy.std(re, axis=1) + 1j*numpy.std(im, axis=1)
            self.F_noise *= numpy.sqrt(self.npixels)
            self.up_to_date["F_noise"] = True
```

```
        elif quantity == "I_noise":
            self.I_noise = numpy.std(self.skymap.I[:, ydown:yup,
                                                   xleft:xright][:, mask], axis=1)
            self.I_noise *= numpy.sqrt(self.npixels)
            self.up_to_date["I_noise"] = True
        elif quantity == "P_noise":
            re = numpy.real(self.skymap.P[:, ydown:yup,
                                          xleft:xright][:, mask])
            im = numpy.imag(self.skymap.P[:, ydown:yup,
                                          xleft:xright][:, mask])
            self.P_noise = numpy.std(re, axis=1) + 1j*numpy.std(im, axis=1)
            self.P_noise *= numpy.sqrt(self.npixels)
            self.up_to_date["P_noise"] = True
        elif quantity == "MFS_I":
            self.MFS_I = numpy.sum(self.skymap.image_data[ydown:yup,
                                                          xleft:xright][:, mask])
            self.up_to_date["MFS_I"] = True
        elif quantity == "MFS_I_noise":
            self.MFS_I_noise = numpy.std(self.skymap.image_data[ydown:yup,
                                                          xleft:xright][:, mask])
            self.MFS_I_noise *= numpy.sqrt(self.npixels)
            self.up_to_date["MFS_I_noise"] = True

def plot_image_histograms(self):
    """Plot the distribution of I, Q, U, F, Fq and Fu, centered on 0."""
    # It is also possible to plot distribution of P by changing the code...
    x, y = self.circle.center
    r = self.circle.radius
    xleft = max(int(x) - int(r+1), 0)
    xright = min(int(x) + int(r+1), self.skymap.image_size)
    ydown = max(int(y) - int(r+1), 0)
    yup = min(int(y) + int(r+1), self.skymap.image_size)
    y_grid, x_grid = numpy.ogrid[ydown:yup, xleft:xright]
    mask = (x_grid - x)**2 + (y_grid - y)**2 <= r**2

    fig, ((iax, fax), (qax, uax), (fqax, fuax)) = pyplot.subplots(3, 2)
    for ax, quant in zip((iax, fax, qax, uax, fqax, fuax),
                         ("I", "F", "Q", "U", "Fq", "Fu")): # ...here.
        if quant == "I":
            data = self.skymap.I[:, ydown:yup, xleft:xright][:, mask]
        if quant == "F":
            data = numpy.absolute(self.skymap.F[:, ydown:yup,
                                                xleft:xright][:, mask])
        if quant == "Fq":
            data = numpy.real(self.skymap.F[:, ydown:yup,
                                            xleft:xright][:, mask])
```

```
        if quant == "Fu":
            data = numpy.imag(self.skymap.F[:, ydown:yup,
                                            xleft:xright][:, mask])
        if quant == "Q":
            data = numpy.real(self.skymap.P[:, ydown:yup,
                                            xleft:xright][:, mask])
        if quant == "U":
            data = numpy.imag(self.skymap.P[:, ydown:yup,
                                            xleft:xright][:, mask])
        if quant == "P":
            data = numpy.absolute(self.skymap.P[:, ydown:yup,
                                                xleft:xright][:, mask])
        data = data.swapaxes(0, 1)
        data[:, :] = data[:, :] - numpy.average(data[:, :], axis=0)
        data = data.swapaxes(0, 1)
        ax.hist(data[:, :].flatten(), 50)
        ax.get_xaxis().get_major_formatter().set_powerlimits((-3, 4))
        ax.set_xlabel(quant)
        ax.set_ylabel("pixels")
    pyplot.show(block=False)

def plot_spectrum_histograms(self, F_exclude=10):
    """Plot the distributions of I, Q, U, F, Fq and Fu, averaged in each
        channel/depth."""
    fig, ((iax, fax), (qax, uax), (fqax, fuax)) = pyplot.subplots(3, 2)
    for ax, quant in zip((iax, fax, qax, uax, fqax, fuax),
                         ("I", "F", "Q", "U", "Fq", "Fu")):
        data = self.get_quantity(quant)["data"]
        if quant in ["F", "Fq", "Fu"] and F_exclude > 0:
            data = data[numpy.absolute(self.phi) > F_exclude]
        ax.hist(data, 50)
        ax.get_xaxis().get_major_formatter().set_powerlimits((-3, 4))
        ax.set_xlabel(quant)
        ax.set_ylabel("points")
    pyplot.show(block=False)


def __update_plots(self):
    # Update all plots.
    if not self.silent:
        for plotargs in self.updatable_plots:
            self.__plot_updatably(**plotargs)
        self.fig.canvas.draw()
```

```
def export_data(self, filename, *cols):
    """Write the specified quatities as columns in a text file.

    cols are the quantities, as named in get_quantity.
    """
    header = ""
    col_list = []
    # Make the header.
    for col in cols:
        quantity = self.get_quantity(col)
        header += quantity["label"] + "[" + quantity["unit"] + "] "
        col_list.append(numpy.array(quantity["data"], ndmin=2).transpose())
    # Make the columns
    data = numpy.concatenate(col_list, axis=1)
    numpy.savetxt(filename + ".tmp", data)
    # Because my version of numpy is too old to support a header in savetxt.
    with open(filename + ".tmp", "r") as orig:
        with open(filename, "w") as new:
            new.write(header + "\n")
            for line in orig.readlines():
                new.write(line)
    os.remove(filename + ".tmp")

def spec_noise(self, quantity, F_exclude=10):
    """Get the noise and average in the spectrum of the quantity.

    Args:
        quantity: The quantity, named as in get_quantity.
        F_exclude: If the quantity is F, Fq or Fu, exclude anything
            within [-F_exclude, F_exclude] to avoid instrumental
            polarization.
    Returns:
        A tuple (std, avg) of the standard deviation and average.
    """
    data = self.get_quantity(quantity)["data"]
    if quantity in ["F", "Fq", "Fu"] and F_exclude > 0:
        data = data[numpy.absolute(self.phi) > F_exclude]
    std = numpy.std(data)
    avg = numpy.average(data)
    return (std, avg)
```

```
def im_noise(self, quantity, F_exclude=10):
    """Get the average (with respect to channel/phi) noise in the image.

    Args:
        quantity: The quantity, named as in get_quantity.
        F_exclude: If the quantity is F, Fq or Fu, exclude anything
            within [-F_exclude, F_exclude] to avoid instrumental
            polarization.
    """
    data = self.get_quantity(quantity, noise=True)["data"]
    if quantity in ["F", "Fq", "Fu"] and F_exclude > 0:
        data = data[numpy.absolute(self.phi) > F_exclude]
    return numpy.average(data)

def get_quantity(self, toplot, noise=False):
    """Get a quantity, along with some pertinent information.

    The quantity can be any of:
        "I", "P" and its components "Q" and "U".
        Lowercase versions of these four, signifying them
            having been divided by I.
        "F" and its components "Fq" and "Fu".
        "chi", the polarization angle.
        "l2", the wavength squared.
        "nu", the frequency.
        "phi", the Faraday depth.

    Args:
        toplot: The quantity.
        noise: For the supported quantities I, P, Q, U, F, Fq and Fu,
            instead return the standard deviation within the region.
    Returns:
        A dict with:
            "label": A string to use in the axis label.
            "unit": A string for the unit to use in the axis label.
            "in_blocks": Whether the quantity is in frequency blocks or not.
            "data": The quantity data.
    """
    if toplot in ["I", "p", "q", "u", "i"]:
        if noise:
            self.update_data("I_noise")
        else:
            self.update_data("I")
```

```
    if toplot in ["P", "p", "Q", "q", "U", "u", "chi"]:
        if noise:
            self.update_data("P_noise")
        else:
            self.update_data("P")
    if toplot in ["F", "Fq", "Fu"]:
        if noise:
            self.update_data("F_noise")
        else:
            self.update_data("F")

    if noise:
        P = self.P_noise
        F = self.F_noise
        I = self.I_noise
    else:
        P = self.P
        F = self.F
        I = self.I
    in_blocks = True
    if toplot == "l2":
        data = self.l2
        unit_str = "m$^2$"
        quantity_str = "$\lambda^2$"
    elif toplot == "nu":
        data = self.nu
        unit_str = "Hz"
        quantity_str = "$\\nu$"
    elif toplot == "phi":
        data = self.phi
        unit_str = "rad/m$^2$"
        quantity_str = "$\phi$"
        in_blocks = False
    elif toplot == "chi":
        if noise:
            print("WARNING: Noise for chi not implemented!"
                  + "Results are likely to be wrong!")
        data = numpy.arctan2(numpy.imag(P), numpy.real(P))*180/math.pi
        unit_str = "$^\circ$"
        quantity_str = "$\chi$"
    elif toplot == "F":
        data = numpy.absolute(F)
        unit_str = "Jy/rmsf"
        quantity_str = "|F|"
        in_blocks = False
```

```
    elif toplot == "Fq":
        data = numpy.real(F)
        unit_str = "Jy/rmsf"
        quantity_str = "Re F"
        in_blocks = False
    elif toplot == "Fu":
        data = numpy.imag(F)
        unit_str = "Jy/rmsf"
        quantity_str = "Im F"
        in_blocks = False
    elif len(toplot) == 1:
        if toplot.lower() == "p":
            data = numpy.absolute(P)
        elif toplot.lower() == "q":
            data = numpy.real(P)
        elif toplot.lower() == "u":
            data = numpy.imag(P)
        elif toplot.lower() == "i":
            data = I
        if toplot.islower():
            if noise:
                print("WARNING: Noise for p,q and u not implemented!"
                      + "Results are likely to be wrong!")
            data = data/I
            unit_str = "%"
        else:
            unit_str = "Jy"
        quantity_str = toplot
    return {"label":quantity_str, "unit":unit_str, "data":data,
            "in_blocks":in_blocks}



def __plot_updatably(self, ax, xaxis, yaxis, loglog=False, avg_block=False,
                     errorbar_x=False, errorbar_y=True, x_prefix=None,
                     y_prefix=None, animate=False, anim_increasing_nu=True,
                     square=False, lines=None, **kwargs):
    # Either create a new plot or update it if it exists.
    # Arguments:
    #     ax: The matplotlib axes object containing the plot.
    #     lines: The matplotlib line object(s) of the plots, if they have
    #         already been initialized.
    #     Other kwargs: See documentation for add_plot.
```

```
# Copy the arguments of the function, to save for updating later.
if lines == None:
    args = locals().copy()
    del args["self"]
#--------------------------------------------------------------------------#
#----------------------Check what exactly to plot--------------------------#
#--------------------------------------------------------------------------#
x_quantity = self.get_quantity(xaxis)
y_quantity = self.get_quantity(yaxis)



x = x_quantity["data"]
x_quantity_str = x_quantity["label"]
x_unit_str = x_quantity["unit"]

y = y_quantity["data"]
y_quantity_str = y_quantity["label"]
y_unit_str = y_quantity["unit"]
in_blocks = y_quantity["in_blocks"]

if avg_block:
    avgx = []
    avgy = []
    stdx = []
    stdy = []
    for i, block in enumerate(self.block_inds):
        avgx.append(numpy.average(x[slice(*block)]))
        avgy.append(numpy.average(y[slice(*block)]))
        if errorbar_x:
            stdx.append(numpy.std(x[slice(*block)]))
        else:
            stdx.append(None)
        if errorbar_y:
            stdy.append(numpy.std(y[slice(*block)]))
        else:
            stdy.append(None)
if lines == None:
#--------------------------------------------------------------------------#
#----------------If this is the first time, make the plot------------------#
#--------------------------------------------------------------------------#
    lines = []
    if in_blocks:
        rainbow = not "color" in kwargs
```

```
            for i, block in enumerate(self.block_inds):
                if rainbow:
                    kwargs["color"] = self.block_colors[i]
                if avg_block:
                    errbar = ax.errorbar(avgx[i], avgy[i], stdy[i], stdx[i],
                                         **kwargs)
                    lines.append(errbar)
                else:
                    line, = ax.plot(x[slice(*block)], y[slice(*block)],
                                    **kwargs)
                    lines.append(line)
        else:
            line, = ax.plot(x, y, label="x=" + xaxis + ",y=" + yaxis,
                            **kwargs)
            lines.append(line)
        args["lines"] = lines
        ax.grid(True)
        if loglog:
            ax.set_xscale("log")
            if x.max() < 5*x.min():
                ax.set_xlim(x.min()*0.9, x.max()*1.1)
                ax.set_xticks(numpy.linspace(x.min(), x.max(), 4))
            ax.set_yscale("log")
            if y.max() < 5*y.min():
                ax.set_ylim(y.min()*0.9, y.max()*1.1)
                ax.set_yticks(numpy.linspace(y.min(), y.max(), 4))
        scales = []
        for unit, prefix in zip((x_unit_str, y_unit_str), (x_prefix,
                                                           y_prefix)):
            if unit == "%":
                scale = 100
            else:
                scale = 1
            if not prefix == None:
                if prefix == "m":
                    scale *= 1e3
                elif prefix == "k":
                    scale *= 1e-3
                elif prefix == "M":
                    scale *= 1e-6
            scales.append(scale)
        x_scale = scales[0]
        y_scale = scales[1]
```

```
        if not x_prefix == None:
            x_unit_str = x_prefix + x_unit_str
        if not y_prefix == None:
            y_unit_str = y_prefix + y_unit_str
        formfunc_x = lambda x, pos: str(float('{0:.3g}'.format(x
                                        *x_scale))).rstrip("0").rstrip(".")
        formfunc_y = lambda y, pos: str(float('{0:.3g}'.format(y
                                        *y_scale))).rstrip("0").rstrip(".")
        ticks_x = ticker.FuncFormatter(formfunc_x)
        ax.xaxis.set_major_formatter(ticks_x)
        ticks_y = ticker.FuncFormatter(formfunc_y)
        ax.yaxis.set_major_formatter(ticks_y)

        ax.set_ylabel(y_quantity_str + " [" + y_unit_str + "]",
                      fontsize=TEXT_SIZE)
        ax.set_xlabel(x_quantity_str + " [" + x_unit_str + "]",
                      fontsize=TEXT_SIZE)
        if square:
            max_x = numpy.max(numpy.absolute(x))
            max_y = numpy.max(numpy.absolute(y))
            axlim = max(max_x, max_y)
            ax.axis([-axlim, axlim, -axlim, axlim])
            ax.set_aspect("equal", adjustable="box")
        ax.tick_params(axis="both", which="major", labelsize=TEXT_SIZE)
        if animate:
            fargs = (lines, x, y, in_blocks, anim_increasing_nu)
            frames = len(x) + ANIM_DISP_LENGTH
            ani = animation.FuncAnimation(self.fig, self.__update_animation,
                                          fargs=fargs, frames=frames,
                                          interval=ANIM_INTERVAL,
                                          blit=True)
            self.anims.append(ani)
        if LEGEND:
            ax.legend()
        self.updatable_plots.append(args)
    else:
        #-------------------------------------------------------------------#
        #----------If there already is a line, update it instead.-----------#
        #-------------------------------------------------------------------#
        if in_blocks:
            for i, block in enumerate(self.block_inds):
                if avg_block:
                    # Make the error bars move around.
                    point, caps, errs = lines[i]
```

```
                    if errorbar_x and not errorbar_y:
                        xcaps = caps
                        xerrline = errs[0]
                    if errorbar_y and not errorbar_x:
                        ycaps = caps
                        yerrline = errs[0]
                    if errorbar_x and errorbar_y:
                        xcaps = caps[0:2]
                        ycaps = caps[2:4]
                        xerrline = errs[0]
                        yerrline = errs[1]
                    yc = avgy[i] # Y center
                    xc = avgx[i] # X center
                    point.set_data(xc, yc)
                    if errorbar_x:
                        xl = avgx[i] - stdx[i] # X left
                        xr = avgx[i] + stdx[i] # X right
                        # X caps
                        xcaps[0].set_data(xl, yc)
                        xcaps[1].set_data(xr, yc)
                        # X errorbar
                        verts = numpy.array([[xl, yc], [xr, yc]])
                        xerrline.get_paths()[0].vertices = verts
                    if errorbar_y:
                        yl = avgy[i] - stdy[i] # Y lower
                        yh = avgy[i] + stdy[i] # Y upper
                        # Y caps
                        ycaps[0].set_data(xc, yl)
                        ycaps[1].set_data(xc, yh)
                        # Y errorbar
                        verts = numpy.array([[xc, yl], [xc, yh]])
                        yerrline.get_paths()[0].vertices = verts
                else:
                    # Update each block
                    lines[i].set_data(x[slice(*block)], y[slice(*block)])
        else:
            # Update the single line
            lines[0].set_data(x, y)

def __update_animation(self, i, lines, x, y, in_blocks, anim_increasing_nu):
    # This function is called each frame in an animated plot.
    # i is the frame number, lines are the animated lines,
    # x and y are the plotted quantities, in_blocks is whether the data is
    # in blocks, anim_increasing_nu determines in which direction the
    # animation runs.
```

```
        # There is some kind of bug when adding and removing plots when there
        # is an animation going.
        if not anim_increasing_nu:
            i = len(x) + ANIM_DISP_LENGTH - i
        if in_blocks:
            for j, block in enumerate(self.block_inds):
                if i >= block[0] and i <= block[1] + ANIM_DISP_LENGTH:
                    if i <= block[0] + ANIM_DISP_LENGTH:
                        xdata = x[block[0]:i]
                        ydata = y[block[0]:i]
                    elif i >= block[1]:
                        xdata = x[i - ANIM_DISP_LENGTH:block[1]]
                        ydata = y[i - ANIM_DISP_LENGTH:block[1]]
                    else:
                        xdata = x[i - ANIM_DISP_LENGTH:i]
                        ydata = y[i - ANIM_DISP_LENGTH:i]
                    lines[j].set_data(xdata, ydata)
                else:
                    lines[j].set_data([], [])
        else:
            if i <= ANIM_DISP_LENGTH:
                xdata = x[0:i]
                ydata = y[0:i]
            elif i >= len(x):
                xdata = x[i - ANIM_DISP_LENGTH:len(x)]
                ydata = y[i - ANIM_DISP_LENGTH:len(x)]
            else:
                xdata = x[i - ANIM_DISP_LENGTH:i]
                ydata = y[i - ANIM_DISP_LENGTH:i]
            lines[0].set_data(xdata, ydata)
        return lines

class Measurement(object):
    """A class for detecting and taking note of polarized sources.

    It has two regions, one for measuring I and one for F. By default they
    are at the same place and the same size.

    The intended working procedure is to move the regions around and recording
    the noise (by add_noise() or middle clicking), and then moving to the source
    and running do(), potentially with an identifier if there are several
    sources in the image. It is important that the regions are the same size(s)
    when recording the noise as when measuring the source, or the noise will
    not be accurate.
    """
```

```
def __init__(self, skymap, x=128, y=128, sigma=5):
    """Creates a new Measurement.

    Args:
        skymap: The SkyMap owning this object.
        x, y: The position of the regions.
        sigma: The signal-to-noise ratio required for a detected F peak.
    """
    self.sigma = sigma
    self.I_region = skymap.add_region(x, y, r=15, silent=True)
    self.I_region.pause()
    self.I_region.measurement = self
    self.I_region.measuring = "I"
    self.F_region = skymap.add_region(x, y, r=15)
    self.F_region.measurement = self
    self.F_region.measuring = "F"
    self.F_region.add_plot("phi_F")
    self.background_I_MFS = numpy.array([])
    self.noise_I_MFS = numpy.array([])
    self.noise_F = numpy.array([])
    self.background_F = numpy.array([])
    #Results
    self.I = None
    self.peakphis = None
    self.peakvals = None

def add_noise(self, noise):
    """Adds a noise sample.

    Args:
        noise: "F" or "I".
    """
    if noise == "I":
        print("Adding I noise")
        self.I_region.update_data("MFS_I")
        self.I_region.update_data("MFS_I_noise")
        self.background_I_MFS = numpy.append(self.background_I_MFS,
                                             self.I_region.MFS_I)
        self.noise_I_MFS = numpy.append(self.noise_I_MFS,
                                        self.I_region.MFS_I_noise)
    elif noise == "F":
        print("Adding F noise")
        self.F_region.update_data("F")
        std, avg = self.F_region.spec_noise("F")
        self.noise_F = numpy.append(self.noise_F, std)
        self.background_F = numpy.append(self.background_F, avg)
```

```
    def clear_noise(self):
    """Remove the collected noise samples."""
    self.background_I_MFS = []
    self.noise_I_MFS = []
    self.noise_F = []
    self.background_F = []

def do(self, source_id=None):
    """Examine a source after collecting noise.

    Gets I, finds F peaks, writes info to status.txt (see 'write') and
    writes the Faraday spectrum to spectrum.txt.
    """
    self.get_I()
    self.find_F_peaks()
    self.write(source_id)
    if not source_id == None:
        id_str = str(source_id) + "_"
    else:
        id_str = ""
    self.F_region.export_data(id_str + "spectrum.txt", "phi", "F")

def clear_data(self):
    """Clear all examined sources from the status file."""
    statusfile.clear_table_info()

def write(self, source_id=None):
    """Write the source to the status file.

    Writes the location and radii of the regions, F and phi for the found
    peaks, source I, S/N detection limit, and noises and backgrounds in
    F and I.
    """
    if self.I == None:
        print("You need to measure I!")
        return
    if self.peakphis == None or self.peakvals == None:
        print("You need to find peaks!")
        return
    F_noise = numpy.sqrt(numpy.average(self.noise_F**2))
    F_bg = numpy.average(self.background_F)
    I_noise = numpy.sqrt(numpy.average(self.noise_I_MFS**2))
    I_bg = numpy.average(self.background_I_MFS)
    circle = self.F_region.circle
    x, y = circle.center
    w = wcs.WCS(self.I_region.skymap.image_header)
```

```
    sky = wcs.utils.pixel_to_skycoord(x, y, w)
    radec = sky.to_string("hmsdms",precision=1)
    ra, dec = radec.split()
    F_r = circle.radius
    circle = self.I_region.circle
    I_r = circle.radius
    statusfile.write_table_info(source_id=source_id, philist=self.peakphis,
                                valuelist=self.peakvals, I=self.I,
                                F_noise=F_noise, F_bg=F_bg, I_noise=I_noise,
                                I_bg=I_bg, reg_ra=ra, reg_dec=dec, F_r=F_r,
                                I_r=I_r, sigma=self.sigma)


def find_F_peaks(self, plot=True):
    """Find peaks in the Faraday spectrum, and maybe plot them."""
    self.F_region.redraw_plots()
    data = self.F_region.get_quantity("F")["data"]
    std = numpy.sqrt(numpy.average(self.noise_F**2))
    avg = numpy.average(self.background_F)
    threshold = avg + std*self.sigma
    if plot:
        ax = self.F_region.fig.axes[0]
        # Draw line for threshold
        x = ax.get_xlim()
        y = [threshold, threshold]
        ax.plot(x, y, linestyle="--", color="k")
        self.F_region.fig.canvas.draw()
    # Find the peaks
    shifted_r = numpy.concatenate((data[:1], data[:-1]))
    shifted_l = numpy.concatenate((data[1:], data[-1:]))
    peaks = numpy.logical_and(data > shifted_r, data > shifted_l)
    peakvals = data[peaks]
    peakphis = self.F_region.get_quantity("phi")["data"][peaks]
    peakfilt = peakvals > threshold
    peakvals = peakvals[peakfilt]
    peakphis = peakphis[peakfilt]
    # Plot red lines at the peaks
    if plot:
        ymax = 0
        for peak, val in zip(peakphis, peakvals):
            if peak > 5 or peak < -5:
                if val > ymax:
                    ymax = val
                x = [peak, peak]
                y = ax.get_ylim()
                ax.plot(x, y, linestyle="--", color="r")
```

```
            if ymax != 0:
                ymax *= 2
                self.F_region.fig.axes[0].set_ylim([0, ymax])
            self.F_region.fig.canvas.draw()
        self.peakphis = peakphis
        self.peakvals = peakvals
        return peakphis, peakvals

    def get_I(self):
        """Return map I in the I region of the measurement."""
        self.I_region.update_data("MFS_I")
        I = self.I_region.MFS_I
        self.I = I
        return I

#Parse the command line
if __name__ == "__main__":
    # Create the list of command line arguments
    arguments = sys.argv[1:]

    # Use what remains as output directory
    if len(arguments) == 1:
        input_dir = arguments[0]
    elif len(arguments) == 0:
        sys.exit("Did you remember to specify the directory?")
    else:
        sys.exit("rmsynth_plot does not take any arguments but the directory.")

    main(input_dir=input_dir)
```

Analyzing examined sources (make_table.py)

```
import os
import math
import shutil
from datetime import timedelta
from datetime import datetime

import numpy
from matplotlib import pyplot
from astropy.io import fits
from astropy.coordinates import SkyCoord

import statusfile
from statusfile import read_table_info as read

RMSF_FWHM = 1 # For calculating phi error.
```

```
def make_table():
    """Analyze all the examined sources in the current directory.

    Reads the information written by rmsynth_plot to status.txt, plots
    quantities and makes a table.
    """
    rows = read_files()
    #plot_time(rows)
    #increase_sigma(rows,5)
    correct_for_bg(rows)
    add_errors(rows)
    #remove_specific(rows, "m51")
    compare_gmrt(rows)
    #remove_specific(rows ,"taylor6", "mulcahy7")
    #phi_histogram(rows)
    #peak_height(rows)
    #remove_instrumental(rows)
    #remove_peakless(rows)
    #find_nvss(rows)
    #sort_rows(rows)
    #output_table(rows)
    #output_figures(rows)


def read_files(whitelist=None):
    """Read status.txt in each subdir, return sources as a list of dicts."""
    rows = []
    dirs = os.listdir("./")
    if not whitelist == None:
        dirs = [d for d in dirs if d in whitelist]
    for d in dirs:
        if os.path.isfile(d + "/" + statusfile.STATUS_FILENAME):
            print("opening " + d)
            os.chdir(d)
            ids = statusfile.find_table_info()
            for source in ids:
                print("source " + str(source))
                row = dict()
                row["dir"] = d
                row["id"] = source
                row["ra"] = read("central RA", source)
                row["dec"] = read("central dec", source)
                phistr = read("phi",source)
                valstr = read("F",source)
```

```
row["peaks"] = []
for phi, val in zip(phistr.split(","), valstr.split(",")):
    try:
        row["peaks"].append((float(phi), float(val)))
    except ValueError:
        print("IGNORED:" + phi + "," + val)
row["I"] = float(read("I", source))
row["radius_F"] = float(read("F radius", source))
row["radius_I"] = float(read("I radius", source))
row["noise_F"] = float(read("F spectrum noise", source))
row["noise_I"] = float(read("I image noise", source))
row["bg_F"] = float(read("F spectrum background", source))
row["bg_I"] = float(read("I image background", source))
row["sigma"] = float(read("Threshold", source))
row["imsize"] = int(statusfile.read_wsclean("image size"))
row["time"] = dict()
timestr = statusfile.read_shift_and_avg("time taken")
try:
    try:
        timeobj = datetime.strptime(timestr, "%H:%M:%S.%f")
    except ValueError:
        try:
            timeobj = datetime.strptime(timestr,
                                        "%d days, %H:%M:%S.%f")
        except ValueError:
            timeobj = datetime.strptime(timestr,
                                        "%d day, %H:%M:%S.%f")
    row["time"]["phase_avg"] = timeobj
    timestr = statusfile.read_wsclean("time taken")
    try:
        timeobj = datetime.strptime(timestr, "%H:%M:%S.%f")
    except ValueError:
        try:
            timeobj = datetime.strptime(timestr,
                                        "%d days, %H:%M:%S.%f")
        except ValueError:
            timeobj = datetime.strptime(timestr,
                                        "%d day, %H:%M:%S.%f")
    row["time"]["wsclean"] = timeobj
    timestr = statusfile.read_pyrmsynth("time taken")
    try:
        timeobj = datetime.strptime(timestr, "%H:%M:%S.%f")
    except ValueError:
        try:
            timeobj = datetime.strptime(timestr,
                                        "%d days, %H:%M:%S.%f")
```

```
                    except ValueError:
                        timeobj = datetime.strptime(timestr,
                                                    "%d day, %H:%M:%S.%f")
                    row["time"]["pyrmsynth"] = timeobj
                except ValueError:
                    pass
                rows.append(row)
            os.chdir("..")
    nb_peaks = 0
    for row in rows:
        nb_peaks += len(row["peaks"])
    print("Peaks from files: "+str(nb_peaks))
    return rows

def plot_time(rows):
    """Plot a histogram of the time taken."""
    dirs = set()
    time_list = []
    for row in rows:
        if not row["imsize"] > 256 and not row["dir"] in dirs:
            dirs.add(row["dir"])
            times = row["time"]
            time_taken = timedelta()
            for t in times:
                time_taken = time_taken + times[t] - datetime(1900,1,1)
            print(row["dir"]+" "+str(time_taken))
            time_list.append(time_taken.total_seconds()/3600)
    pyplot.hist(time_list)
    pyplot.xlabel("time [h]")
    pyplot.show()

def increase_sigma(rows, sigma):
    """Increase the threshold for F peak detection."""
    for row in rows:
        threshold = row["bg_F"] + row["noise_F"]*sigma
        newpeaks = []
        for peak in row["peaks"]:
            if peak[1] > threshold:
                newpeaks.append(peak)
        row["peaks"] = newpeaks
        row["sigma"] = sigma
    nb_peaks=0
    for row in rows:
        nb_peaks += len(row["peaks"])
    print("Peaks after sigma=" + str(sigma) + ": " + str(nb_peaks))
```

```
def remove_specific(rows, *args):
    """Remove the sources from the given directories from the list."""
    for row in [r for r in rows if r["dir"] in args]:
        rows.remove(row)


def remove_instrumental(rows):
    """Remove instrumental polarization.

    Note that this is just a way to remove those peaks that I by eye interpreted
    as instrumental polarization, not a robust way to remove it in general.
    """
    inner_lim = 3
    outer_lim = 10
    for row in rows:
        if row["peaks"]:
            newpeaks = []
            # Find the largest peak, which in my case always is instrumental
            max_peak = max(row["peaks"], key=lambda x: x[1])[1]
            for peak in row["peaks"]:
                if math.fabs(peak[0]) >= inner_lim:
                    if math.fabs(peak[0]) >= outer_lim or peak[1]/max_peak>0.1:
                        newpeaks.append(peak)
            row["peaks"] = newpeaks

    nb_peaks = 0
    for row in rows:
        nb_peaks += len(row["peaks"])
    print("Peaks after removing instrumental polarization: " + str(nb_peaks))

def correct_for_bg(rows):
    """Subtract the background from I and F."""
    for row in rows:
        newpeaks = []
        for peak in row["peaks"]:
            newpeaks.append((peak[0], peak[1] - row["bg_F"]))
        row["peaks"] = newpeaks
        row["I"] -= row["bg_I"]
```

```
def add_errors(rows):
    """Calculates and adds the errors to the source intensities and peaks."""
    for row in rows:
        row["peaks_err"] = []
        for peak in row["peaks"]:
            phi_err = 0.5*RMSF_FWHM*row["noise_F"]/peak[1] # From Mulcahy
            val_err = row["noise_F"]
            row["peaks_err"].append((phi_err, val_err))
        row["I_err"] = row["noise_I"]*numpy.sqrt(numpy.pi*row["radius_I"]**2)

def remove_peakless(rows):
    """Removes those sources that do not have detected peaks."""
    for r in [r for r in rows if len(r["peaks"]) == 0]:
        rows.remove(r)


def phi_histogram(rows):
    """Plot a histogram of the peak Faraday depths."""
    phis = []
    for row in rows:
        for peak in row["peaks"]:
            phis.append(peak[0])
    pyplot.hist(phis, 800, (-100 - 0.125, 99.75 + 0.125))
    pyplot.show()

def peak_height(rows):
    """Plot peaks normalized to the main instrumental peaks of the sources."""
    phis = []
    peaks = []
    for row in rows:
        for peak in row["peaks"]:
            phis.append(peak[0])
            peaks.append(peak[1])
        peak_nb = len(row["peaks"])
        max_peak = numpy.max(peaks[-peak_nb:])
        peaks[-peak_nb:] = peaks[-peak_nb:] / max_peak
    phis = numpy.array(phis)
    peaks = numpy.array(peaks)
    # This colors the instrumental peaks red for the dataset I have, probably
    # not for others.
    discardfilt = numpy.logical_and(numpy.absolute(phis) < 10, peaks < 0.1)
    discardfilt = numpy.logical_or(discardfilt, numpy.absolute(phis) < 3)
    keepfilt = numpy.logical_not(discardfilt)
    pyplot.plot(phis[discardfilt], peaks[discardfilt], ".", color="r")
    pyplot.plot(phis[keepfilt], peaks[keepfilt], ".", color="b")
```

```
    pyplot.xlabel("$\phi$ [rad m$^{-2}$]", fontsize=16)
    pyplot.ylabel("Relative peak height", fontsize=16)
    pyplot.tick_params(axis="both", which="major", labelsize=16)
    pyplot.axis([-50, 50, 0, 1.1])
    pyplot.grid()
    pyplot.show()

def find_nvss(rows):
    """Find the closest NVSS source to each source."""
    fitsfile = fits.open("/home/anton/matlab/nvss.fits")
    name = fitsfile[1].data["NVSS"]
    ra = fitsfile[1].data["RAJ2000"]
    dec = fitsfile[1].data["DEJ2000"]
    # Filter out sources not in the field.
    rafilt = numpy.logical_and(ra > 195, ra < 210)
    decfilt = numpy.logical_and(dec > 40, dec < 55)
    filt = numpy.logical_and(rafilt, decfilt)
    name = name[filt]
    ra = ra[filt]
    dec = dec[filt]
    catalog = SkyCoord(ra, dec, unit="deg")
    names = set()
    duplicates = set()
    for row in rows:
        idx, sep2d, _ = SkyCoord(row["ra"],
                                 row["dec"]).match_to_catalog_sky(catalog)
        d, m, s = sep2d.dms
        nvss_name = "J" + name[idx]
        # Print a warning if the separation is too large.
        if d > 0 or m > 0 or s > 25:
            print("Warning: " + nvss_name + " " + str(sep2d))
            print(row["dir"])
            if not row["id"] == None:
                print(row["id"])
            print(row["ra"] + " " + row["dec"])
        row["NVSS"] = nvss_name
        if nvss_name in names:
            duplicates.add(nvss_name)
        else:
            names.add(nvss_name)
    for n in duplicates:
        # Append letter to the sources that correspond to the same NVSS source.
        suffixes = ["a","b","c"] # More could be added, of course.
        for row in rows:
            if row["NVSS"] == n:
                row["NVSS"] += suffixes.pop(0)
```

```
def compare_gmrt(rows):
    """Compare the source intensities to TGSS Alternative Data Release.

    http://tgssadr.strw.leidenuniv.nl/doku.php
    """
    fitsfile = fits.open("/home/anton/gmrt.fits")
    name = fitsfile[1].data["ID"]
    ra = fitsfile[1].data["RA"]
    dec = fitsfile[1].data["DEC"]
    I = fitsfile[1].data["Sint"]
    # Filter out sources outside the field.
    rafilt = numpy.logical_and(ra > 195, ra < 210)
    decfilt = numpy.logical_and(dec > 40, dec < 55)
    filt = numpy.logical_and(rafilt, decfilt)
    name = name[filt]
    ra = ra[filt]
    dec = dec[filt]
    I = I[filt]
    catalog = SkyCoord(ra, dec, unit = "deg")
    names = set()
    duplicates = set()
    for row in rows:
        idx, sep2d, _ = SkyCoord(row["ra"],
                                 row["dec"]).match_to_catalog_sky(catalog)
        d, m, s = sep2d.dms
        gmrt_name = name[idx]
        # Print a warning and skip the source if the distance is too large.
        if d > 0 or m > 0 or s > 25:
            print("Warning: " + gmrt_name + " " + str(sep2d))
            print(row["dir"])
            if not row["id"] == None:
                print(row["id"])
            print(row["ra"] + " " + row["dec"])
            continue
        row["GMRT_I"] = I[idx]
        row["GMRT_name"] = gmrt_name
        if gmrt_name in names:
            duplicates.add(gmrt_name)
            print("duplicate " + gmrt_name)
        else:
            names.add(gmrt_name)
    lofar_I = []
    gmrt_I = []
```

```
    for row in rows:
        if "GMRT_name" in row and not row["GMRT_name"] in duplicates:
            lofar_I.append(row["I"] * 1000)
            gmrt_I.append(row["GMRT_I"])
    # Add together any sources corresponding to the same GMRT source.
    for name in duplicates:
        lof = 0
        gmr = None
        for row in rows:
            if "GMRT_name" in row and row["GMRT_name"] == name:
                lof += row["I"]*1000
                gmr = row["GMRT_I"]
        lofar_I.append(lof)
        gmrt_I.append(gmr)

    coeffs = numpy.polyfit(gmrt_I, lofar_I, 1)
    print("Coefficients")
    print(coeffs)

    pyplot.plot(gmrt_I, lofar_I, ".")
    pyplot.xlabel("GMRT flux [mJy]", fontsize=16)
    pyplot.ylabel("LOFAR flux [mJy]", fontsize=16)
    pyplot.grid(True)
    x = numpy.array([0, 10000])
    y = x*coeffs[0] + coeffs[1]
    #pyplot.plot(x, y)
    x = numpy.array([0, 10000])
    y = x
    pyplot.plot(x, y, linestyle = "--", color="k")
    pyplot.axis([0, 10000, 0, 10000])
    pyplot.tick_params(axis="both", which="major", labelsize=16)
    pyplot.show()

def sort_rows(rows):
    """Sort the sources by NVSS name."""
    rows.sort(key=lambda k: k["NVSS"])

def output_table(rows, all_peaks=False):
    """Make a LaTeX table of the sources and write it to generated_table.txt."""
    strings = []
    strings.append(r"\begin{tabular}{|c|c|c|c|c|c|} \hline")
    strings.append(r"{\bf Name} & {\bf RA (J2000)} & {\bf Dec (J2000)} & "
                   + r"{\bf I [mJy]} & {\bf $\phi$ [rad m$^{-2}$]} & "
                   + r"{\bf PI [$\unit{\mu Jy}$]} \\ \hline")
```

```
for row in rows:
    peaks = row["peaks"]
    if len(peaks) > 3 and not all_peaks:
        peaks = [None] # Will print as 'Complex' instead of including all.
    for i, peak in enumerate(peaks):
        peak_err = row["peaks_err"][i]
        if i == 0:
            nvss = row["NVSS"]
            ra = row["ra"]
            dec = row["dec"]
            # These type conversions are a hack to not use scientific
            # notation for sources stronger than 1000 mJy.
            I = ("$ " + str(int(float(format(1000*row["I"], ".2g"))))
                + r" \pm " + str(float(format(1000*row["I_err"], ".2g")))
                + " $")
            if not row["id"] == None:
                idstr = row["id"]
            else:
                idstr = ""
            comment = " %" + row["dir"] + "," + idstr
        else: # Do not print duplicate information for the additional peaks.
            nvss = ""
            ra = ""
            dec = ""
            I = ""
            comment = ""
        if not peak == None:
            phi = "$ " + str(peak[0]) + r" \pm " + str(peak_err[0]) + " $"
            F = ("$ " + str(int(float(format(1000000*peak[1], ".2g"))))
    + r" \pm " + str(int(float(format(1000000*peak_err[1], ".2g")))) + " $")
        else:
            phi = "Complex"
            F = "Complex"
        if i == len(peaks) - 1:
            hline = r" \hline" # A line between the sources.
        else:
            hline = ""
        strings.append(nvss + " & " + ra + " & " + dec + " & " + I + " & "
                       + phi + " & " + F + r" \\" + hline + comment)

strings.append(r"\end{tabular}")
# Add line breaks to each line.
for i, s in enumerate(strings):
    strings[i] = s + "\n"
with open("generated_table.txt", "w") as f:
    f.writelines(strings)
```

```
def output_figures(rows):
    """Show Faraday spectra of the sources."""
    for row in rows:
        if row["id"] == None:
            idstr = ""
        else:
            idstr = row["id"] + "_"
        print(row["dir"] + "," + idstr)
        spectrum_file = row["dir"] + "/" + idstr + "spectrum.txt"
        spectrum_data = numpy.loadtxt(spectrum_file, skiprows=1)
        threshold = row["bg_F"] + row["noise_F"]*row["sigma"]
        fig = pyplot.figure()
        ax = fig.add_subplot(111)
        ax.plot(spectrum_data[:,0], spectrum_data[:,1], color="k")
        x = ax.get_xlim()
        y = [threshold, threshold]
        ax.plot(x, y, linestyle="--", color="k")
        # We don't want to see the entire instrumental peak.
        ymax = 0
        for phi, peak in row["peaks"]:
            if peak > ymax:
                ymax = peak
            x = [phi, phi]
            y = ax.get_ylim()
            ax.plot(x, y, linestyle="--", color="r")
        if ymax != 0:
            ymax *= 2
            ax.set_ylim([0, ymax])
        pyplot.show()

if __name__=="__main__":
    make_table()
```

Default pyrmsynth parameter file (default_params.par)

```
% parameter file for rmsynthesis python code
% capable of running standard RM Synthesis as well as RM Clean

% Parameter file format:
% Comments can be added on their own lines, starting with a %, this must be
% followed by a space
% Parameters are given as keyword value pairs, with spaces as delimiters

% ra and dec min and max of the subimage to process, given in pixels
% a value of -1 means to use the bound of the image
dec_min -1
dec_max -1
ra_min -1
ra_max -1

% Define the phi axis, dphi in rad/m/m
phi_min -100
nphi 200
dphi 1

% Mask file. Pixels with non-zero values in the image will be used for
% RM Synthesis
% Comment the following line out if you don't wish to use a mask
% Mask image must have the same number of pixels as the Stokes-Q and U images
% irrespective of the ra, dec min and max values.
%imagemask ./mask.fits

% Clean parameters.  gain is the loop gain, niter is the number of
% clean iterations, cutoff sets the value of the max of the residual image
% at which point the procedure stops, defined in Jy
do_clean True
gain 0.1
niter 100
cutoff 0.0001

% weighting parameter. Give the name of the weight
% file (located in the input_dir).
% If you leave it out, all channels will be given a weight of 1.0.
%do_weight weight.txt
```

```
% spectral index option. Give directory or global average value. If wanted,
% specify reference frequency.
% alpha 0
%ref_freq

% Detection threshold on polarized intensity map
threshold 0.1

% output file
outputfn ./

% directory where the input fits file can be found
input_dir ./
```

Default fsclean parameter file (default_fsclean.parset)

```
nphi = 400
nra = 256
ndec = 256
cellsize = 1.0
dphi = 0.25
gain = 0.1
niter = 2400
cutoff = 0
bmaj = 0
bmin = 0
bpa = 0
bphi = 0
verbosity = 1
clean_type = 1
```

Bibliography

Bell, M. R. and T. A. Ensslin (2015). *fsclean: Faraday Synthesis CLEAN imager*. Astrophysics Source Code Library. ascl: 1506.006.

Bell, M. R. and T. A. Enßlin (2012). "Faraday synthesis. The synergy of aperture and rotation measure synthesis". In: A&A 540, A80, A80. DOI: 10.1051/0004-6361/201118672. arXiv: 1112.4175 [astro-ph.IM].

Bernet, M. L., F. Miniati, and S. J. Lilly (2010). "Mg II Absorption Systems with W $_0$ >= 0.1 Å for a Radio Selected Sample of 77 Quasi-Stellar Objects and their Associated Magnetic Fields at High Redshift". In: ApJ 711, pp. 380–388. DOI: 10.1088/0004-637X/711/1/380. arXiv: 1001.2311.

– (2012). "The Interpretation of Rotation Measures in the Presence of Inhomogeneous Foreground Screens". In: ApJ 761, 144, p. 144. DOI: 10.1088/0004-637X/761/2/144. arXiv: 1209.2410.

– (2013). "The Extent of Magnetic Fields around Galaxies out to z ~ 1". In: ApJ 772, L28, p. L28. DOI: 10.1088/2041-8205/772/2/L28. arXiv: 1307.2250.

Bernet, M. L., F. Miniati, S. J. Lilly, et al. (2008). "Strong magnetic fields in normal galaxies at high redshift". In: Nature 454, pp. 302–304. DOI: 10.1038/nature07105. arXiv: 0807.3347.

Bonafede, A. et al. (2011). "Fractional polarization as a probe of magnetic fields in the intra-cluster medium". In: A&A 530, A24, A24. DOI: 10.1051/0004-6361/201016298. arXiv: 1103.0277.

Brentjens, M. A. and A. G. de Bruyn (2005). "Faraday rotation measure synthesis". In: A&A 441, pp. 1217–1228. DOI: 10.1051/0004-6361:20052990. eprint: astro-ph/0507349.

Burn, B. J. (1966). "On the depolarization of discrete radio sources by Faraday dispersion". In: MNRAS 133, p. 67. DOI: 10.1093/mnras/133.1.67.

Chen, H.-W. et al. (2010). "What Determines the Incidence and Extent of Mg II Absorbing Gas Around Galaxies?" In: ApJ 724, pp. L176–L182. DOI: 10.1088/2041-8205/724/2/L176. arXiv: 1011.0735.

Clark, B. G. (1980). "An efficient implementation of the algorithm 'CLEAN'". In: A&A 89, p. 377.

Condon, J. J. et al. (1998). "The NRAO VLA Sky Survey". In: AJ 115, pp. 1693–1716. DOI: 10.1086/300337.

de Vaucouleurs, G. et al. (1995). "VizieR Online Data Catalog: Third Reference Cat. of Bright Galaxies (RC3) (de Vaucouleurs+ 1991)". In: *VizieR Online Data Catalog* 7155.

Farnes, J. S., D. A. Green, and N. G. Kantharia (2013). "Probing Magnetic Fields using the Giant Metrewave Radio Telescope". In: *ArXiv e-prints*. arXiv: 1309.4646 [astro-ph.IM].

Farnes, J. S., S. P. O'Sullivan, et al. (2014). "Faraday Rotation from Magnesium II Absorbers toward Polarized Background Radio Sources". In: ApJ 795, 63, p. 63. DOI: 10.1088/0004-637X/795/1/63. arXiv: 1406.2526.

Fomalont, E. B. et al. (1989). "Depolarization silhouettes and the filamentary structure in the radio source Fornax A". In: ApJ 346, pp. L17–L20. DOI: 10.1086/185568.

Helfand, D. J., R. L. White, and R. H. Becker (2015). "The Last of FIRST: The Final Catalog and Source Identifications". In: ApJ 801, 26, p. 26. DOI: 10.1088/0004-637X/801/1/26. arXiv: 1501.01555.

Helou, G. et al. (1991). "The NASA/IPAC extragalactic database." In: *Databases and On-line Data in Astronomy*. Ed. by M. A. Albrecht and D. Egret. Vol. 171. Astrophysics and Space Science Library, pp. 89–106. DOI: 10.1007/978-94-011-3250-3_10.

Hernández-Monteagudo, C. et al. (2015). "Evidence of the Missing Baryons from the Kinematic Sunyaev-Zeldovich Effect in Planck Data". In: *Physical Review Letters* 115.19, 191301, p. 191301. DOI: 10.1103/PhysRevLett.115.191301. arXiv: 1504.04011.

Högbom, J. A. (1974). "Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines". In: A&AS 15, p. 417.

Horellou, C. et al. (2001). "Atomic and molecular gas in the merger galaxy NGC 1316 (Fornax A) and its environment". In: A&A 376, pp. 837–852. DOI: 10.1051/0004-6361:20011039. eprint: astro-ph/0107390.

Intema, H. T. et al. (2016). "The GMRT 150 MHz All-sky Radio Survey: First Alternative Data Release TGSS ADR1". In: *ArXiv e-prints*. arXiv: 1603.04368.

Kennicutt Jr., R. C. et al. (2003). "SINGS: The SIRTF Nearby Galaxies Survey". In: PASP 115, pp. 928–952. DOI: 10.1086/376941. eprint: astro-ph/0305437.

Klein, Ulrich and Andrew Fletcher (2015). *Galactic and Intergalactic Magnetic Fields*. Springer. ISBN: 9783319089423.

Lehner, N., J. C. Howk, and B. P. Wakker (2015). "Evidence for a Massive, Extended Circumgalactic Medium Around the Andromeda Galaxy". In: ApJ 804, 79, p. 79. DOI: 10.1088/0004-637X/804/2/79. arXiv: 1404.6540.

Liang, C. J. and H.-W. Chen (2014). "Mining circumgalactic baryons in the low-redshift universe". In: MNRAS 445, pp. 2061–2081. DOI: 10.1093/mnras/stu1901. arXiv: 1402.3602.

Mao, S. A. et al. (2015). "Properties of the Magneto-ionic Medium in the Halo of M51 Revealed by Wide-band Polarimetry". In: ApJ 800, 92, p. 92. DOI: 10.1088/0004-637X/800/2/92. arXiv: 1412.8320.

Mulcahy, D. D. et al. (2014). "The nature of the low-frequency emission of M 51. First observations of a nearby galaxy with LOFAR". In: A&A 568, A74, A74. DOI: 10.1051/0004-6361/201424187. arXiv: 1407.1312.

Muñoz-Mateos, J. C. et al. (2009). "Radial Distribution of Stars, Gas, and Dust in SINGS Galaxies. I. Surface Photometry and Morphology". In: ApJ 703, 1569-1596, pp. 1569–1596. DOI: 10.1088/0004-637X/703/2/1569. arXiv: 0909.2648.

Ofek, E. O. (2014). *MATLAB package for astronomy and astrophysics*. Astrophysics Source Code Library. ascl: 1407.005.

Offringa, A. R. et al. (2014). "WSCLEAN: an implementation of a fast, generic wide-field imager for radio astronomy". In: MNRAS 444, pp. 606–619. DOI: 10.1093/mnras/stu1368. arXiv: 1407.1943 [astro-ph.IM].

Oppermann, N. et al. (2015). "Estimating extragalactic Faraday rotation". In: A&A 575, A118, A118. DOI: 10.1051/0004-6361/201423995. arXiv: 1404.3701 [astro-ph.IM].

Schulman, E. and E. B. Fomalont (1992). "The distribution of ionized gas in the Sc-galaxy NGC 1310". In: AJ 103, pp. 1138–1145. DOI: 10.1086/116131.

Schwab, F. R. (1984). "Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry". In: AJ 89, pp. 1076–1081. DOI: 10.1086/113605.

Sotomayor-Beltran, C. et al. (2013). "Calibrating high-precision Faraday rotation measurements for LOFAR and the next generation of low-frequency radio telescopes". In: A&A 552, A58, A58. DOI: 10.1051/0004-6361/201220728. arXiv: 1303.6230 [astro-ph.IM].

Taylor, A. R., J. M. Stil, and C. Sunstrum (2009). "A Rotation Measure Image of the Sky". In: ApJ 702, pp. 1230–1236. DOI: 10.1088/0004-637X/702/2/1230.

van Haarlem, M. P. et al. (2013). "LOFAR: The LOw-Frequency ARray". In: A&A 556, A2, A2. DOI: 10.1051/0004-6361/201220873. arXiv: 1305.3550 [astro-ph.IM].

Werk, J. K. et al. (2014). "The COS-Halos Survey: Physical Conditions and Baryonic Mass in the Low-redshift Circumgalactic Medium". In: ApJ 792, 8, p. 8. DOI: 10.1088/0004-637X/792/1/8. arXiv: 1403.0947.

Wilson, Thomas L., Kristen Rohlfs, and Susanne Hüttemeister (2009). *Tools of radio astronomy*. 5th ed. Berlin: Springer. ISBN: 9783540851219 (cased).