Probabilistic inversion of satellite magnetic data using geostatistical simulation in spherical geometry

Master's Thesis

Mikkel Otzen
February 2019

Supervisor:
Chris Finlay

DTU Space
National Space Institute

Abstract

In this thesis, steps are taken toward better separation of geomagnetic field sources at the core and lithosphere, by accounting in new ways for the available prior information while modelling satellite magnetic data. Probabilistic inversion is carried out based on a forward scheme using Green's functions for Laplace's equation in spherical geometry with Neumann boundary conditions. Inversion is accomplished through direct sequential simulation based on ordinary Kriging, for a source surface defined by an approximate equal area grid. Prior information is implemented through semi-variogram analysis and generation of local conditional probability distributions at the source surface. The prior information used, consists of training images of the core mantle boundary field from core dynamo simulations, and for the lithosphere field, from models of remanent magnetization of the oceans in combination with full Earth models of induced magnetization. Stochastic prior realizations of the core mantle boundary and lithosphere field, which converge toward the target statistics, are achieved through spherical direct sequential simulation. In addition, probabilistic and regular inversion using synthetic and Swarm satellite observations have been attempted at the core mantle boundary. In this case, it is possible to derive smooth least squares solutions, as well as posterior realizations which have a mean that converges toward fitting the observations. It is found that posterior realizations can be generated through the use of observations with less than global coverage, by way of an approximate global coverage method. This allows for faster computations. However, results indicate that better posterior realizations are found when using global observations, showing the approximate global coverage method, as being a poor approximation in this implementation of the geomagnetic vector field description using Green's functions. Lack of posterior convergence to observation fit shows that longer simulations should be carried out, but the current results looks promising in this regard. Finally, the developed systems offer possibilities for including prior information from more than one source, and possible expansion of the estimation to two simultaneous estimation locations. Such implementations may open the door to new source separation techniques in the future.

Contents

Chapter 1

Introduction

Earth is permeated by a magnetic field. The main field is generated by dynamo action in Earth's core, while other sources include electrical currents in the magnetosphere and ionosphere, magnetized rocks in the crust, and induced electrical currents in the oceans and mantle. The electrical currents in the outer atmosphere are considered distinctly external, with the collective rest considered internal. For centuries, scientists have measured the sum of all field sources directly through Earth-bound methods, before culminating with measurements of global coverage during the age of spaceflight, and hence, satellites. Despite differences in available technology, a common trait throughout the time of geomagnetic measurements, is that those with the knowledge at hand seek to explain them. A main tool in this endeavour has been global models of Earth's magnetic field (see e.g. Hulot et al., 2015), first developed by Carl Friedrich Gauss around 1830 and leading up to modern results such as the global-research-driven International Geomagnetic Reference Field (IGRF, Thébault et al., 2015). The IGRF is a continuously updated (semi-decennial) global field model of the main field and its secular variation. It is derived from proposals of sophisticated models such as CHAOS-6 from Finlay et al. (2016).
Parallel to the development of global field models, is the understanding of core dynamics. For the majority of geomagnetic research history, these two concepts have not been tightly linked (see e.g. Olson, 2015). Only late in the twentieth century, did understanding reach a level that could explain dynamo action in Earth's core as being the main source of the geomagnetic field. Since then, and with increasing computational power, developments in core dynamo simulation continue to improve. Modern numerical simulation efforts (Aubert, 2017) now allow the generation of highly resolved training images of the core field. In addition, these numerical simulation results can be expressed in the same mathematical form as observation-based global models of Earth's magnetic field.

1.1 Global models of Earth's magnetic field

The description of Earth's global magnetic field mainly used today is still based on Gauss' work of spherical harmonic analysis (SHA). I here give a brief description following Kono (2015) and Geomagnetism lectures at Technical University of Denmark. In order to arrive at the description, two assumptions are made. The quasi-static approximation and zero electrical current density at observation height. The two approximations allow Ampère's law (with Maxwell addition) as seen in equation 1.1.1, to go to zero. Note which assumption affect each part of the equation.

$$\nabla \times \boldsymbol{B} = \underbrace{\mu_0 \boldsymbol{J}}_{\text{zero current}} + \underbrace{\mu_0 \epsilon_0 \frac{\partial}{\partial t} \boldsymbol{E}}_{\text{quasi-static}} \approx 0 \tag{1.1.1}$$

Here I denote $\boldsymbol{B}$ as the magnetic field vector (also known as magnetic flux density), $\boldsymbol{J}$ as electric current density, μ_0 is magnetic permeability in vacuum, ϵ_0 is vacuum permittivity, and $\boldsymbol{E}$ is the electric field vector. Note that strictly, $\boldsymbol{B} = \mu_r \mu_0 \boldsymbol{H}$, where μ_r is relative permeability, and $\boldsymbol{H}$ is magnetic field intensity. Geomagnetic observations measure $\boldsymbol{H}$, not $\boldsymbol{B}$, but in the atmosphere, relative permeability is approximately one, leading to a simple relation between measurements and magnetic flux density. This

has led to $\boldsymbol{B}$ being referred to as the "magnetic field vector".
The validity of the assumptions as applied to the global geomagnetic field descriptions are generally accepted. The quasi-static approximation depends on the field changing sufficiently slowly, and this being the case can be shown through scale analysis. The current free region is an approximation, the ionosphere at satellite altitude is known to have small electrical currents as mentioned. However, accepting the assumptions leading to a curl-free magnetic field vector, it is now possible to describe the global magnetic field through a scalar potential, V, since the curl of the gradient of a (twice-differentiable) scalar field is always zero.

$$\nabla \times \boldsymbol{B} = \nabla \times (-\nabla V) = 0 \quad \rightarrow \quad \boldsymbol{B} = -\nabla V \tag{1.1.2}$$

Here the negative sign is convention. This relation between magnetic field vector and scalar potential can be taken further through Gauss' law for magnetism, leading to Laplace's equation.

$$\nabla \cdot \boldsymbol{B} = \nabla \cdot -\nabla V = 0 \quad \rightarrow \quad \nabla^2 V = 0 \tag{1.1.3}$$

Which has a solution through spherical harmonic expansion. In spherical coordinates it is of the form given in equation 1.1.4.

$$\boxed{\begin{aligned} V(r,\theta,\phi) = a \sum_{n=1}^{\infty} \sum_{m=0}^{n} \Bigg\{ & \overbrace{\left[g_n^m \cos m\phi + h_n^m \sin m\phi \right] \left(\frac{a}{r} \right)^{n+1}}^{\text{internal field}} \\ & + \underbrace{\left[q_n^m \cos m\phi + s_n^m \sin m\phi \right] \left(\frac{r}{a} \right)^{n}}_{\text{external field}} \Bigg\} P_n^m(\cos\theta) \end{aligned}} \tag{1.1.4}$$

This description of the scalar potential field is given as a function of radius, r, co-latitude, θ, and longitude, ϕ. In addition, a is a reference radius, usually chosen at Earth's surface. The collection of g_n^m, h_n^m, q_n^m, and s_n^m are Gauss coefficients of degree n and order m, and finally P_n^m are Schmidt-normalized associated Legendre functions also of degree n and order m. Spherical harmonic analysis like this allow upward and downward continuation. E.g. a model estimated from observations at satellite altitude, may infer information about the geomagnetic field at the core mantle boundary (CMB) or Earth's surface. Note the association with internal field sources and external field sources as dictated by the relative radii fractions. Finally, this field description may be arranged as a system of linear equations, with the Gauss coefficients as model parameters. Using inversion techniques it is thus possible to determine a global model of Earth's magnetic field.

1.2 Source separation: core and lithosphere

As shown in equation 1.1.4, separation of the internal and external field is well described by spherical harmonic analysis. What is not as well defined, but indeed a well known problem in geomagnetism, is separation of the sources that contribute to these fields. The lithosphere and core field separation problem is usually shown through the squared magnetic intensity, $W_n(r) = \langle |\boldsymbol{B}|^2 \rangle$, by looking at internal sources only. This is defined through the Lowes-Mauersberger spectrum seen in equation 1.2.1.

$$W_n(r) = (n+1) \left(\frac{a}{r} \right)^{2n+4} \sum_{m=0}^{n} \left\{ (g_n^m)^2 + (h_n^m)^2 \right\} \tag{1.2.1}$$

Which can be computed at some radius, r, for each degree in the spherical harmonic model. A spectrum of this kind is shown in figure 1.2.1. Here it is computed for three global field models, CM4, CHAOS-4,

and a lithosphere field model, MF7. Note the sharp change in power spectrum around degree 14. This is taken to be the point at which field source dominance in SH models change, specifically a change from the core to lithosphere field. As such, this overlap inhibits us from seeing the core field above degree 14 and the lithosphere field below degree 14. In practical terms, due to the nature of spherical harmonic functions, this means that we can't resolve large scale features of the lithosphere field, and small scale features of the core field.

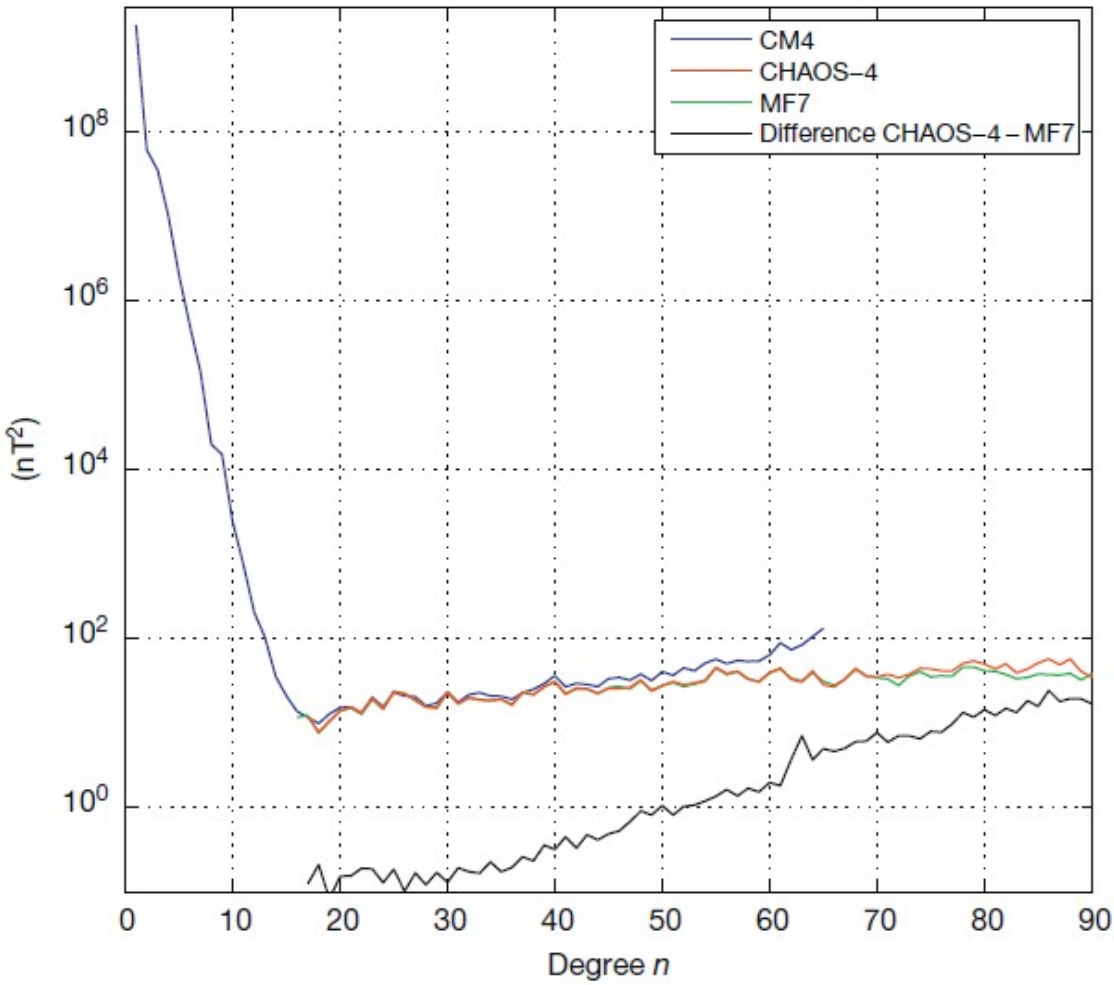


Figure 1.2.1: Spatial power spectrum of the field of internal origin at Earth's surface, as described by field models CM4, CHAOS-4, and the lithosphere field model, MF7. From Hulot et al. (2015).

1.3 A probabilistic approach

This thesis aims at working toward achieving better core and lithosphere separation. With increasing capabilities of simulations based on core dynamics, it is thought that including such information may lead to good alternative methods. Especially if it is possible to include prior information about the core and crust at the same time, in addition to satellite observations. In this regard, the direct sequential simulation Markov-Chain Monte Carlo methods used in the Matlab toolbox SIPPI (Hansen and Mosegaard, 2013a,b) was found to be interesting. Here, two-point statistical information from training images are used to invert systems of linear equations in Cartesian geometry. Inspired by this implementation, as well as the underlying GSLIB library (Deutsch and Journel, 1998), this thesis is based on my development of a geostatistical simulation tool, suitable for spherical geometry. The tool incorporates two-point statistical conditioning of training images, to do probabilistic inversion of a system of linear equations describing the geomagnetic vector field through direct sequential simulation. Something which hasn't been attempted before.

Chapter 2

Theory

This section describes the underlying theory behind the inverse problem to be solved through direct sequential simulation. The forward problem under consideration is derived from the same physical considerations used to set up the spherical harmonic expansion description of the geomagnetic scalar potential outlined in the introduction. These physical considerations lead to Laplace's equation, and where spherical harmonic functions is a conventional solution to the homogenous equation, another solution is found by subjecting Laplace's equation to the use of Green's second identity with Neumann type inhomogenous boundary conditions. As solution I use the work of Hammer (2018), in which a thorough derivation of a Green's function description of the geomagnetic vector field can be found.

In addition, I describe the relevant theory in regards to generating realizations of a random variable on a defined grid through the two-point statistical method of Kriging based direct sequential simulation. In this endeavour I follow established geostatistical work such as Deutsch and Journel (1998), Oz and Xie (2003), and Hansen and Mosegaard (2008).

2.1 The forward problem through Green's functions

2.1.1 Green's function description of the geomagnetic vector field

The wanted Green's function solution to the governing equation of the geomagnetic field potential is found in Hammer (2018) chapter 5. The solution is given by equation 2.1.1.

$$B_k(\boldsymbol{r}) = \oint_S G_k(\boldsymbol{r}, \boldsymbol{r'}) B_r(\boldsymbol{r'}) dS = \int_0^{2\pi} \int_0^{\pi} G_k(\boldsymbol{r}, \boldsymbol{r'}) B_r(\boldsymbol{r'}) \sin\theta' d\theta' d\phi' \tag{2.1.1}$$

This is the forward problem under consideration. Specifically, it is the exterior Green's function description linking the geomagnetic vector field at an observation location, $\boldsymbol{r}$, to the radial field at a source location, $\boldsymbol{r'}$, below the observation location. The full solution includes an interior Green's function term, accounting for external sources, as well as secular variation of the sources. These have been omitted, as only internal sources and observations separated in time below the shortest secular variation, are considered. For equation 2.1.1, $k = r, \theta, \phi$ are the vector field components, $G_k(\boldsymbol{r}|\boldsymbol{r'})$ is the exterior Green's function, and $B_r(\boldsymbol{r'})$ is the radial field at source surface $dS = \sin\theta' d\theta' d\phi'$. The exterior Green's functions are defined by equations 2.1.2-2.1.4.

$$G_r = \frac{1}{4\pi} \frac{h^2(1-h^2)}{f^3} \tag{2.1.2}$$

$$G_\theta = -\frac{1}{r} \frac{\partial N_i}{\partial \mu} [\cos\theta \sin\theta' \cos(\phi - \phi') - \sin\theta \cos\theta'] \tag{2.1.3}$$

$$G_\phi = \frac{1}{r} \frac{\partial N_i}{\partial \mu} [\sin\theta' \sin(\phi - \phi')] \tag{2.1.4}$$

Where the expressions in equations 2.1.5-2.1.7 are defined for simplicity of appearance in the above equations.

$$\frac{1}{r}\frac{\partial N_i}{\partial \mu} = \frac{h}{4\pi}\left[\frac{1-2h\mu+3h^2}{f^3} + \frac{\mu}{f(f+h-\mu)} - \frac{1}{1-\mu}\right] \tag{2.1.5}$$

$$h = \frac{r'}{r}, \quad f = \frac{R}{r}, \quad R = \sqrt{r^2 + r'^2 - 2rr'\mu} \tag{2.1.6}$$

$$\mu = \cos\theta\cos\theta'\sin\theta\sin\theta'\cos(\phi-\phi') \tag{2.1.7}$$

Here r, θ, and ϕ are the geographic spherical polar coordinates of the observation location, and r', θ', ϕ' are the source locations.

2.1.2 Discretization of the forward problem

Integral equation 2.1.1 can be approximated by equation 2.1.8 over a collection of discrete N_S source grid points, defined on the source surface dS. Naturally, it follows that the approximation accuracy increases with increasing grid size/resolution.

$$B_k(\boldsymbol{r}) \approx \sum_{m=1}^{N_S} G_k(\boldsymbol{r}, \boldsymbol{r'_m}) B_r(\boldsymbol{r'_m}) \sin\theta'_m \Delta\theta'_m \Delta\phi'_m \tag{2.1.8}$$

Here $\Delta\theta'_m$ and $\Delta\phi'_m$ are discretizations of the differentials $d\theta'$ and $d\phi'$, the product of which can be seen as the integration area. Performing this approximation requires knowledge of source grid parameters, such that a definition of $\Delta\theta'_m\Delta\phi'_m$ is possible for each grid location, $\boldsymbol{r'_m}$, and that the sum of each approximate integration area approaches the total area, the product of the integration limits. This is described by equation 2.1.9.

$$\sum_{m=1}^{N_S} \Delta\theta'_m \Delta\phi'_m = 2\pi^2 \tag{2.1.9}$$

If this is done correctly such that the condition is upheld reasonably, the expression can now be used to equate the magnetic field components at N_{obs} satellite observation locations, to a matrix kernel associated with observation and source geometry, and the radial field at N_S source locations of e.g. the core mantle boundary or Earth's surface.

2.2 The inverse problem through ordinary Kriging

2.2.1 Ordinary Kriging

For each radial field source target location, $\boldsymbol{r_t}$, Ordinary Kriging can be used to find the best linear unbiased estimate of the random variable at said target location. This is accomplished by considering available known realizations of the random variable. In this case a target source variable, $B_r(\boldsymbol{r'_t})$, subject to observations, $B_k(\boldsymbol{r_i})$. Consider the linear system as given in equation 2.2.1, depending only on observation values and the weighting factors known as the Kriging weights, ω_i.

$$\hat{B}_r(\boldsymbol{r'_t}) = \sum_{i=1}^{N_{obs}} \omega_i B_k(\boldsymbol{r_i}) \tag{2.2.1}$$

In order to ensure exact estimation, the difference between the expected value of the estimator, $E\{\hat{B}_r(\boldsymbol{r'_t})\}$, and the estimated parameter, $E\{B_r(\boldsymbol{r'_t})\}$, should be zero. Writing out the difference leads to equation 2.2.2, following lecture notes by Nielsen (2004) in conjunction with Journel and Huijbregts (1978).

$$
\begin{aligned}
E\Big\{B_r(\boldsymbol{r_t'}) - \hat{B}_r(\boldsymbol{r_t'})\Big\} &= E\Big\{B_r(\boldsymbol{r_t'}) - \sum_{i=1}^{N_{obs}} \omega_i B_k(\boldsymbol{r_i})\Big\} \\
E\Big\{B_r(\boldsymbol{r_t'})\Big\} - \sum_{i=1}^{N_{obs}} \omega_i E\Big\{B_k(\boldsymbol{r_i})\Big\} &= 0 \\
E\Big\{B_r(\boldsymbol{r_t'})\Big\} &= \sum_{i=1}^{N_{obs}} \omega_i E\Big\{B_k(\boldsymbol{r_i})\Big\} \qquad (2.2.2)
\end{aligned}
$$

In order to ensure minimum difference in expected values, it is clear that the Kriging weights should sum to one for all radial field source target locations, $\boldsymbol{r_t}$. This is the required condition for the estimator to be unbiased. Equation 2.2.3 expresses this unbiasedness condition as a sum, and vectorized with $\mathbf{1}$ as a vector of ones.

$$
\sum_{i=1}^{N_{obs}} \omega_i = \mathbf{1}^T \boldsymbol{\omega} = 1 \qquad (2.2.3)
$$

In addition to being unbiased, Kriging minimizes estimation variance. A useful expression arises from vectorization and substitution of the linear system in equation 2.2.1, into the following general expression for estimation variance.

$$
\begin{aligned}
\sigma_{est}^2 &= E\Big\{\Big(B_r(\boldsymbol{r_t'}) - \hat{B}_r(\boldsymbol{r_t'})\Big)^2\Big\} = V\Big\{B_r(\boldsymbol{r_t'}) - \hat{B}_r(\boldsymbol{r_t'})\Big\} \\
&= V\Big\{B_r(\boldsymbol{r_t'}) - \boldsymbol{\omega B_k(r)}\Big\} \\
&= V\big\{B_r(\boldsymbol{r_t'})\big\} + \boldsymbol{\omega}^T C\big\{\boldsymbol{B_k(r)}\big\}\boldsymbol{\omega} - 2\boldsymbol{\omega}^T C\big\{B_r(\boldsymbol{r_t'}), \boldsymbol{B_k(r)}\big\}
\end{aligned}
$$

For simplicity, I write the variance and covariances above as seen in equation 2.2.4.

$$
\sigma_{est}^2 = \sigma_{exp}^2 + \boldsymbol{\omega}^T \boldsymbol{C} \boldsymbol{\omega} - 2\boldsymbol{\omega}^T \boldsymbol{c} \qquad (2.2.4)
$$

Where σ_{exp}^2 is the expected variance of the estimated parameter, $\boldsymbol{C}$ is the observation to observation covariance matrix, and $\boldsymbol{c}$ is the radial field source target to observation covariance vector. Minimization of the estimation variance is achieved through the method of Lagrange multipliers. The optimization problem is given by equation 2.2.5.

$$
\begin{aligned}
&minimize \quad \sigma_{exp}^2 + \boldsymbol{\omega}^T \boldsymbol{C} \boldsymbol{\omega} - 2\boldsymbol{\omega}^T \boldsymbol{c} \qquad (2.2.5) \\
&subject \quad to \quad \mathbf{1}^T \boldsymbol{\omega} - 1 = 0
\end{aligned}
$$

Choosing the Langrange multiplier as $-2\boldsymbol{\Lambda}$, the Lagrange function follows in equation 2.2.6.

$$
\mathcal{L} = \sigma_{exp}^2 + \boldsymbol{\omega}^T \boldsymbol{C} \boldsymbol{\omega} - 2\boldsymbol{\omega}^T \boldsymbol{c} + 2\boldsymbol{\Lambda}(\mathbf{1}^T \boldsymbol{\omega} - 1) \qquad (2.2.6)
$$

Differentiation with respect to the Kriging weights and the lagrange multiplier, and equating to zero, yields equations 2.2.7 and 2.2.8.

$$
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\omega}} = 2\boldsymbol{C}\boldsymbol{\omega} - 2\boldsymbol{c} + 2\boldsymbol{\Lambda}\mathbf{1}^T = \boldsymbol{C}\boldsymbol{\omega} - \boldsymbol{c} + \boldsymbol{\Lambda}\mathbf{1}^T = \mathbf{0} \qquad (2.2.7)
$$

$$
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\Lambda}} = 2\mathbf{1}^T \boldsymbol{\omega} - 2 = \mathbf{1}^T \boldsymbol{\omega} - 1 = 0 \qquad (2.2.8)
$$

With a bit of rearrangement, these two equations can be set up as a single system of linear equations, the ordinary Kriging system given in equation 2.2.9.

$$\begin{bmatrix} \boldsymbol{C} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c} \\ 1 \end{bmatrix} \tag{2.2.9}$$

Merging $\boldsymbol{C}$ and the Lagrange related one vectors into $\boldsymbol{K}$, the Kriging weights and the Lagrange multiplier into $\boldsymbol{\lambda}$, and $\boldsymbol{c}$ and the single one into $\boldsymbol{k}$, I present, in notation following Hansen and Mosegaard (2008), the ordinary Kriging system in equation 2.2.10.

$$\boldsymbol{K\lambda} = \boldsymbol{k} \tag{2.2.10}$$

Solving the ordinary Kriging system to get the Kriging weights, $\boldsymbol{\omega}$, and the Lagrange multiplier $\boldsymbol{\Lambda}$, can be done by inversion or similar linear systems solutions. Once the Kriging weights and Largrange multiplier are known, it is possible to determine the Kriging mean, μ_K, and Kriging variance σ_K^2. These results represent the mean and variance of a Gaussian probability density function at the target source location, conditional to known observations. The Kriging mean is simply the initial linear system that was laid out in equation 2.2.1, here vectorized in equation 2.2.11.

$$\mu_K = \boldsymbol{\omega}^T \boldsymbol{B_k}(\boldsymbol{r}) \tag{2.2.11}$$

And a simple expression for the Kriging variance is found from writing out the top row of the ordinary Kriging system shown in equation 2.2.9.

$$\begin{aligned} \boldsymbol{C\omega} + \boldsymbol{\Lambda 1} &= \boldsymbol{c} \\ \boldsymbol{C\omega} &= \boldsymbol{c} - \boldsymbol{\Lambda 1} \end{aligned}$$

And inserting into the estimation variance that was minimized from equation 2.2.4.

$$\begin{aligned} \sigma_K^2 &= \sigma_{exp}^2 + \boldsymbol{\omega}^T(\boldsymbol{c} - \boldsymbol{\Lambda 1}) - 2\boldsymbol{\omega}^T\boldsymbol{c} \\ &= \sigma_{exp}^2 + \boldsymbol{\omega}^T\boldsymbol{c} - \boldsymbol{\omega}^T\boldsymbol{\Lambda 1} - 2\boldsymbol{\omega}^T\boldsymbol{c} \\ &= \sigma_{exp}^2 - \boldsymbol{\omega}^T\boldsymbol{\Lambda 1} - \boldsymbol{\omega}^T\boldsymbol{c} \end{aligned}$$

Arriving at a simple expression for the Kriging variance in equation 2.2.12 by way of the unbiasedness condition, $\mathbf{1}^T\boldsymbol{\omega} = 1$.

$$\sigma_K^2 = \sigma_{exp}^2 - \boldsymbol{\omega}^T\boldsymbol{c} - \boldsymbol{\Lambda} \tag{2.2.12}$$

2.2.2 Ordinary Kriging applied to the Green's function description

In order to get the best linear unbiased estimate of the random variable at a target source location, conditional on available observations, it should now be clear that the ordinary Kriging system of equation 2.2.10 is to be solved. This will lead to the Kriging weights and Lagrange multiplier allowing computation of the random variable statistics, the Kriging mean and variance. In the case being considered here, the Green's function description of equation 2.1.8 is the link between observations and the target random variables at radial field source locations. The ordinary Kriging system of equation 2.2.10 is a method by which this system can be inverted, such that the random variable can be estimated through the linear

system of the Kriging mean and variance in equations 2.2.11 and 2.2.12. This collection of required equations are shown in 2.2.13.

$$\boxed{\begin{aligned} B_k(\boldsymbol{r}) &\approx \sum_{m=1}^{N_S} G_k(\boldsymbol{r}, \boldsymbol{r'_m}) B_r(\boldsymbol{r'_m}) \sin\theta'_m \Delta\theta'_m \Delta\phi'_m \\ \boldsymbol{K\lambda} &= \boldsymbol{k} \\ \mu_K &= \boldsymbol{\omega}^T \boldsymbol{B_k}(\boldsymbol{r}) \\ \sigma_K^2 &= \sigma_{exp}^2 - \boldsymbol{\omega}^T \boldsymbol{c} - \boldsymbol{\Lambda} \end{aligned}} \tag{2.2.13}$$

Using the Kriging system defined in the first row of 2.2.9, and given N_{obs} observations, $\boldsymbol{B_k}(\boldsymbol{r})$, the system depends on the Kriging weights, the data to data covariance, and the data to source target covariance. This can be written out as the sum seen in 2.2.14.

$$\sum_{j}^{N_{obs}} \omega_j C\Big\{ B_k(\boldsymbol{r_i}), B_k(\boldsymbol{r_j}) \Big\} = C\Big\{ B_k(\boldsymbol{r_i}), B_r(\boldsymbol{r'_t}) \Big\} \quad \forall i = 1, \ldots, N_{obs} \tag{2.2.14}$$

Where ω_j are the Kriging weights, $C\big\{B_k(\boldsymbol{r_i}), B_k(\boldsymbol{r_j})\big\}$ are the data to data covariances, and $C\big\{B_k(\boldsymbol{r_i}), B_r(\boldsymbol{r'_t})\big\}$ the data to target covariances. Using the Green's function description of the geomagnetic field of equation 2.1.8, the covariance expressions can be substituted as seen in equation 2.2.15 and 2.2.16, where the integration area factor is written $\Delta_x = \sin\theta'_x \Delta\theta'_x \Delta\phi'_x$ for simplicity.

$$\begin{aligned} C\big\{B_k(\boldsymbol{r_i}), B_k(\boldsymbol{r_j})\big\} &= C\Big\{ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) B_r(\boldsymbol{r'_m}) \Delta_m, \sum_{n=1}^{N_S} G_k(\boldsymbol{r_j}, \boldsymbol{r'_n}) B_r(\boldsymbol{r'_n}) \Delta_n \Big\} \\ &= \sum_{m=1}^{N_S} \sum_{n=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_j}, \boldsymbol{r'_n}) \Delta_m \Delta_n C\Big\{ B_r(\boldsymbol{r'_m}), B_r(\boldsymbol{r'_n}) \Big\} \\ C_{i,j} &= \sum_{m=1}^{N_S} \sum_{n=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_j}, \boldsymbol{r'_n}) \Delta_m \Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) \end{aligned} \tag{2.2.15}$$

$$\begin{aligned} C\big\{B_k(\boldsymbol{r_i}), B_r(\boldsymbol{r'_t})\big\} &= C\Big\{ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) B_r(\boldsymbol{r'_m}) \Delta_m, B_r(\boldsymbol{r'_t}) \Big\} \\ &= \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) \Delta_m C\Big\{ B_r(\boldsymbol{r'_m}), B_r(\boldsymbol{r'_t}) \Big\} \\ c_i &= \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \end{aligned} \tag{2.2.16}$$

This shows that the Kriging system depend on two geometric factors, the Green's function kernel, G_k, subject to the integration area factor, as well as the target source covariance matrices $C(\boldsymbol{r'_m}, \boldsymbol{r'_n})$ and $C(\boldsymbol{r'_m}, \boldsymbol{r'_t})$. The target source covariance terms can be found from modelling of training images and will be covered in chapter 3. The covariances take the matrix shapes given in 2.2.17.

$$\boldsymbol{C} = \begin{bmatrix} C(\boldsymbol{r'_1}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_1}, \boldsymbol{r'_2}) & \ldots & C(\boldsymbol{r'_1}, \boldsymbol{r'_n}) \\ C(\boldsymbol{r'_2}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_2}, \boldsymbol{r'_2}) & \ldots & C(\boldsymbol{r'_2}, \boldsymbol{r'_n}) \\ \vdots & \vdots & \ddots & \vdots \\ C(\boldsymbol{r'_m}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_m}, \boldsymbol{r'_2}) & \ldots & C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) \end{bmatrix}, \quad \boldsymbol{C_s} = \begin{bmatrix} C(\boldsymbol{r'_1}, \boldsymbol{r'_t}) \\ C(\boldsymbol{r'_2}, \boldsymbol{r'_t}) \\ \vdots \\ C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \end{bmatrix} \tag{2.2.17}$$

Where $C(\boldsymbol{r'_x}, \boldsymbol{r'_y})$ are the covariance between two points of a given spherical distance on the target source surface. Given observations, $\boldsymbol{B_k(r)}$, equation 2.2.15 and 2.2.16 can be written out as the $\boldsymbol{k}$ vector and $\boldsymbol{K}$ matrix of the ordinary Kriging system in equation 2.2.10.

$$\boldsymbol{c} = \begin{bmatrix} G_k(\boldsymbol{r_1}, \boldsymbol{r'_1})\Delta_1 & G_k(\boldsymbol{r_1}, \boldsymbol{r'_2})\Delta_2 & \dots & G_k(\boldsymbol{r_1}, \boldsymbol{r'_m})\Delta_m \\ \vdots & \vdots & \ddots & \vdots \\ G_k(\boldsymbol{r_i}, \boldsymbol{r'_1})\Delta_1 & G_k(\boldsymbol{r_i}, \boldsymbol{r'_2})\Delta_2 & \dots & G_k(\boldsymbol{r_i}, \boldsymbol{r'_m})\Delta_m \end{bmatrix} \cdot \begin{bmatrix} C(\boldsymbol{r'_1}, \boldsymbol{r'_t}) \\ C(\boldsymbol{r'_2}, \boldsymbol{r'_t}) \\ \vdots \\ C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \end{bmatrix}$$

$$\boldsymbol{k} = \begin{bmatrix} \sum_{m=1}^{N_S} G_k(\boldsymbol{r_1}, \boldsymbol{r'_m})\Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \\ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_2}, \boldsymbol{r'_m})\Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \\ \vdots \\ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m})\Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_t}) \\ 1 \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} G_k(\boldsymbol{r_1}, \boldsymbol{r'_1})\Delta_1 & G_k(\boldsymbol{r_1}, \boldsymbol{r'_2})\Delta_2 & \dots & G_k(\boldsymbol{r_1}, \boldsymbol{r'_m})\Delta_m \\ \vdots & \vdots & \ddots & \vdots \\ G_k(\boldsymbol{r_i}, \boldsymbol{r'_1})\Delta_1 & G_k(\boldsymbol{r_i}, \boldsymbol{r'_2})\Delta_2 & \dots & G_k(\boldsymbol{r_i}, \boldsymbol{r'_m})\Delta_m \end{bmatrix} \cdot \begin{bmatrix} C(\boldsymbol{r'_1}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_1}, \boldsymbol{r'_2}) & \dots & C(\boldsymbol{r'_1}, \boldsymbol{r'_n}) \\ C(\boldsymbol{r'_2}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_2}, \boldsymbol{r'_2}) & \dots & C(\boldsymbol{r'_2}, \boldsymbol{r'_n}) \\ \vdots & \vdots & \ddots & \vdots \\ C(\boldsymbol{r'_m}, \boldsymbol{r'_1}) & C(\boldsymbol{r'_m}, \boldsymbol{r'_2}) & \dots & C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) \end{bmatrix}$$

$$\cdot \begin{bmatrix} G_k(\boldsymbol{r_1}, \boldsymbol{r'_1})\Delta_1 & G_k(\boldsymbol{r_2}, \boldsymbol{r'_1})\Delta_1 & \dots & G_k(\boldsymbol{r_j}, \boldsymbol{r'_1})\Delta_1 \\ G_k(\boldsymbol{r_1}, \boldsymbol{r'_2})\Delta_2 & G_k(\boldsymbol{r_2}, \boldsymbol{r'_2})\Delta_2 & \dots & G_k(\boldsymbol{r_j}, \boldsymbol{r'_2})\Delta_2 \\ \vdots & \vdots & \ddots & \vdots \\ G_k(\boldsymbol{r_1}, \boldsymbol{r'_n})\Delta_n & G_k(\boldsymbol{r_2}, \boldsymbol{r'_n})\Delta_n & \dots & G_k(\boldsymbol{r_j}, \boldsymbol{r'_n})\Delta_n \end{bmatrix}$$

$$\boldsymbol{K} = \begin{bmatrix} \sum G_k(\boldsymbol{r_1}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_1}, \boldsymbol{r'_n})\Delta_m\Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) & \dots & \sum G_k(\boldsymbol{r_1}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_j}, \boldsymbol{r'_n})\Delta_m\Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \sum G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_1}, \boldsymbol{r'_n})\Delta_m\Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) & \dots & \sum G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_j}, \boldsymbol{r'_n})\Delta_m\Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) & 1 \\ 1 & \dots & 1 & 0 \end{bmatrix}$$

Ensuring the summations given by equation 2.2.15 and 2.2.16, where in the above, the single summation represents $\sum_{m=1}^{N_S}\sum_{n=1}^{N_S}$ for brevity. From these two matrices, the Kriging weights and Lagrange multiplier can be found by inversion of **K**, as given by equation 2.2.18, or by other linear system solutions.

$$\boldsymbol{\lambda} = \boldsymbol{K}^{-1}\boldsymbol{k}, \quad \boldsymbol{\lambda} = \begin{bmatrix} \boldsymbol{\omega_1} \\ \vdots \\ \boldsymbol{\omega_i} \\ \boldsymbol{\Lambda} \end{bmatrix} \tag{2.2.18}$$

The Kriging mean and variance can then finally be determined from these parameters and the available observations.

$$\mu_K = \boldsymbol{\omega}^T \boldsymbol{B_k(r)}$$
$$\sigma_K^2 = \sigma_{exp}^2 - \boldsymbol{\omega}^T \boldsymbol{c} - \boldsymbol{\Lambda}$$

2.3 Direct Sequential Simulation

For each radial field source target location, Kriging lets us compute the best linear unbiased estimate, however, in order to infer information about the local probability distributions beyond the mean, variance, and Gaussian shape, direct sequential simulation is used. In addition, direct sequential simulation allow reproduction of the prior two-point statistical information in a training image, like the covariance structure used in Kriging.

2.3.1 Sequential simulation

Sequential simulation is an application of conditional probabilities to infer random variable estimates based on known information, and sequentially generate posterior realizations of the Gaussian random field. Deutsch and Journel (1998) considers the complementary cumulative distribution function (CCDF) of N random variables, Z_i, conditioned on an arbitrary set of known variables, n.

$$ccdf_N(z_1, ..., z_N|n) = P\{Z_i \leq z_i, i = 1, ..., N|n\} \tag{2.3.1}$$

N samples can be drawn from this type of CCDF in N steps, estimating the random variables while increasing the conditioning in each step. The process is the following.

1. A value is drawn from the Z_1 CCDF, this draw is denoted z_1 and becomes part of the conditional set of known variables n, such that it grows to $n + 1$ known variables.
2. Values are drawn sequentially until all N random variables have been estimated. The conditional set of known variables grow by one for each draw.

The process requires computation of N CCDFs taking the form of equation 2.3.2.

$$\begin{aligned} &P\{Z_1 \leq z_1|n\} \\ &P\{Z_2 \leq z_2|(n + z_1)\} \\ &P\{Z_N \leq z_N|(n + z_1 + ... + z_{N-1})\} \end{aligned} \tag{2.3.2}$$

The complete sampling of N random variables constitute one realization of the Gaussian random field. Further realizations simply require rerunning the process. As Kriging estimates represent the mean and variance of the local Gaussian probability density function, conditional to known variables, it is a useful method for finding the basis of correct sequential Gaussian CCDFs. In order to apply this concept to ordinary Kriging of the Green's function description of geomagnetic vector field observations, such that the correct non-Gaussian local probability density functions are used, further measures need to be used. These measures result in direct sequential simulation with histogram reproduction.

2.3.2 Direct sequential simulation with histogram reproduction

To generate the covariance information needed for ordinary Kriging in equation 2.2.14, I explained that training images would need to be used. The same is true for direct sequential simulation (DSSIM). A prior two-point statistical idea of the radial field at the target source locations are needed in order to define a target histogram. In chapter 3 I outline the specifics of these training images to be used. The goal with DSSIM is to generate realizations of a non-Gaussian random field, in which the mean of realizations has mean and prior statistics reproduced, while honoring available observations. To achieve this, a range of local probability density functions conditioned to the target histogram are generated. Following Oz and Xie (2003), this is accomplished by first normal-score transforming the target histogram as given in equation 2.3.3.

$$\boldsymbol{y_h} = G_n^{-1}(F_h(\boldsymbol{z_h})) \tag{2.3.3}$$

Here, $\boldsymbol{y_h}$ is the normal-score transformed histogram values, G_n is a standard normal Gaussian cdf, F_h is the target histogram cdf, and $\boldsymbol{z_h}$ are the histogram values undergoing transformation. The cdf inverse is the quantile function, given in this case by G_n^{-1}. Knowing the normal-score transformation from equation (2.3.3), a back-transformation to the original histogram is possible through equation 2.3.4.

$$\boldsymbol{z_h} = F_h^{-1}(G_n(\boldsymbol{y_h})) \tag{2.3.4}$$

This is a tool that can be used to generate local conditional distributions by substituting $\boldsymbol{y_h}$ for non-standard normal-score values, as given by equation 2.3.5. Where $\boldsymbol{Z_F}$ is a matrix of the local conditional distributions, $\boldsymbol{G_{mv}^{-1}}$ are quantile functions for a range of non-standard normal distributions with given means and variances, and $\boldsymbol{q}$ are regularly spaced values between zero and one, defining the local conditional distribution size/resolution.

$$\boldsymbol{Z_F} = F_h^{-1}(G_n(\boldsymbol{G_{mv}^{-1}(q)})) \tag{2.3.5}$$

Defined like this, these local conditional distributions will have the correct shape conditional to the target histogram, a certain mean, μ_F, and a certain variance, σ_F^2. As mentioned, performing ordinary Kriging will yield an estimated Kriging mean and variance of the local Gaussian distributions, i.e. the wrong shape. As such, finding the correct local conditional distribution amounts to finding the back-transformed Gaussian distribution with mean and variance closest to the Kriging mean and variance.
Thus the target histogram enables the possibility of drawing samples from correctly shaped local conditional distributions. However, this ensures reproduction of the histogram, as well as its mean and variance, only if the applied local conditional distribution has mean and variance equal to the Kriging mean and variance (Journel, 1994). This further requires that the found local conditional distributions are scaled from near, to precisely the Kriging mean and variance. For a value sampled from one of the local conditional distributions, this can be achieved using equation 2.3.6.

$$z_s = (z_F - \mu_F) \cdot \frac{\sigma_k}{\sigma_F} + \mu_k \tag{2.3.6}$$

Here z_s is the final simulated value, z_F is the sample from the correctly shaped conditional distribution, μ_F is the mean, and σ_F is the standard deviation of that same distribution, with μ_k and σ_k being the Kriging mean and variance. Adding the above methods to the usual sequential simulation, direct sequential simulation proceeds as shown below. This procedure will ensure that simulations lead to a posterior probability density function (pdf) of the source radial field with correct mean, variance, covariance structure, and histogram.

1. Compute a lookup-table of local conditional distributions.
2. Determine a random path through the target source locations.
3. At each location in the random path, the Kriging mean, μ_k, and variance, σ_k^2, are calculated using all available observations and previously simulated values. Only the nearest values may need to be used in cases of large amounts of available variables.
4. Find the correct local conditional distribution. This corresponds to the Gaussian distribution closest to the Kriging mean and variance pre-back-transformation.
5. A simulated value is drawn from this correctly shaped local conditional distribution.
6. The simulated value is scaled such that it originates from a correctly shaped local conditional distribution, with mean and variance equal to the Kriging mean and variance.
7. The simulated value is added to the conditional data for use in the rest of the simulation.
8. 3.-7. is repeated until all target source locations have been visited.

2.3.3 DSSIM of ordinary Kriging applied to the Green's function description

With the addition of accounting for previously simulated values in DSSIM, the ordinary Kriging system must expand. The new system will include values determined at the radial field source target locations, and as such there is a need to include a covariance matrix between all variables at the source, as well as cross-covariance matrices from the values at the source to the observations. In addition, I include covariance associated with observation errors as $\boldsymbol{C_E}$. This results in the ordinary Kriging system shown in equation 2.3.7.

$$\boldsymbol{K\lambda} = \boldsymbol{k} \rightarrow \begin{bmatrix} \boldsymbol{C_{obs}} + \boldsymbol{C_E} & \boldsymbol{C_{cr}} & \boldsymbol{1} \\ \boldsymbol{C_{cr}^T} & \boldsymbol{C_S} & \boldsymbol{1} \\ \boldsymbol{1}^T & \boldsymbol{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega_{obs}} \\ \boldsymbol{\omega_S} \\ \boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c_{obs}} \\ \boldsymbol{c_S} \\ 1 \end{bmatrix} \tag{2.3.7}$$

The cross-covariances are simply a collection of the previously used observation to target source covariance, $\boldsymbol{c_{obs}}$, while the source to source and source to current target covariances are given directly by the modelled training image covariance. These matrices are shown in 2.3.8.

$$\begin{aligned} \boldsymbol{C_{cr}} &= \begin{bmatrix} \sum_{m=1}^{N_S} G_k(\boldsymbol{r_1}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{t1}}) & \dots & \sum_{m=1}^{N_S} G_k(\boldsymbol{r_1}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{ts}}) \\ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_2}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{t1}}) & \vdots & \sum_{m=1}^{N_S} G_k(\boldsymbol{r_2}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{ts}}) \\ \vdots & \vdots & \vdots \\ \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{t1}}) & \dots & \sum_{m=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) \Delta_m C(\boldsymbol{r'_m}, \boldsymbol{r'_{ts}}) \end{bmatrix} \\ \boldsymbol{C_S} &= \begin{bmatrix} C(\boldsymbol{r'_1}, \boldsymbol{r'_1}) & \dots & C(\boldsymbol{r'_1}, \boldsymbol{r'_n}) \\ \vdots & \ddots & \vdots \\ C(\boldsymbol{r'_m}, \boldsymbol{r'_1}) & \dots & C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) \end{bmatrix}, \quad \boldsymbol{c_S} = \begin{bmatrix} C(\boldsymbol{r'_{t1}}, \boldsymbol{r'_t}) \\ \vdots \\ C(\boldsymbol{r'_{ts}}, \boldsymbol{r'_t}) \end{bmatrix} \end{aligned} \tag{2.3.8}$$

The ordinary Kriging system is still solved by inversion or similar, the only difference being, that part of the Kriging weights are now associated with the previously simulated values as shown in 2.3.9.

$$\boldsymbol{\lambda} = \boldsymbol{K}^{-1}\boldsymbol{k}, \quad \boldsymbol{\lambda} = \begin{bmatrix} \boldsymbol{\omega_{obs}} \\ \boldsymbol{\omega_S} \\ \boldsymbol{\Lambda} \end{bmatrix} \tag{2.3.9}$$

Similarly, the Kriging mean and variance now depend on observations and previously simulated values, as well as both types of target covariance. The Kriging mean and variance computation is shown in equation 2.3.10. These results are then used to sample from the correctly shaped conditional distributions.

$$\boxed{\begin{aligned} \mu_K &= \begin{bmatrix} \boldsymbol{\omega_{obs}} \\ \boldsymbol{\omega_S} \end{bmatrix}^T \cdot \begin{bmatrix} \boldsymbol{B_k}(\boldsymbol{r}) \\ \hat{\boldsymbol{B}}_{\boldsymbol{r}}(\boldsymbol{r'_{ts}}) \end{bmatrix} \\ \sigma_K^2 &= \sigma_{exp}^2 - \begin{bmatrix} \boldsymbol{\omega_{obs}} \\ \boldsymbol{\omega_S} \end{bmatrix}^T \cdot \begin{bmatrix} \boldsymbol{c_{obs}} \\ \boldsymbol{c_S} \end{bmatrix} - \boldsymbol{\Lambda} \end{aligned}} \tag{2.3.10}$$

This concludes the direct sequential simulation implementation of ordinary Kriging applied to the Green's function description of the geomagnetic vector field. I now move on to the details of implementing the above, while using prior information from training images. This is followed by a description of used synthetic and Swarm satellite observations, as well as a description of the developed Python tool, in which it is all implemented.

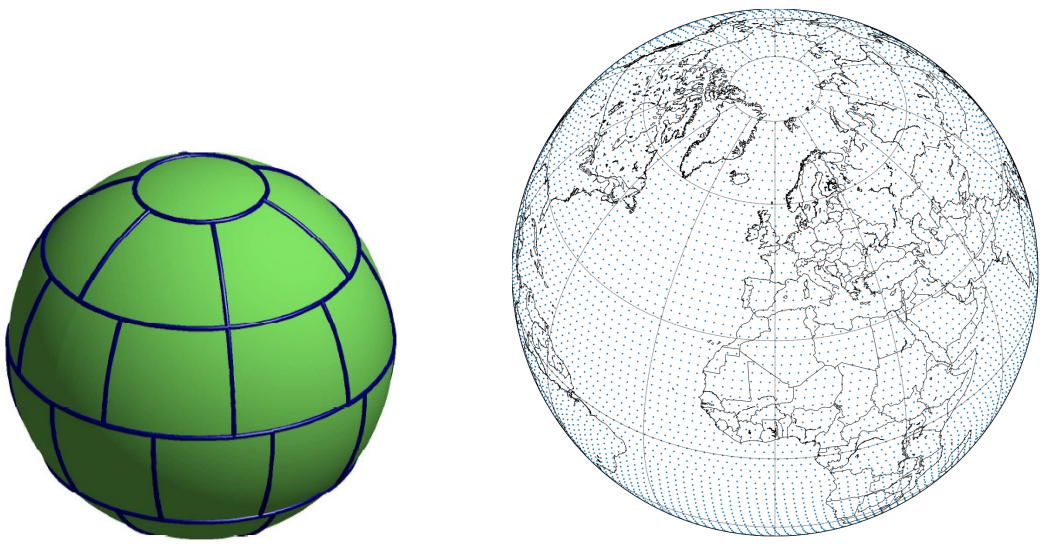
Chapter 3

Data

In order to carry out sequentially simulated ordinary Kriging as described in chapter 2, four methods have been used to handle and set up prior data. A pre-defined grid is necessary to work as the target source locations, training images of the target source work as prior information, semi-variogram modelling of the training images define required covariances, and correctly shaped local conditional distributions are needed to sample the simulation values. In the following I describe the origin and method by which these concepts have been implemented for use in spherical direct sequential simulation. In addition, a part of this chapter is dedicated to the used synthetic and Swarm satellite observations, followed by an overview of my spherical direct sequential simulation tool, bringing theory and data implementation together.

3.1 Approximate equal area grid

In the approximated integration of the Green's function description of the geomagnetic vector field, a grid of well known geometric properties is essential to properly define the differentials. One such grid can be found implemented in Matlab, and is a grid of locations covering approximate equal areas on the sphere. The implementation is formally called the Recursive Zonal Equal Area (EQ) Sphere Partitioning Toolbox and is based on Leopardi (2005). Figure 3.1.1 shows examples of this partitioning method. The grid is made from a unit sphere partition into regions optimized on equal area and small diameter, with the grid locations as the center point of these regions.



(a) **(b)**

Figure 3.1.1: Examples of recursive zonal equal area sphere partitioning. **(a)** shows a partitioned sphere of 33 areas from Leopardi (2005). **(b)** shows an Earth radius scaled grid of center points, for a partition of 10,000 areas using the Recursive Zonal Equal Area Sphere Partitioning Toolbox.

Properly defining the integration differentials can be done with the information seen on the sphere cross section of figure 3.1.2. The figure shows how the equal area spherical partition is defined through a certain amount of latitudinal collars and two polar caps. The collar edges are given by the red lines and the collar edge colatitude angles by the blue lines. The polar caps will have a single grid point and the collars an amount depending on the size of the partition.

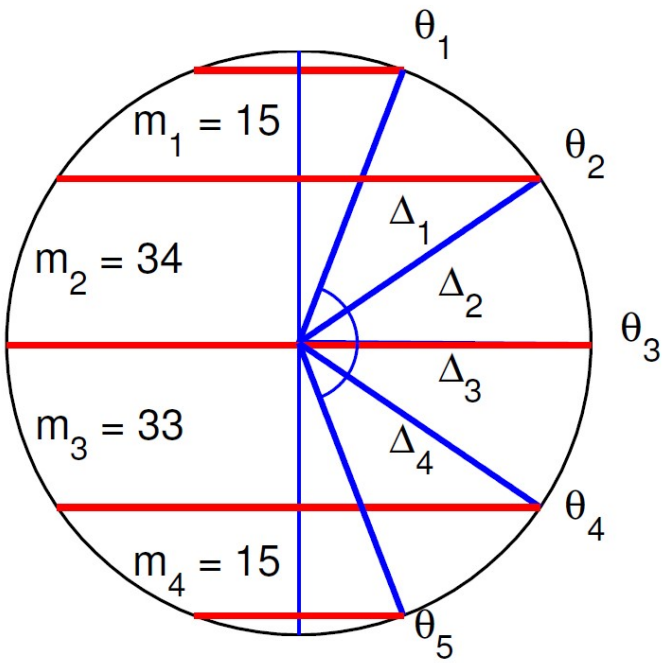


Figure 3.1.2: Partitioning system as shown in Leopardi (2005). A partition consists of two polar caps, a certain amount of collars, and a certain amount of areas in each collar, each with a center location. The red lines show the edges of the collars and the blue lines the colatitude angle to each collar edge. m_y shows the amount of areas in each collar for this specific partition.

The colatitudinal collar edge angles, θ_x, the amount of collar grid points, m_y, and the latitudinal angle width of each collar, Δ_z, as shown in figure 3.1.2, is available for any given partition. Through these, a simple definition of each grid location differential is achieved by equation 3.1.1.

$$\Delta\theta'_c = \theta_i - \theta_{i-1} = \Delta_z, \quad \Delta\phi'_c = \frac{2\pi}{m_y} \tag{3.1.1}$$

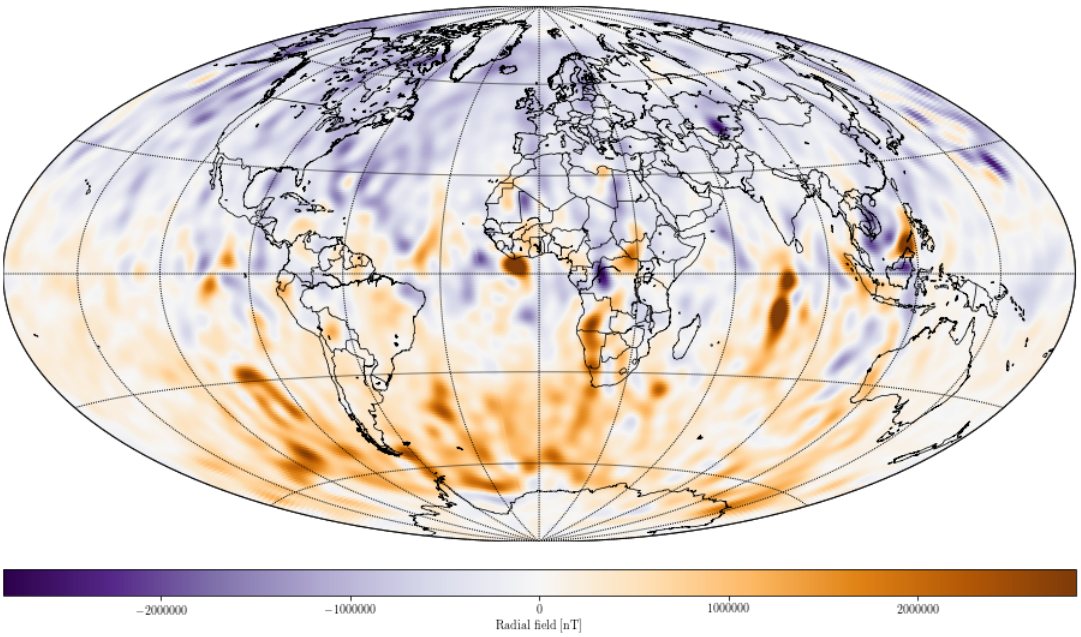
Where (θ'_c, ϕ'_c) is a collar with associated actual grid locations, (θ'_m, ϕ'_m), depending on the number of partitioned areas in each collar. From this, it is given that grid points of the same collar have identical differentials. Naturally, this is an approximation of the infinitesimal differentials of the integration, as such, with increasing grid size the sum of all approximated differentials should approach the product of the integration limits as previously described in equation 2.1.9 and repeated below. If this doesn't hold, the used grid size may be too small.

$$\sum_{m=1}^{N_S} \Delta\theta'_m \Delta\phi'_m = 2\pi^2$$

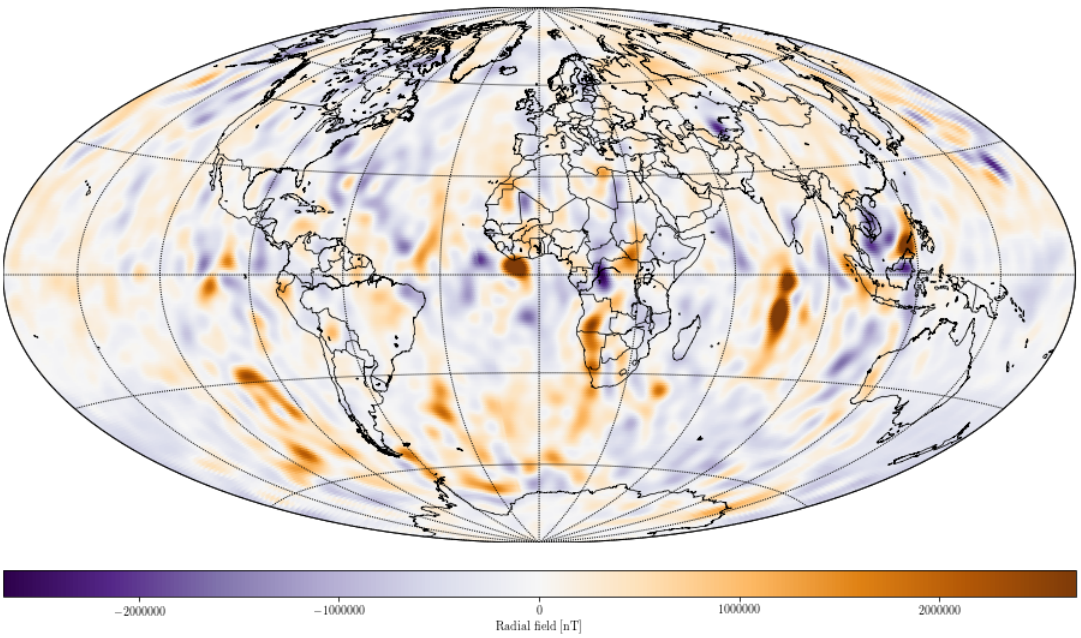
Finally, the approximate equal area grid is not just used during simulation to properly estimate integration. It is the surface on which the training image is generated and also how, through spherical distances from grid location to grid location, the covariance is modelled by semi-variogram analysis.

3.2 Training images

Training images encompass all prior information available for the simulation in the form of the covariance model and local conditional distributions. The first training image originate from core dynamo simulations through the work of Aubert (2017) and targets the radial field at the core mantle boundary. The model consists of Gauss coefficients up to degree 60 and has been determined at an approximate equal area grid with radius $r_{cmb} = 3480.0\,\text{km}$, using design matrices by Olsen (2018). Examples of the computed training images can be seen in figure 3.2.1, where they have been generated on a large grid with 100,000 target locations. A version is generated simultaneously each time with the latitudinal trend / dipole removed. This is necessary for semi-variogram modelling.



(a) Radial field model at the core mantle boundary.



(b) Latitudinal mean removed

Figure 3.2.1: Training images for the radial field at the core mantle boundary derived from a spherical harmonic model up to degree 60. The model is based on core dynamo simulations by Aubert (2017). **(a)** is the full model while **(b)** has the latitudinal mean removed.

The second training image is from modelling the remanent magnetization of the oceans in combination with a full Earth model of induced magnetization (Masterton, 2013). The model has Gauss coefficients up to degree 256, however, only up to degree 100 has been utilized here due to computational memory constraints. The training image is again the radial field computed in an approximate equal area grid, but this time with a lithospheric radius, $r_{LS} = 6371.2$ km, at Earth's surface. The training image can be seen in figure 3.2.2, where it has been generated on a large grid with 100.000 target locations. While these examples have been generated on large grids, any desirable size and radius of grid is possible.

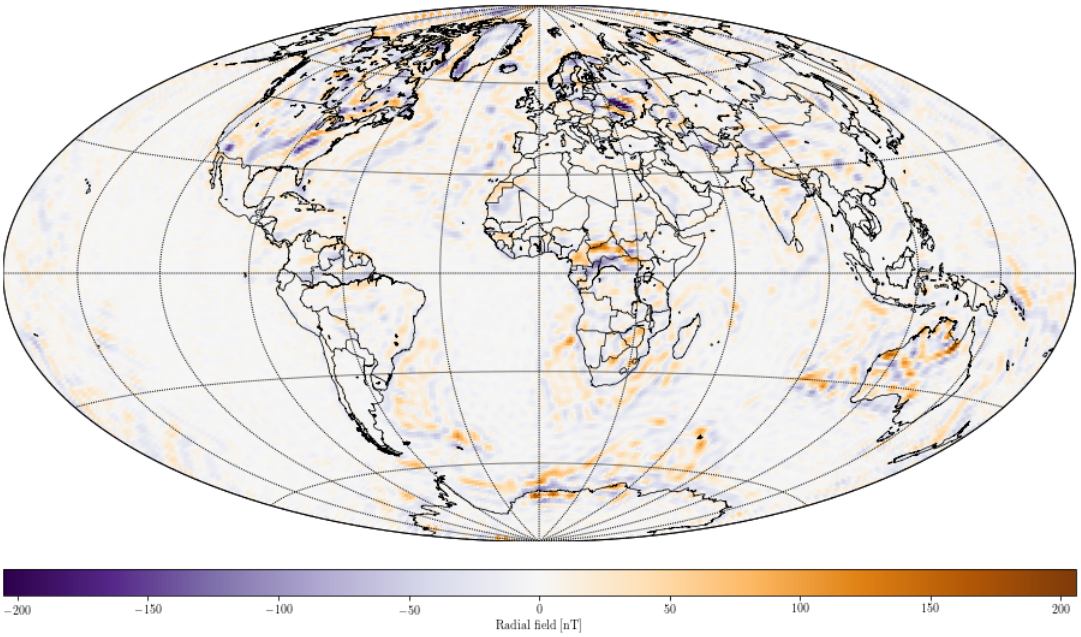


Figure 3.2.2: Training image for the radial field at Earth's surface derived from a spherical harmonic model up to degree 100. The model is based on modelling of remanent magnetization in the oceans combined with a full Earth model of induced magnetization (Masterton, 2013)

Note that since the training images are produced from models based on spherical harmonic expansion, they are not defined at the poles, but the grids do include the poles. These locations will have to be removed or estimated based on nearby points. Their exact values won't have a great impact as the training image information is only explicitly used through modelling/analysis of the overall statistical structure. This will be in the form of histograms for local conditional distribution generation and second order stationary spatial variability for semi-variogram modelling, neither of which take exact position into account.

3.3 Semi-variogram modelling of training images

In order to define the covariance based Kriging systems discussed in chapter 2, semi-variogram models are used. In general, a covariance function is a traditional measure for the spatial variability of a random variable (RV). The covariance function for a RV with realizations separated spatially and with the intrinsic hypothesis that their expected values are constant, is given by equation 3.3.1.

$$C(\boldsymbol{u}, \boldsymbol{h}) = E\Big\{[(Z(\boldsymbol{u}) - \mu)][Z(\boldsymbol{u} + \boldsymbol{h}) - \mu]\Big\} \tag{3.3.1}$$

Where $E\{\}$ denotes the expected value, Z is a random variable, $\boldsymbol{u}$ is a location in space, $\boldsymbol{h}$ is a vector pointing to a second location, and μ is the mean of the random variable. In semi-variogram analysis, the covariance of data to be analysed is assumed to only depend on separation distance. In other words, it is second order stationary such that $C(\boldsymbol{u}, \boldsymbol{h}) = C(\boldsymbol{h})$.

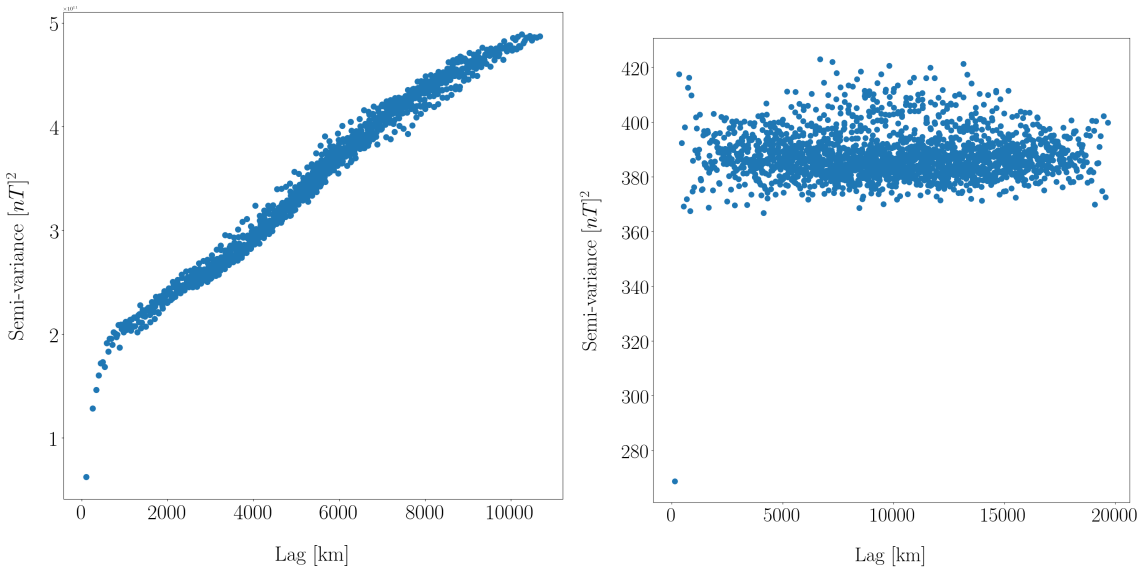
Applying this principle to a location with itself ($\boldsymbol{h} = 0$), it follows that under these conditions the random variable has the same variance everywhere in considered space.

$$C(\boldsymbol{u}, 0) = C(0) = E\Big\{(Z(0) - \mu)(Z(0) - \mu)\Big\} = E\Big\{(Z(0) - \mu)^2\Big\} = Var\{Z\} \tag{3.3.2}$$

Assuming validity in the intrinsic hypothesis of constant exptected value (mean) and second order stationarity, a semi-variogram is then defined through the two simple relations given in equation 3.3.3, following Deutsch and Journel (1998). The first relation connects covariance to semi-variograms as needed in Kriging.

$$\begin{aligned} \gamma &= C(0) - C(\boldsymbol{h}), \quad \forall \boldsymbol{u} \\ \gamma &= \frac{1}{2} E\Big\{[Z(\boldsymbol{u}) - Z(\boldsymbol{u} + \boldsymbol{h})]^2\Big\} \end{aligned} \tag{3.3.3}$$

Where γ is a semi-variogram value for a location in the grid, $\boldsymbol{u}$, with respect to some other location at $\boldsymbol{u} + \boldsymbol{h}$. In implementation, semi-variogram values are calculated by ordering the squared location-pair difference values, $[Z(\boldsymbol{u}) - Z(\boldsymbol{u} + \boldsymbol{h})]^2$, for all grid locations in accordance to distance from each other. These distances are denoted the lag, $\boldsymbol{h}$. Once the values are ordered according to lag, the mean is taken over equal ranges of lags followed with division by two.
The semi-variogram of the full core mantle boundary and lithosphere training images can be seen in figure 3.3.1. Note the upward trend of the CMB and how very few values are available at small lags for the lithosphere. For the CMB the trend indicates a non-constant mean in the training image, and for the lithosphere a larger grid may be needed to properly resolve the semi-variogram.



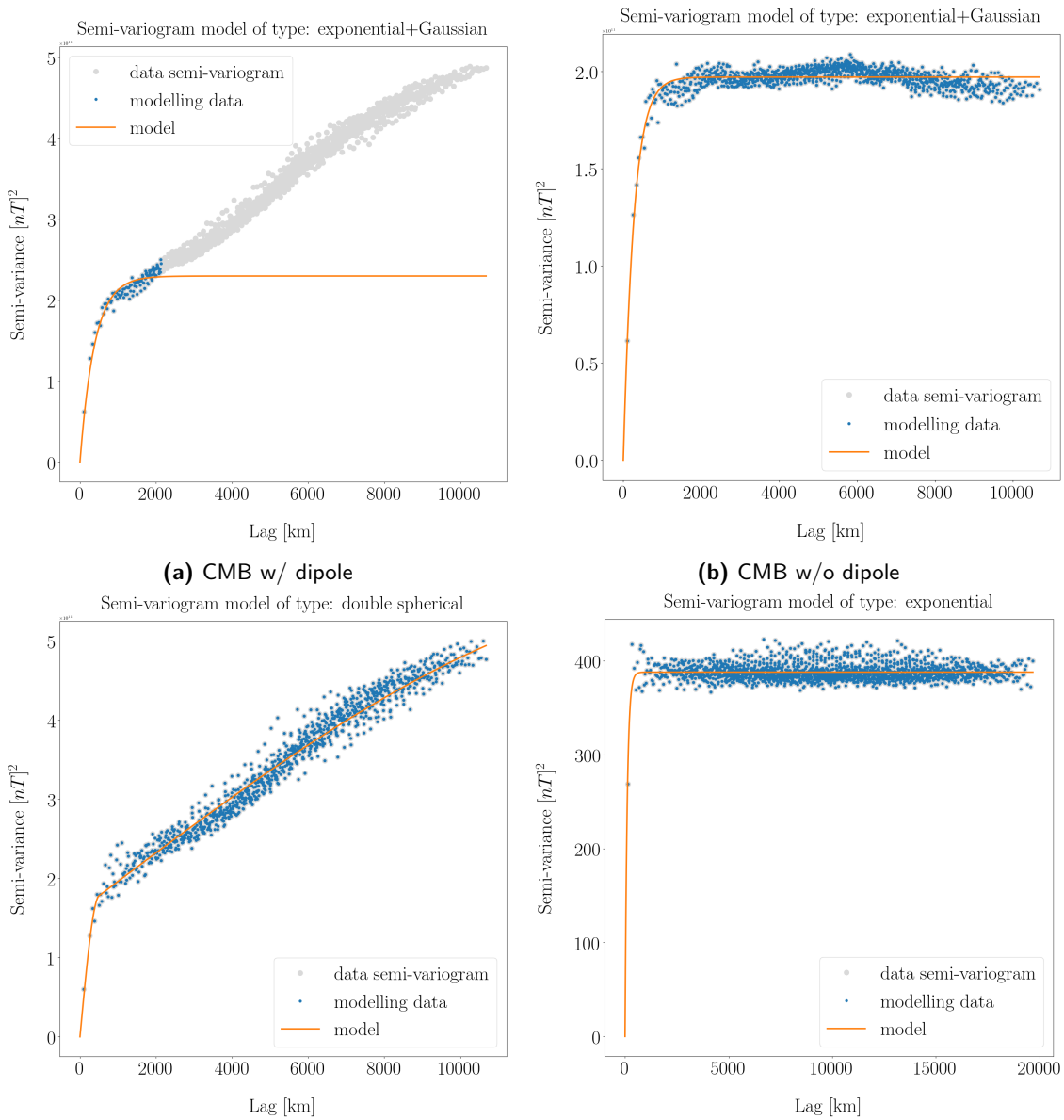
(a) CMB **(b)** Lithosphere

Figure 3.3.1: Semi-variograms of the core mantle boundary and lithosphere training images for a source grid of 10,000 locations.

There are many possible models that can be applied to characterize these semi-variograms. In this case an exponential, double spherical, or nested exponential and Gaussian model has primarily been used here. The nested model is seen in equation 3.3.4, where C_0 is called the nugget effect (not to be confused with $C(0) = Var\{Z\}$), $C_0 + C_1 + C_2$ the sill, and a the range of the model.

$$\gamma(h) = C0 + C1\left(1 - \exp\left(-\frac{3h}{a}\right)\right) + C2\left(1 - \exp\left(-\frac{(3h)^2}{a^2}\right)\right) \tag{3.3.4}$$

The nugget is a discontinuity at $\boldsymbol{h} = 0$ due to uncertainty or small scale variability, the sill is the training image variance, and the range is the point of no correlation between the source locations. In figure 3.3.2 models have been fit to the training image semi-variograms. A nested exponential and Gaussian, as well as a double spherical model for the core mantle boundary, and an exponential model for the lithosphere. The nugget has been set to zero, the sill is the semi-variance value at the leveled out part of the model, and the range is the lag at which the model levels out. The lag is set to zero as analysis excluding longer scale variability show the true nugget to be close to zero in both cases. It is difficult to capture the structure of both in a single model fit, as the number of lags required to see small scale variability quickly overwhelms the long scale variability. As seen, a model has also been generated from the CMB training image with latitudinal mean removed. This is to get around the intrinsic hypothesis not being valid, while not leaving out modelling data. Note that not all data is used in modelling the dipole CMB semi-variogram. Including more data will disturb the small scale fit for this particular model. In addition, the double spherical model has been determined from visual inspection.



(a) CMB w/ dipole

(b) CMB w/o dipole

(c) CMB double spherical

(d) Lithosphere

Figure 3.3.2: **(a)** and **(b)** Nested exponential and Gaussian semi-variogram models of the core mantle boundary training image. **(c)** Double spherical semi-variogram model of the core mantle boundary training image. **(d)** Exponential semi-variogram model of the lithosphere training image. All models have been generated from a source grid of 10,000 locations.

3.4 Local conditional distributions

In sequential simulation, conditional probabilities are used to infer values of a random variable on a spatial grid, based on available observations and previously determined values of the random variable. Kriging is a method of determining these conditional probabilities based on available values and a covariance function linking them. What Kriging specifically yields, is the mean and variance of the local normal probability density function, conditional to known variables. However, in direct sequential simulation the local probability density functions are not normal distributions, but depends on the available data, also called the target histogram. Luckily, it is possible to generate the proper local probability densities prior to running direct sequential simulation. These local probability densities are the contents of the conditional distribution table. The local probability densities can take any appropriate shape as long as values can be drawn from them such that increasing draws move toward the shape of the local probability distribution. Here they are quantile functions (QFs), also called inverse cumulative distribution functions.
In the following I describe my current process of generating a conditional distribution table for spherical direct sequential simulation. In brief, the target histogram is first normal score transformed and then used as input for back-transformation of a range of normal quantile functions, varying in mean and variance. This is accomplished either by loading Fortran scripts from GSLIB (Deutsch and Journel, 1998) as used in VISIM (Hansen and Mosegaard, 2008) and SIPPI (Hansen and Mosegaard, 2013a), or directly in Python using the function *QuantileTransformer* from the scikit-learn API by Buitinck et al. (2013). This brief explanation is outlined in figure 3.4.1.

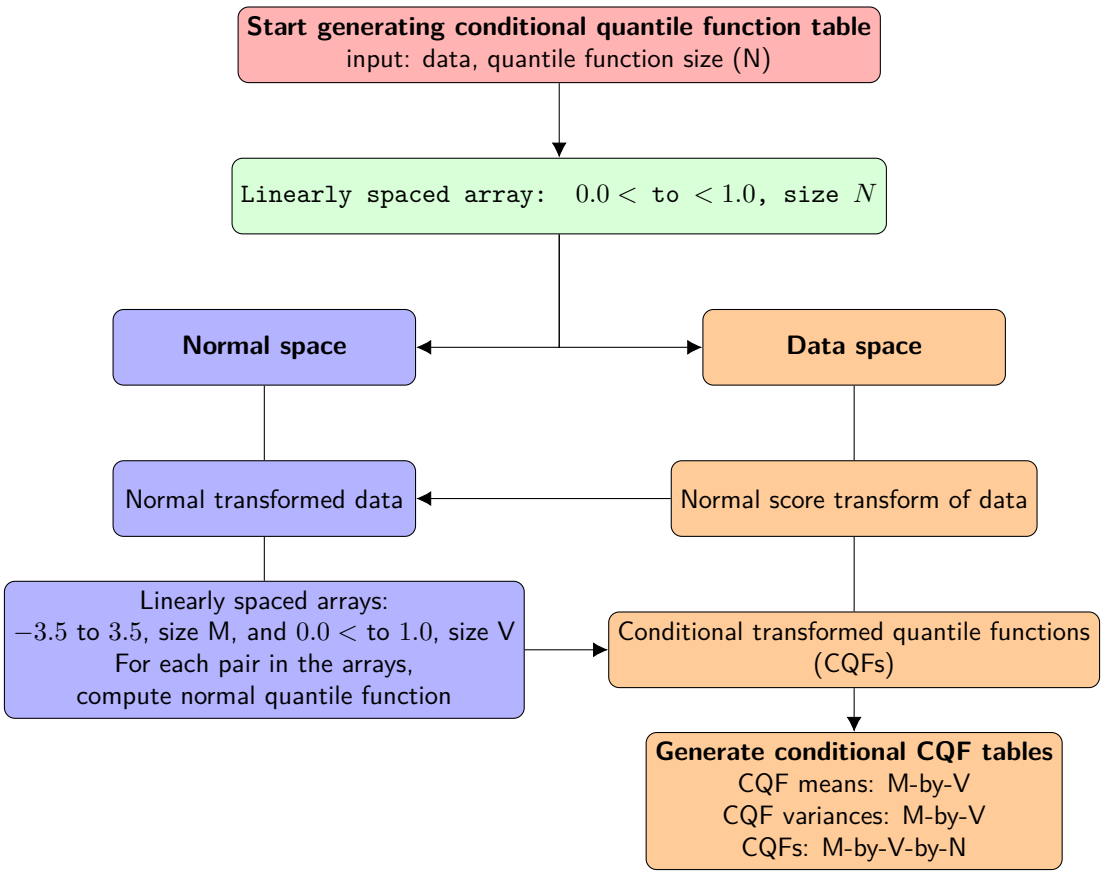


Figure 3.4.1: Flowchart showing the process of generating correctly shaped local conditional distributions with data in the form of training images.

The algorithm is activated with the data (training image histogram) and desired size, N, of the QFs as input. Then a linearly spaced array of size N, ranging from near zero to near one is generated. Excluding zero and one, as the QFs become infinite here. Moving on in the algorithm, the computations are split into two spaces, normal and data. From data space, the target histogram is normal score transformed such that its values resemble the standard normal distribution (mean zero, variance one). In normal space, two

linearly spaced arrays are generated. One ranging across the likely values of the standard normal random variable (about -3.5 to 3.5) and the other from near zero variance to the standard normal variance of one. Each pair in these two arrays are now used to compute a normal QF, with mean and variance given by the pairs. This is why the variance can't be zero, as it is used in division to scale the standard normal QF to a normal QF close to the desired mean and variance. These generated normal QFs are now transformed into data space using the parameters from the normal score transform of the data. After transformation, the conditional QFs (CQFs) represent the local probability density functions desired for direct sequential simulation. In addition to the CQFs, their mean and variances are saved in tables such that the correct function can be found given Kriging mean and variances. It is important to note that the found mean and variance of the CQFs are only approximately equal to the Kriging mean and variances, however, scaling according to Oz and Xie (2003) can be used to correct this as explained in chapter 2. In the following sections I further explain the concepts of normal score transformation, quantile functions, and conditional transformation (back-transformation from normal space to data space).

3.4.1 Normal score transformation

Normal score transformation can be applied to any histogram. The result is scaling and spreading of the histogram values such that they follow a standard normal distribution. In figure 3.4.2, the histogram of the lithospheric and core mantle boundary training images have been normal score transformed through the available methods (Fortran and Python based). The specifics of the Fortran method are described in appendix C. Note how the histogram values are successfully scaled down to the standard normal distribution value range (about -3.5 to 3.5).

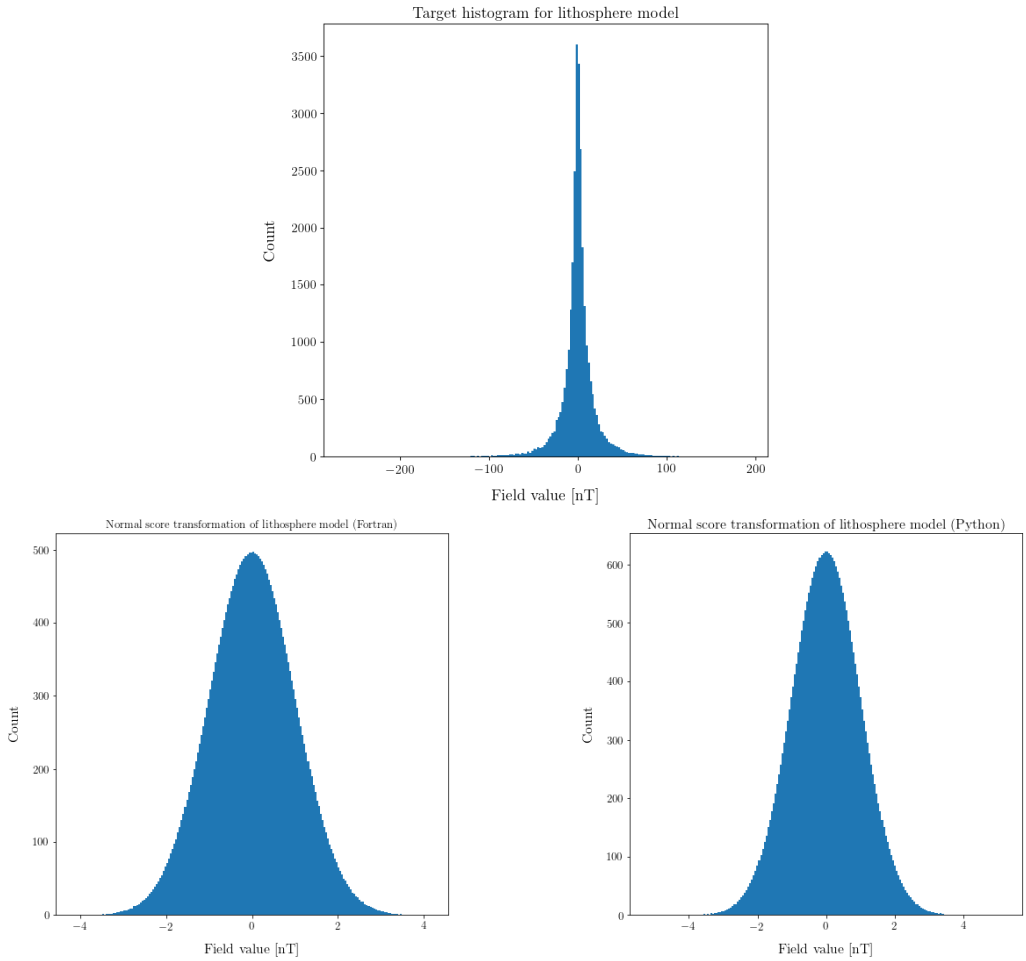


Figure 3.4.2: Target histogram for lithospheric training image with its normal score transformations using Fortran and Python methods.

3.4.2 Quantile functions and conditional transformation

The quantile function, or inverse cumulative distribution, is associated with a probability distribution of a random variable. Specifically, it is a function of the probability that the random variable is less than or equal to its function value.

$$F(P(Z \leq F)) = Z_F \quad \text{for} \quad 0 \leq P(Z \leq F) \leq 1 \tag{3.4.1}$$

Here F is the quantile function, Z is the random variable, Z_F is a value of the random variable, and $P(Z \leq F)$ is the probability that the random variable is less than or equal to the function value. Figure 3.4.3 shows the lithosphere and core training image quantile functions in relation to normal quantile functions using the same mean and variance.

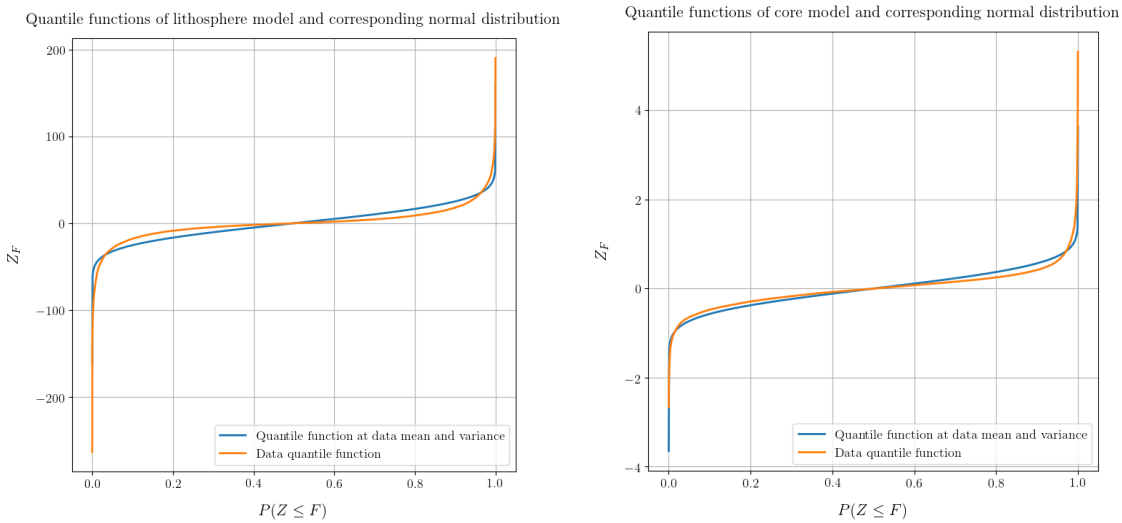


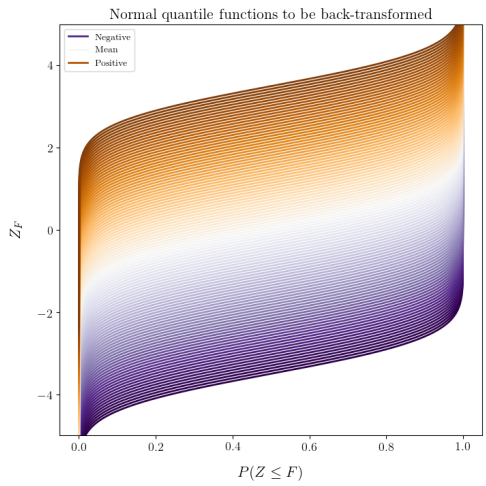
Figure 3.4.3: Quantile function comparisons between lithosphere (left) and core (right) model with a normal distribution of the same mean and variance as the respective models.

These differences are what makes the direct sequential simulation method necessary, due to Kriging only estimating the mean and variance of the local *normal* pdf. The method is based on the transformation of the data QF into the normal QF described in section 3.4.1 (with the additional scaling down to the standard normal QF). This is a reversible transformation that can be used to transform local normal distributions into local conditional distributions. As previously described I use quantile functions to represent the distributions, as such, to end up with a table of conditional QFs, first a table of normal QFs are generated. The normal QFs are generated from ranges of mean and variance such that they cover most of the standard normal distribution (about mean: -3.5 to 3.5, and var: 0.0 to 1.0). These ranges are shown in equations 3.4.2 and 3.4.3. Note the 0.0 in the mean range, I have found it very important to include this midpoint. From these ranges, normal quantile functions are computed for the $M \times V$ pairs, this is illustrated in equation 3.4.4. The form of these normal quantile functions are shown in figure 3.4.4a, for the full mean range with constant variance at 0.5.

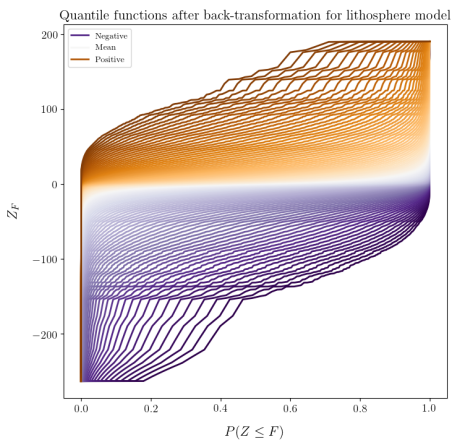
$$m_M = \begin{bmatrix} -3.5 & \dots & 0.0 & \dots & 3.5 \end{bmatrix} \tag{3.4.2}$$

$$v_V = \begin{bmatrix} \approx 0.0 & \dots & 1.0 \end{bmatrix} \tag{3.4.3}$$

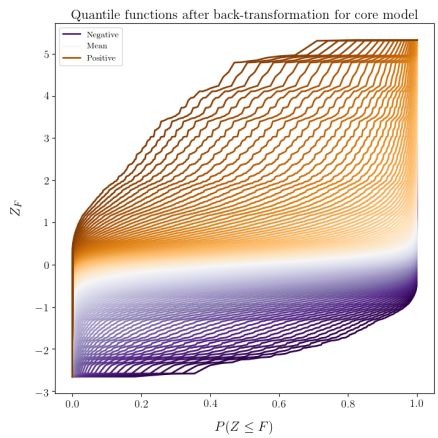
$$QF(m, v) = \begin{bmatrix} QF(m_0, v_0) & QF(m_0, v_1) & \dots & QF(m_0, v_V) \\ QF(m_1, v_0) & QF(m_1, v_1) & \dots & QF(m_1, v_V) \\ \vdots & \vdots & \ddots & \vdots \\ QF(m_M, v_0) & QF(m_M, v_1) & \dots & QF(m_M, v_V) \end{bmatrix} \tag{3.4.4}$$



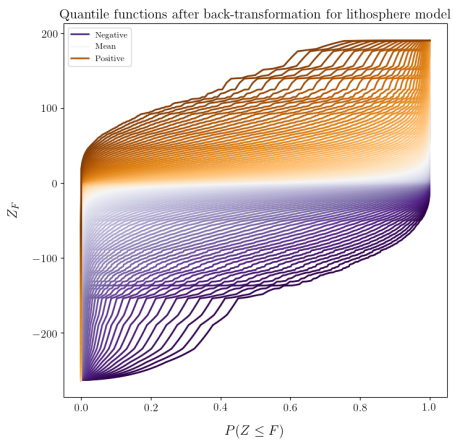
(a) Normal QFs.



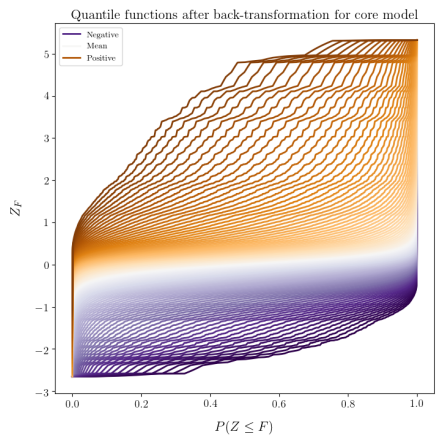
(b) Lithosphere, Fortran.



(c) Core, Fortran.



(d) Lithosphere, Python.



(e) Core, Python.

Figure 3.4.4: The range of normal quantile functions for conditional transformation are shown in 3.4.4a for the full mean range -3.5 to 3.5 with constant variance at 0.5. The back-transformed quantile functions follow in figure 3.4.4b to 3.4.4e (with model type and method used for computation shown).

In figure 3.4.4b to 3.4.4e the normal quantile functions have been back-transformed using the normal data transformation of the lithosphere and core training image respectively.
These conditionally transformed quantile functions are the final contents of the conditional distribution table. The table is illustrated in equation 3.4.5 where the conditional quantile functions are denoted CQF. Note that their position in the table still depend on the defined normal mean and variance ranges.

$$CQF(m,v) = \begin{bmatrix} CQF(m_0,v_0) & CQF(m_0,v_1) & \dots & CQF(m_0,v_V) \\ CQF(m_1,v_0) & CQF(m_1,v_1) & \dots & CQF(m_1,v_V) \\ \vdots & \vdots & \ddots & \vdots \\ CQF(m_M,v_0) & CQF(m_M,v_1) & \dots & CQF(m_M,v_V) \end{bmatrix} \tag{3.4.5}$$

In addition to the transformed quantile functions themselves, tables of the expected value (mean) and variance are saved as well. These are illustrated in equations 3.4.6 and 3.4.7, with shorthand E and Var for brevity.

$$E\Big[CQF(m,v)\Big] = E(m,v) = \begin{bmatrix} E(m_0,v_0) & E(m_0,v_1) & \dots & E(m_0,v_V) \\ E(m_1,v_0) & E(m_1,v_1) & \dots & E(m_1,v_V) \\ \vdots & \vdots & \ddots & \vdots \\ E(m_M,v_0) & E(m_M,v_1) & \dots & E(m_M,v_V) \end{bmatrix} \tag{3.4.6}$$

$$Var\Big[CQF(m,v)\Big] = Var(m,v) = \begin{bmatrix} Var(m_0,v_0) & Var(m_0,v_1) & \dots & Var(m_0,v_V) \\ Var(m_1,v_0) & Var(m_1,v_1) & \dots & Var(m_1,v_V) \\ \vdots & \vdots & \ddots & \vdots \\ Var(m_M,v_0) & Var(m_M,v_1) & \dots & Var(m_M,v_V) \end{bmatrix} \tag{3.4.7}$$

The length of the $CQFs$ depend on the normal quantile functions generated. Given a QF length of N, the conditional distribution table has size $N \times M \times V$, with the mean and variance tables having size $M \times V$. The resulting mean and variances of the $CQFs$ are illustrated in figure 3.4.5 for each training image, such that each point represents a conditional transformed quantile function of the shown mean and variance.

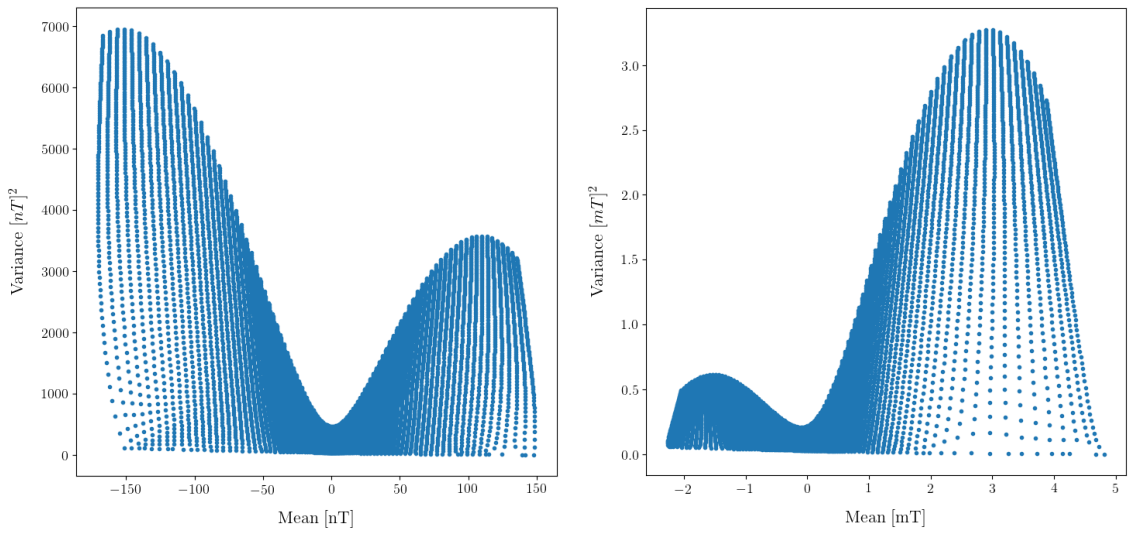
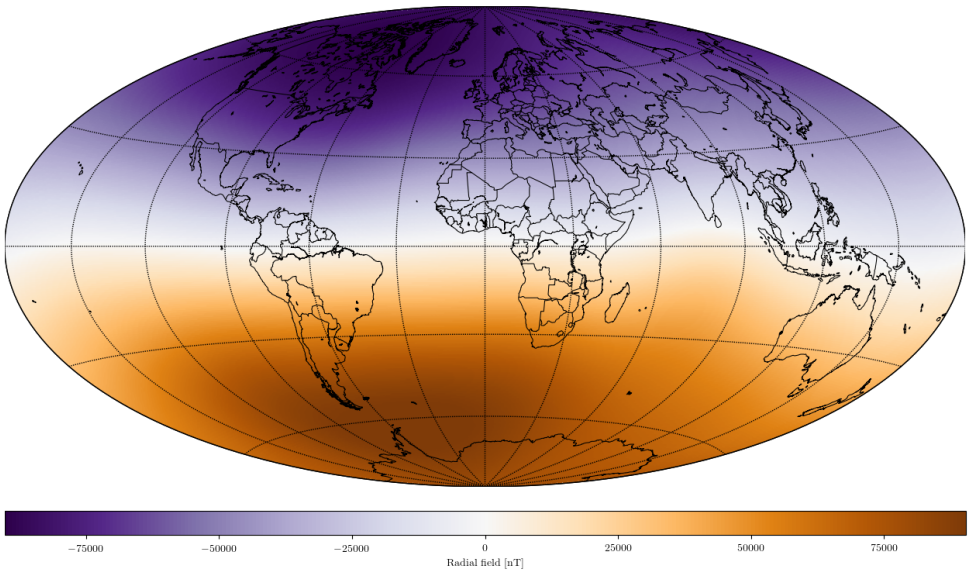


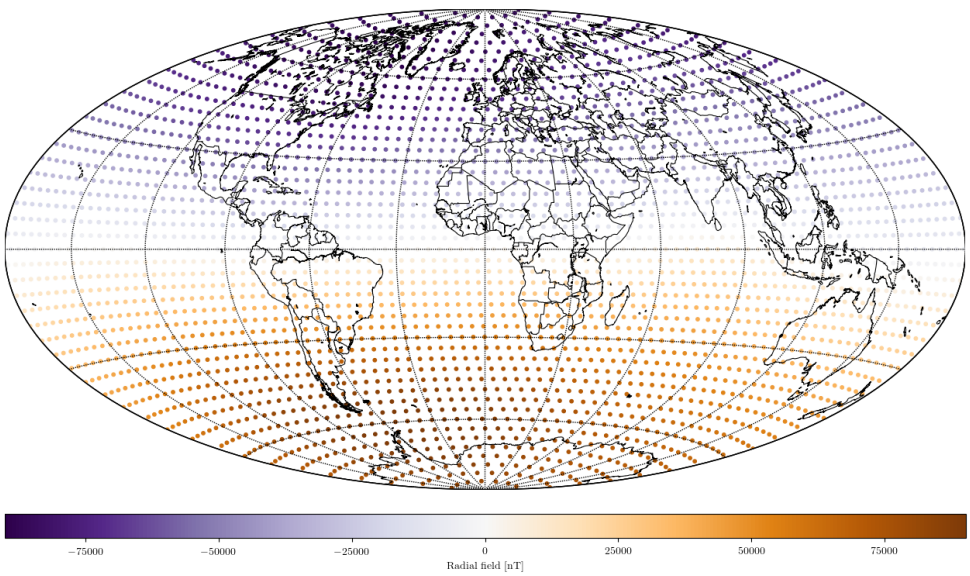
Figure 3.4.5: Spread of the mean and variance for the conditional distributions. The lithosphere is shown to the left and the core to the right. Each point represents a conditional transformed quantile function.

3.5 Synthetic satellite observations

The last piece needed before simulations can be run, is having useful observations to base the simulations on. I perform extensive preliminary testing with a set of synthetic observations of the same origin as the core mantle boundary training images. That is, they are based on core dynamo simulation by Aubert (2017). The target location in this case is satellite altitude at 300 km above Earth's surface. This puts the synthetic observations at an optimistic low altitude, which normally occur late in a satellites active life, but yield the best measurements.
Figure 3.5.1 show synthetic observations for the radial component of the geomagnetic vector field at satellite altitude. The Green's function description encompass all three components, latitudinal, longitudinal, and radial, however only the radial component has been considered in this project. While setting up the Kriging system, $1\,\mathrm{nT}^2$ of observation covariance noise is added as described in chapter 2. In addition, the pole locations not defined by the spherical harmonic model has been removed from the synthetic observation sets.



(a) 100,000 synthetic radial field observations



(b) 3,000 synthetic radial field observations

Figure 3.5.1: Synthetic observations for the radial field at the satellite altitude derived from a spherical harmonic model up to degree 60. The model is based on core dynamo simulations by Aubert (2017). (a) is synthetic observations generated on a grid of 99,998 locations (b) is a grid of only 2,998 locations.

3.6 Satellite observations from Swarm

The final results are based on radial field observations from the Swarm satellites over a three month period from April-June 2018. Swarm is a constellation of three satellites (alpha, beta, charlie) in polar orbits, grouped as shown on the illustration in figure 3.6.1. The observations have been selected following the criteria used for the CHAOS-6 geomagnetic field model (Finlay et al., 2016). These criteria follow selection only during dark, i.e. as the Sun is 10 degrees below the horizon, as well as geomagnetically quiet conditions. Specifically, vector field data is only selected when the field strength due to magnetospheric ring currents changes with less than 2 nT/h and when the geomagnetic activity index is below the threshold, $K_p \leq 2^0$ for latitudes of $\pm 55°$ in Quasi-Dipole (QD) coordinates (coordinate system defined from magnetic field lines, see Richmond, 1995).
The datum rate of selection was every 5 minutes from ESA's L1b Swarm 1 Hz data files, with vector alignment of Swarm vector magnetometer and star camera provided by CHAOS-6. The available observation amount is 2,773, 2,509, and 2,696 radial field observations respectively for alpha, beta, and charlie, leading to a total of 7,978 observations. The used radial field satellite observations selected from Swarm alpha can be seen on figure 3.6.2.

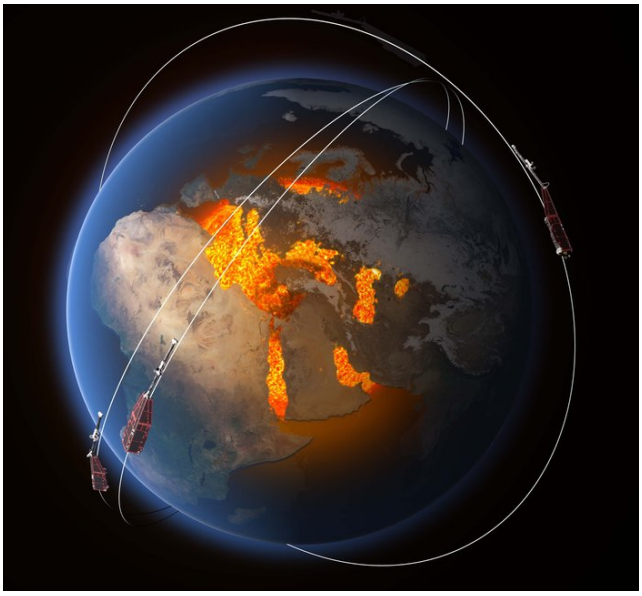


Figure 3.6.1: The Swarm satellite constellation. From: esa.int/spaceinimages

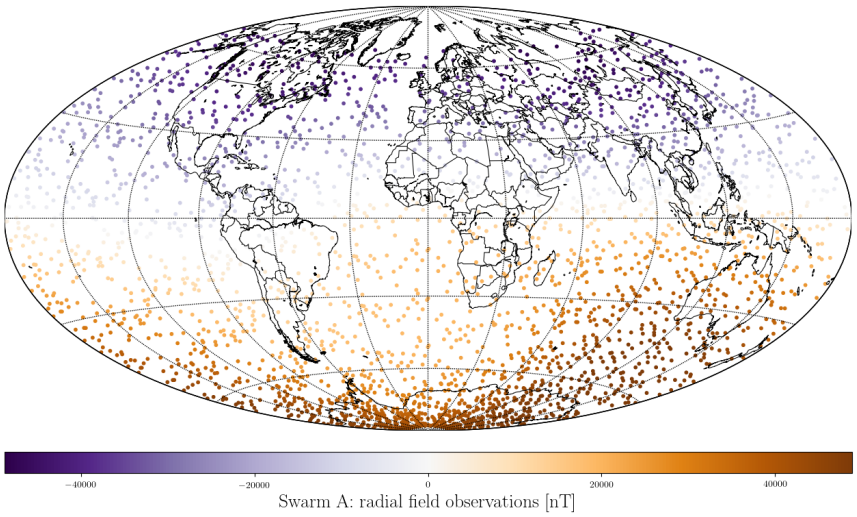


Figure 3.6.2: 2,773 radial field observations from Swarm alpha.

3.7 Geostatistical tool: SDSSIM

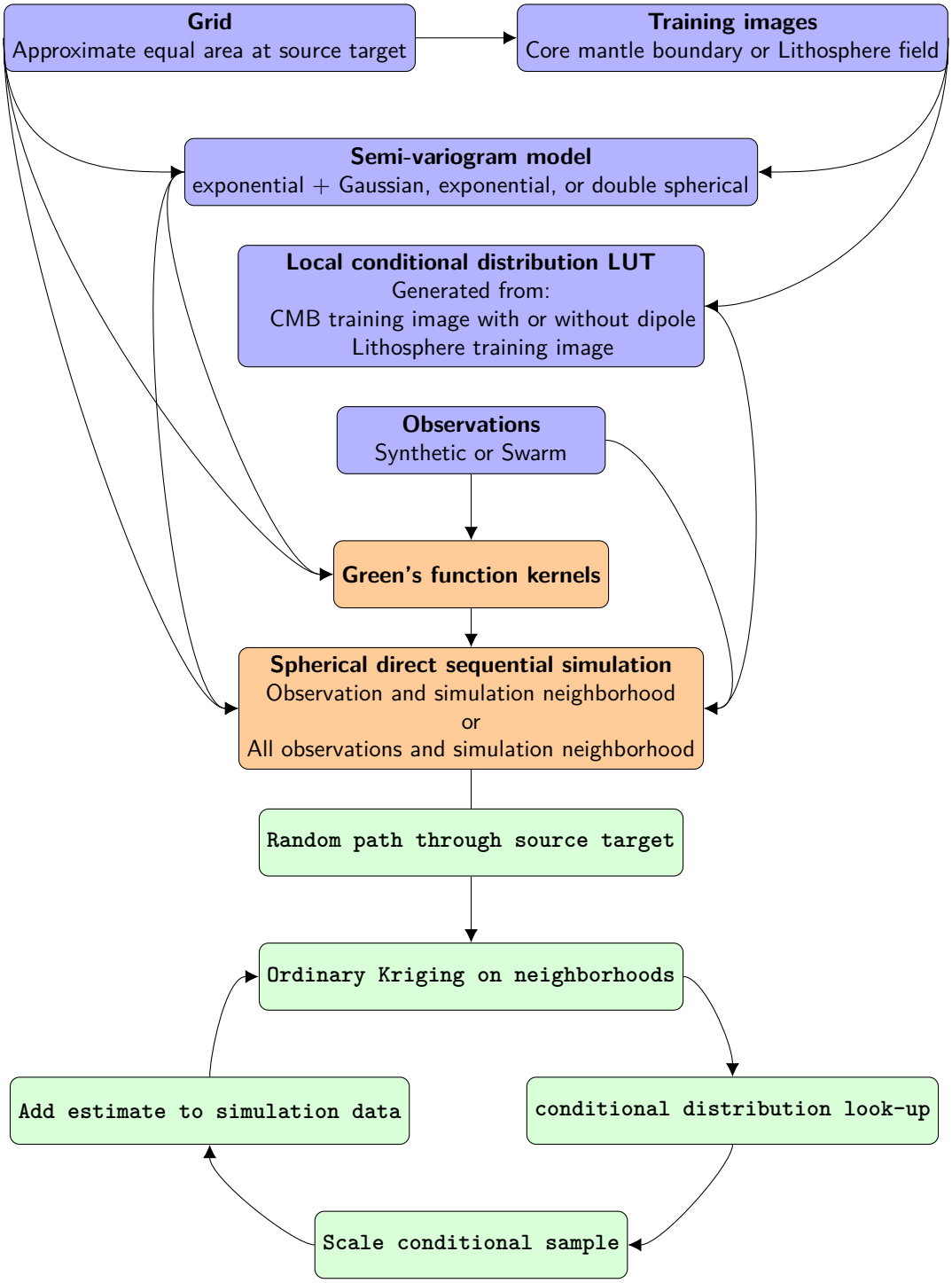


Figure 3.7.1: Flowchart showing the developed spherical direct sequential simulation algorithm structure.

The theory in chapter 2 and the data structures described in this chapter are implemented in a Python tool of my own development. It is inspired by the Matlab toolbox SIPPI (Hansen and Mosegaard, 2013a), which includes VISIM (Hansen and Mosegaard, 2008), and underlying GSLIB structures (Deutsch and Journel, 1998). I currently refer to this tool as SDSSIM, for spherical direct sequential simulation, and it is planned to be made publicly available shortly after the conclusion of the defense of this thesis.
The interaction of structures leading into SDSSIM and the base SDSSIM algorithm is seen in figure 3.7.1. Arrows indicate where each data structure is used. The algorithm follows the process of direct sequential

simulation with histogram reproduction as described with greater detail in section 2.3.2, and does so by implementation of Green's function kernels in an ordinary Kriging system as described in sections 2.2.2 and 2.3.3. Altering the SDSSIM algorithm allow for investigations into certain aspects of direct sequential simulation. Two alternate systems have been used in testing during next chapter, I describe these below.

3.7.1 Algorithm for stochastic realizations of the prior

Stochastic realizations of the prior can be generated by only considering the simulation neighborhood. Through this method, it is possible to determine whether the prior conditioning is working as intended, as random realizations should be generated whose mean is centered on the target semi-variogram and histogram.
The used ordinary Kriging system setup changes such that there is no reliance on Green's functions, only the semi-variogram model. The ordinary Kriging system for this case, as well as Kriging mean and variance, is shown in equation 3.7.1.

$$\begin{aligned} \boldsymbol{K\lambda} = \boldsymbol{k} \rightarrow \begin{bmatrix} \boldsymbol{C_S} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega_S} \\ \boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c_S} \\ 1 \end{bmatrix} \\ \mu_K = \boldsymbol{\omega_S^T} \hat{\boldsymbol{B}}_{\boldsymbol{r}}(\boldsymbol{r'_{ts}}) \\ \sigma_K^2 = \sigma_{exp}^2 - \boldsymbol{\omega_S^T c_S} - \boldsymbol{\Lambda} \end{aligned} \tag{3.7.1}$$

Figure 3.7.2 shows the algorithm, which is identical to the general system, except for the only neighborhood under consideration being previously simulated values (the source neighborhood).

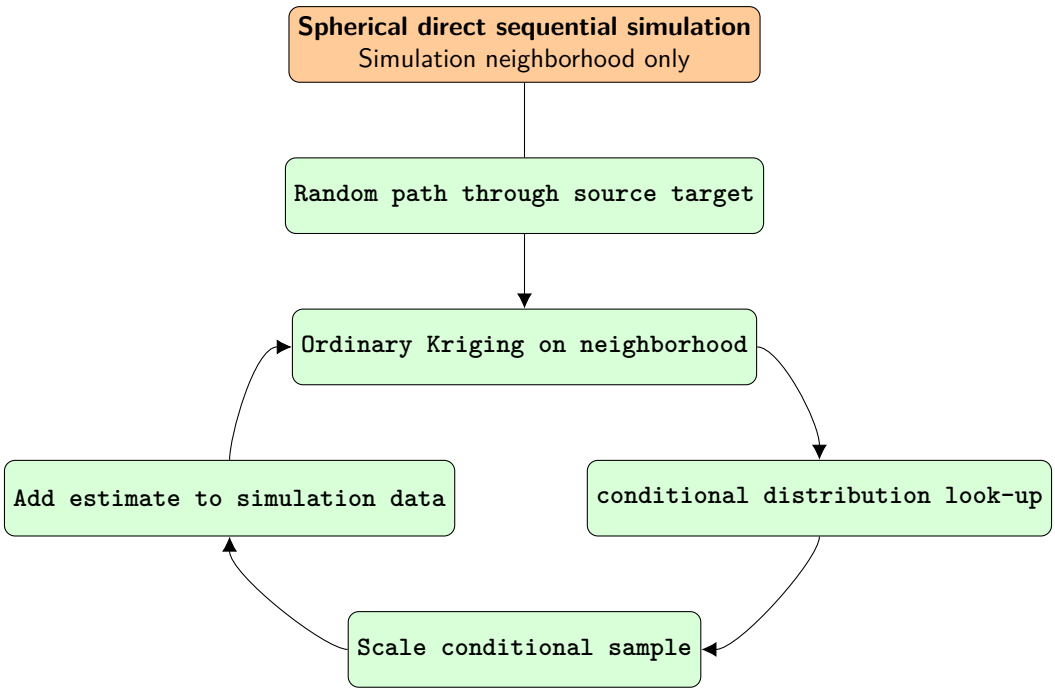


Figure 3.7.2: Flowchart showing the algorithm when generating stochastic realizations with SDSSIM.

3.7.2 Algorithm for sequential least squares estimation

A least squares estimate of available observations is also possible through direct sequential simulation. In this case, only observations are considered. This leads to the ordinary Kriging system shown in equation 3.7.2, where the Kriging mean and variance is also given.

$$\begin{aligned} \boldsymbol{K\lambda} = \boldsymbol{k} \rightarrow \begin{bmatrix} \boldsymbol{C_{obs}} + \boldsymbol{C_E} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega_{obs}} \\ \boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c_{obs}} \\ 1 \end{bmatrix} \\ \mu_K = \boldsymbol{\omega_{obs}^T B_k(r)} \\ \sigma_K^2 = \sigma_{exp}^2 - \boldsymbol{\omega_{obs}^T c_{obs}} - \boldsymbol{\Lambda} \end{aligned} \tag{3.7.2}$$

Figure 3.7.2 shows the algorithm, which removes the conditional look-up and scaling of the general system. Note that the path in this case doesn't need to be random, as each estimate will always be based on the same observation neighborhood (if not all the observations).

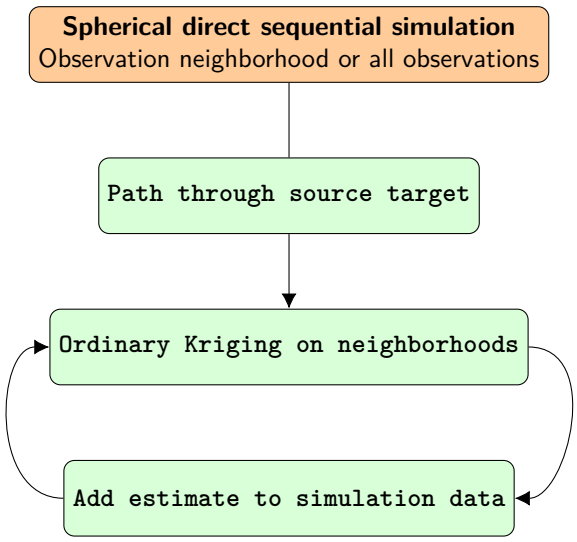


Figure 3.7.3: Flowchart showing the algorithm when generating sequential least squares estimations with SDSSIM.

3.7.3 Solving the ordinary Kriging system in Python

A large part of the algorithm is solving the ordinary Kriging system well and efficiently. The possibilities available in Python that are also applicable to the implemented system is direct inversion, LU-decomposition, or least squares through singular-value decomposition (SVD). The ill-conditioned nature of the ordinary Kriging system has led to the use of least squares through singular-value decomposition. This is a slower approach than LU-decomposition, but less prone to accumulating errors. In solving the ordinary Kriging system, the system is found to have full column rank in all tested cases, indicating an exact solution to the linear system.
In Python it is possible to use accelerated libraries designed for Nvidia type graphics processing units (GPUs). This is currently only available for the LU-decomposition system solver, but is expected to be released for least squares SVD. This may be a way to generate faster realizations in the future.

Chapter 4

Tests and Results

I now present the results of implementing the data described in chapter 3 with the theory described in chapter 2. This is an attempt to give a probabilistic description of the geomagnetic vector field, as observed by satellites, at source target locations. The two source target locations under consideration are the core mantle boundary and the lithosphere at Earth's surface. Due to time limitations the lithospheric field is only considered for stochastic realizations of the prior pdf, while the core mantle boundary field is implemented in full such that posterior realizations using synthetic and real satellite observations are generated. In all cases, the values used to set up the Kriging system are part of the available observations and previously simulated values. I denote the amount of values used, as the observation and source neighborhood respectively. The use of these neighborhoods is due to computational/time constraints in solving large systems of linear equations. Formally, all observations and previously simulated values should be used, but neighborhoods of correct shape and size may still yield good approximations as shown by Hansen and Mosegaard (2008). The neighborhoods are chosen by ordering of the values with respect to the covariance model and Green's function for the source and observations respectively. The size of the source target grid and observation grids will be investigated in the tests described below. Extensive testing of the implementation with available computer resources, has shown that a source grid of 5,000 and observation grid of 2,998 (no poles) locations, are reasonable with respect to computation time when larger numbers of realizations are required.
The chapter starts with two preliminary sections, the first testing the stochastic nature of sampling the prior pdf described by the training images, and the second testing reproduction of synthetic observations unconditional to previously simulated values. Once these capabilities are demonstrated, the rest of the chapter focus on the possibilities of generating posterior realizations reproducing the prior training image statistics given the observational data. In the following, note that I often refer to the core mantle boundary field training image and it's histogram as either including or excluding the dipole. This is not strictly a correct description. The training image in the excluding case has had the latitudinal mean removed, which corresponds to more than just removing the dipole, but I still refer to it as such for simplicity, and use the terms interchangeably. In table 4.0.1, I give an overview of the parameters that have been tested in each section dealing with synthetic observations (or none) of this chapter.

Test simulations overview

prior	observations	prior + observations
CMB	**CMB**	**CMB**
Source neighborhood	Observation neighborhood	Neighborhoods
	Smooth LSQ	Grid size
Lithosphere		
Source neighborhood		

Table 4.0.1: Parameters tested in this chapter.

4.1 Diagnostics description

Before showing results I here give a general overview of the test diagnostics used throughout this chapter, to ensure clarity. Figure 4.1.1 is a typical plot of relevant information in regards to the success of direct sequential simulation.

The upper left plot is an observation reproduction histogram. This is a histogram of the target, predictions, and the mean of the predictions. The target (black line) is the histogram of synthetic or real observations at satellite altitude, depending on which is used. The predictions (grey lines) are histograms of each posterior realization at satellite altitude, as computed by the forward problem described in equation 2.1.8. Finally, the mean (red dotted line) is the mean of the predictions computed for each observation location. E.g. for 100 realizations there will be, at any given observation location, an associated observation value and 100 predicted values. The mean is the mean of these 100 values at each observation location, plotted as a histogram. In addition, statistics parameters are shown for the target (observations) and the mean (prediction mean). This plot is expected to show the posterior realizations as a spread around the target, with the mean approaching a fit of the target. For infinite realizations this should be reflected by identical values in the statistics overview of observations and prediction mean.

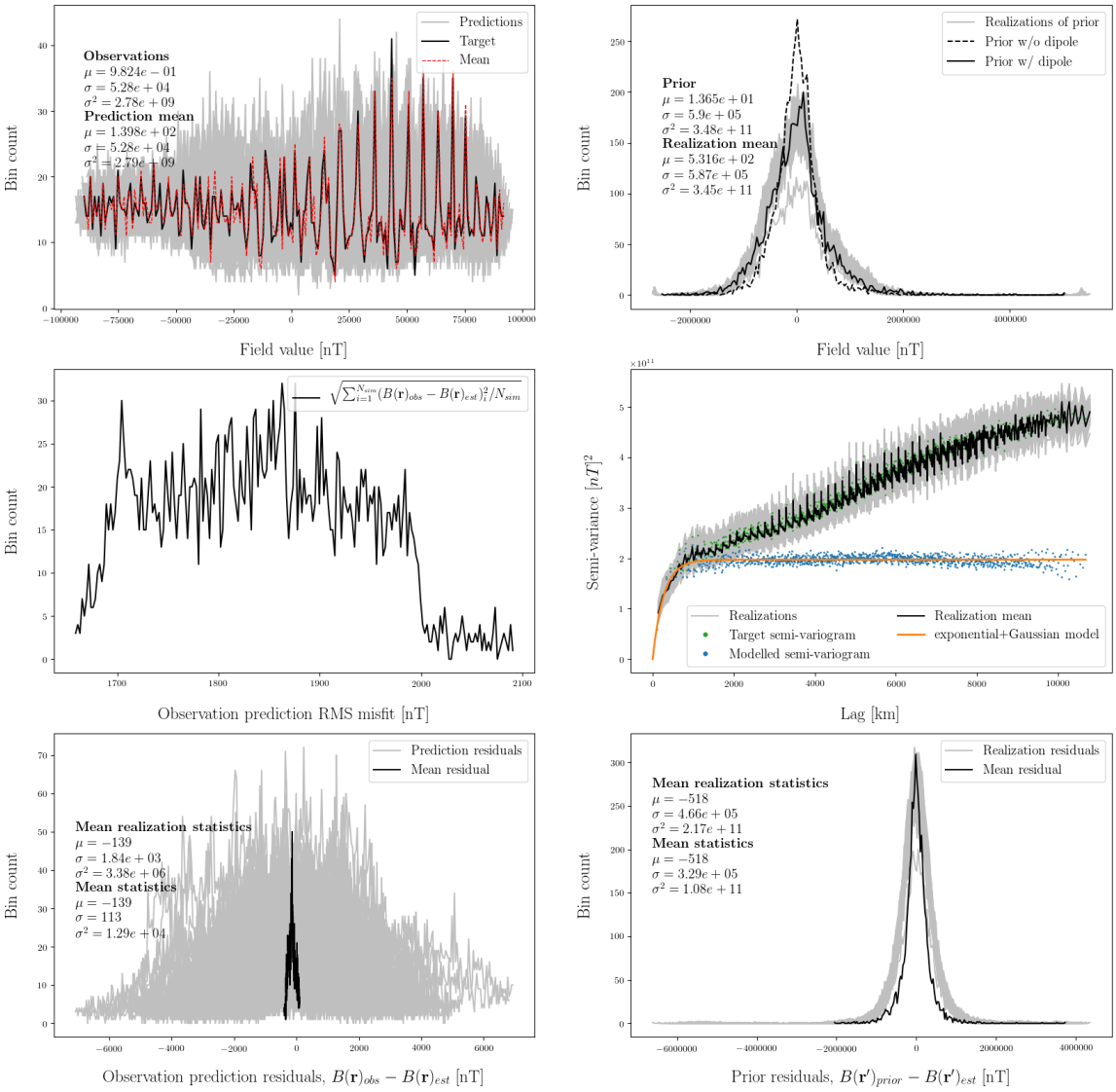
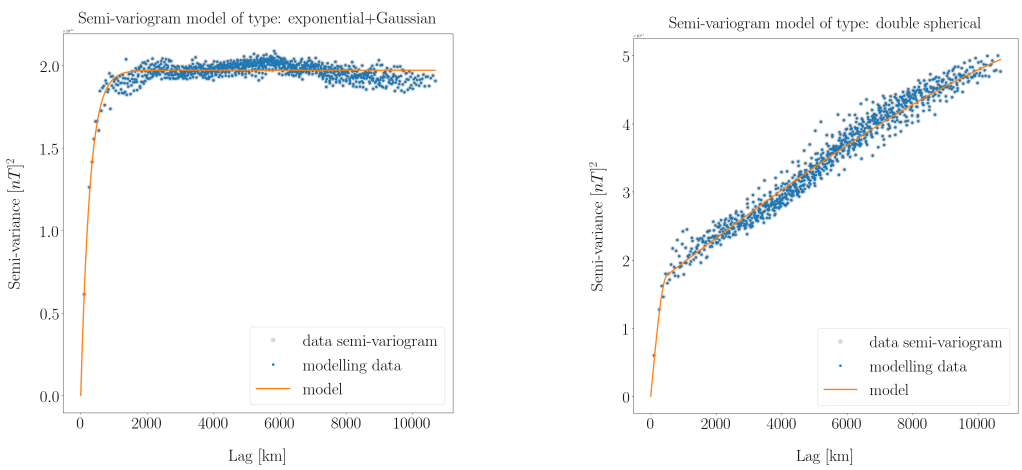


Figure 4.1.1: Diagnostics example of direct sequential simulation result. Depicted are an observation reproduction histogram (upper left), a CMB estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

The upper right plot is a similar histogram plot, but for the source target, i.e. the CMB or lithosphere field, using the training image histograms. In this case the shown plot is for the CMB and includes both the histogram for the training image with and without dipole. This plot has the same expectation, the realizations should be a spread around the target, with the target depending on used prior and test type. In general, when realizations are conditional to observations, this data will condition toward the prior with dipole, as the dipole is part of the observation values. However, the target histogram used to generate the local conditional distributions, will be conditioned toward by the previously simulated values. For unconditional realizations, the target is always the histogram used in generating the local conditional distributions. The middle left plot is a histogram of observation prediction root-mean-square misfit. The RMS value is computed for each collection of realization values (amount N_{sim}) at each observation location. This is an estimate of overall spread for each mean prediction value. Lower values indicate sharper posterior probability density function.
The middle right plot show various relevant semi-variograms depending on the semi-variogram model used. Similarly to the other plots, the posterior realization semi-variograms are shown in grey with their mean value in black. In addition to this, the semi-variogram model used for the simulation is shown in orange, along with the data upon which it is based in blue. Finally, in the case shown here, the target semi-variogram is known from the training image and is shown in green. This is the semi-variogram which the posterior realizations are expected to reproduce from conditioning to the synthetic observations. Knowing this allow tests to be carried out giving an indication of how small an observation neighborhood is possible, without losing reproduction of large scale variability structure. This is crucial before using real observations, as no true semi-variogram is known at the source target in that case, and small neighborhoods are desired in order to carry out many realizations. In addition, note the small scale fit to the model semi-variogram. This fit is of interest as it shows impact of the prior knowledge, which should be conditioning the posterior realization to follow the model semi-variogram as far as the observations allow it. In this synthetic case, it is a very close fit since the observations and prior have the same origin. The final two plots at the bottom left and right are both histograms of residuals. The left histogram is the residuals of prediction values at satellite altitude, with respect to the used observations. The right histogram is the residuals of estimated values at the source target, with respect to the prior training image reference. Again the mean is computed for each collection of predictions/estimates, at the respective observation and source target location. The *mean realization statistics* show the statistics of the residuals as one big cluster, with the *mean statistics* being for the mean residual alone (black line). The intention for the observation prediction residual plot is to give a closer look at the mean fit to the target, as this is harder to see from simulation to simulation in the field prediction histogram alone. In contrast, the prior residuals should indicate a general fit to the prior without getting too close, as this would indicate over-conditioning to the training image.
Figure 4.1.2 show the two semi-variogram models used to compute observation conditional results. One is modelled from the CMB without dipole and the other with the dipole. While both are technically stationary covariance functions through equation 3.3.3, the double spherical model has range beyond the longest distance between two locations on the CMB grid, i.e. stationarity is never reached.



(a) SV for CMB w/o dipole

(b) SV for CMB w/ dipole

Figure 4.1.2

4.2 Sampling the prior

In order to ensure correct implementation of direct sequential simulation, unconditional realizations of the prior training image should produce stochastic realizations reproducing the target statistics (Journel, 1994). In the following, I test this capability for a 5,000 location source grid, $N_S = 5000$, while varying the source neighborhood size, N_{nsv}. Ideally, realizations should be generated until the mean converges to the target, and longer stochastic realizations not shown here confirm that the mean converges in the current implementation. However, these tests express the stochastic results for simulation sizes equivalent to the full simulations to be run, which due to time constraints are 100 realizations.
For the core mantle boundary field, the effect of training image choice with or without dipole in generating the local conditional distributions have been investigated. The differences in the histograms obtained with and without the dipole are shown on figure 4.2.1. Stochastic realizations using the prior histogram without the dipole is included in this section, with the other case available in appendix D.1.1. Table 4.2.1 gives an overview of the testing parameters. Note that the source neighborhoods are defined as a fraction of the total grid. Given the use of an approximate equal area grid, this is a rough measure of the source neighborhood area, which becomes more accurate towards the end of a simulation as more simulated values are available.

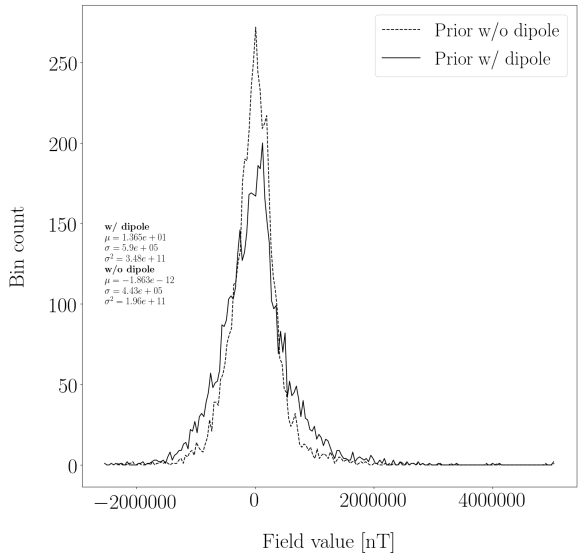


Figure 4.2.1: Histogram of the CMB field prior training image with and without the dipole removed

Test overview for sampling the prior

Source target: Core mantle boundary field		Source target: Lithospheric field	
Source neighborhood		**Source neighborhood**	
CMB grid size	$N_S = 5000$	Lithosphere grid size	$N_S = 5000$
N_{nsv}		N_{nsv}	
A_1	$N_S/1000$	B_1	$N_S/1000$
A_2	$N_S/500$	B_2	$N_S/500$
A_3	$N_S/100$	B_3	$N_S/100$
A_4	$N_S/50$	B_4	$N_S/50$
Semi-variogram type	exp + Gau	Semi-variogram type	exp
Realizations	100	Realizations	100

Table 4.2.1: Parameters used and under consideration for direct sequential simulation of priors.

4.2.1 CMB field with dipole removed

Four stochastic simulations are run with increasing neighborhood size, for a core mantle boundary training image with the dipole removed. Figure 4.2.2 and the table below it show the statistics for each simulation, and figure 4.2.3 shows samples of the realizations. The realizations show a very uniform fit to the semi-variogram model for varying neighborhood sizes. In addition, the histogram fit clearly realized the target w/o dipole as expected, although with some outliers of smaller peaks across the tests. The mean fit looks to be distributed around zero with values up to $\pm 3000\,\mathrm{nT}$, and the standard deviation fit is improving from the smallest neighborhood to the three larger, with no conclusive improvement between those. The large mean fit distribution is an indication of more realizations being required to run a full simulation. The sample realizations appear stochastic across tests and resemble the structure of the prior. As expected, computation time increases with a larger neighborhood size.

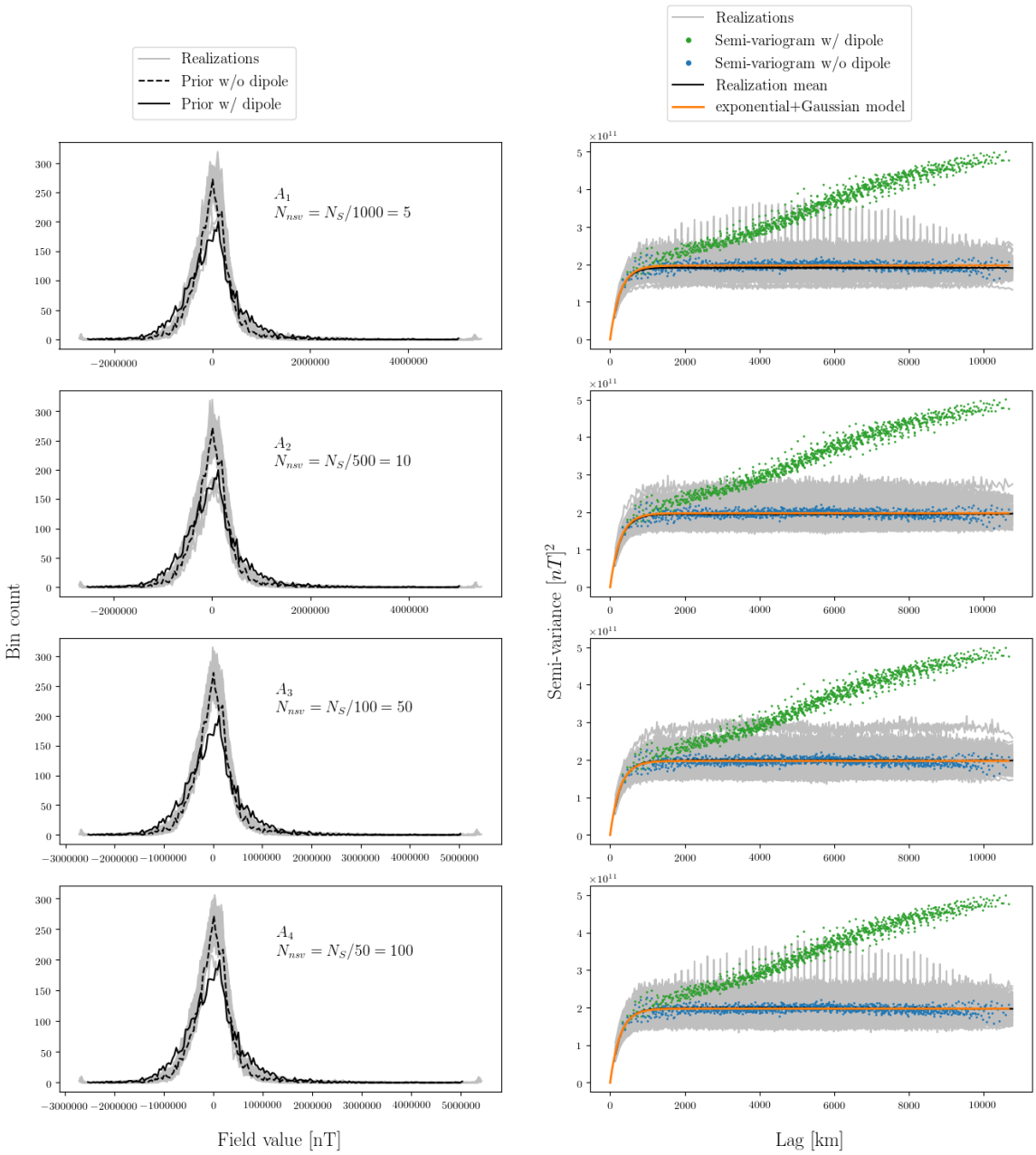
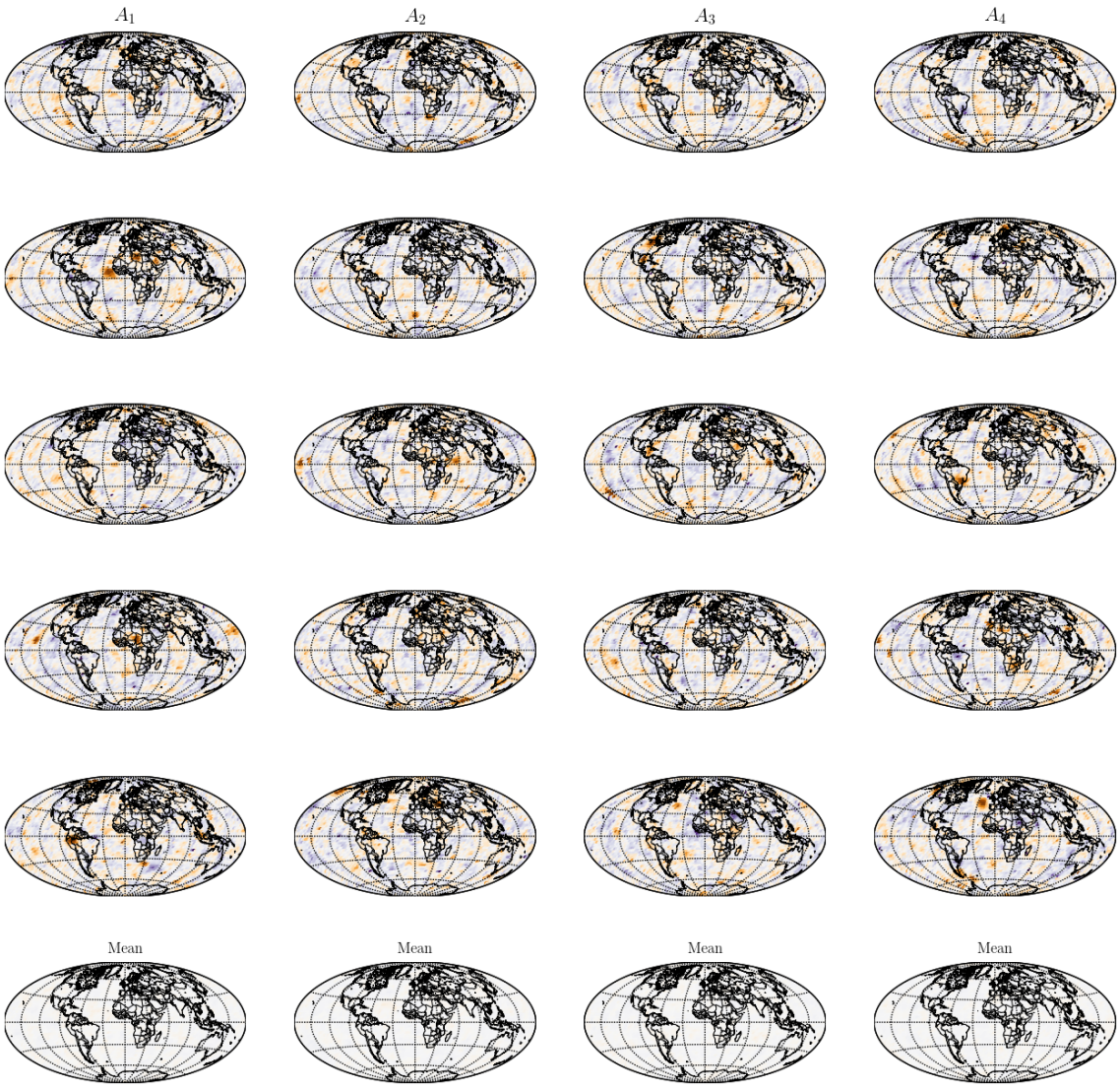
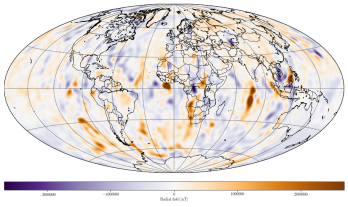


Figure 4.2.2: Stochastic simulation realizations and their statistics reproduction of the dipole removed CMB training image. Source grid size is $N_S = 5000$ and each simulation consists of 100 realizations.

Test	**Mean** $[nT]$	**Std.dev.** $[nT]$	**Compute time** $[hrs]$
A_1	$1.644e+03$	$4.37e+05$	0.26
A_2	$-3.003e+03$	$4.42e+05$	0.265
A_3	$2.996e+03$	$4.45e+05$	0.38
A_4	$-2.963e+03$	$4.45e+05$	0.65
Target	$-1.863e-12$	$4.43e+05$	N/A



(a) Realization examples



(b) Prior reference

Figure 4.2.3: **(a)** Sample realizations for each simulation run. **(b)** CMB training image used to generate local conditional distributions.

4.2.2 Lithosphere field at Earth's surface

Testing of the source neighborhood size for the lithosphere field is also carried out with four stochastic simulations of increasing neighborhood size. Figure 4.2.4 and the table below it show the statistics fit for each simulation, and figure 4.2.5 show samples of the realizations. It is seen that increasing the neighborhood size improves the semi-variogram fit, however the target histogram fits do not reach a good reproduction of statistics. This is most clear from the standard deviation of the realizations, which improve toward the target with increasing neighborhood, but never reach it. The mean fit is off by an order of magnitude for all tests, with no notion of ordered approach to the target, this is similar to the previous test. For the sample realizations, while some do appear to reproduce the prior structure, there is a tendency of the global mean being either positive or negative compared to the prior reference. Further tests have shown that the statistics reproduction improves for much larger grids. However, due to the system size requirements following this increase in size, no attempt has been made to use synthetic observations with the lithosphere. As such, further tests have been omitted.

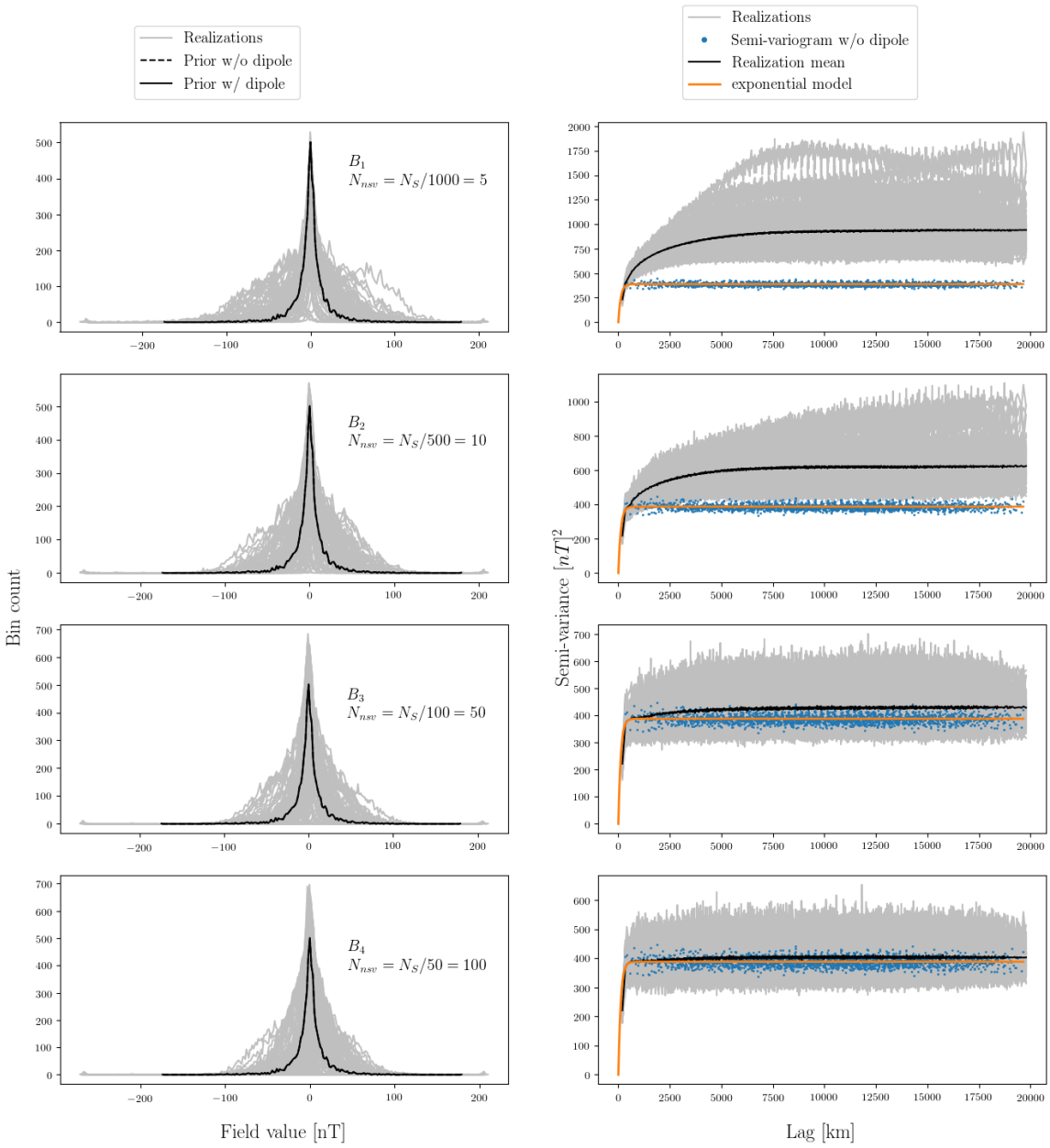
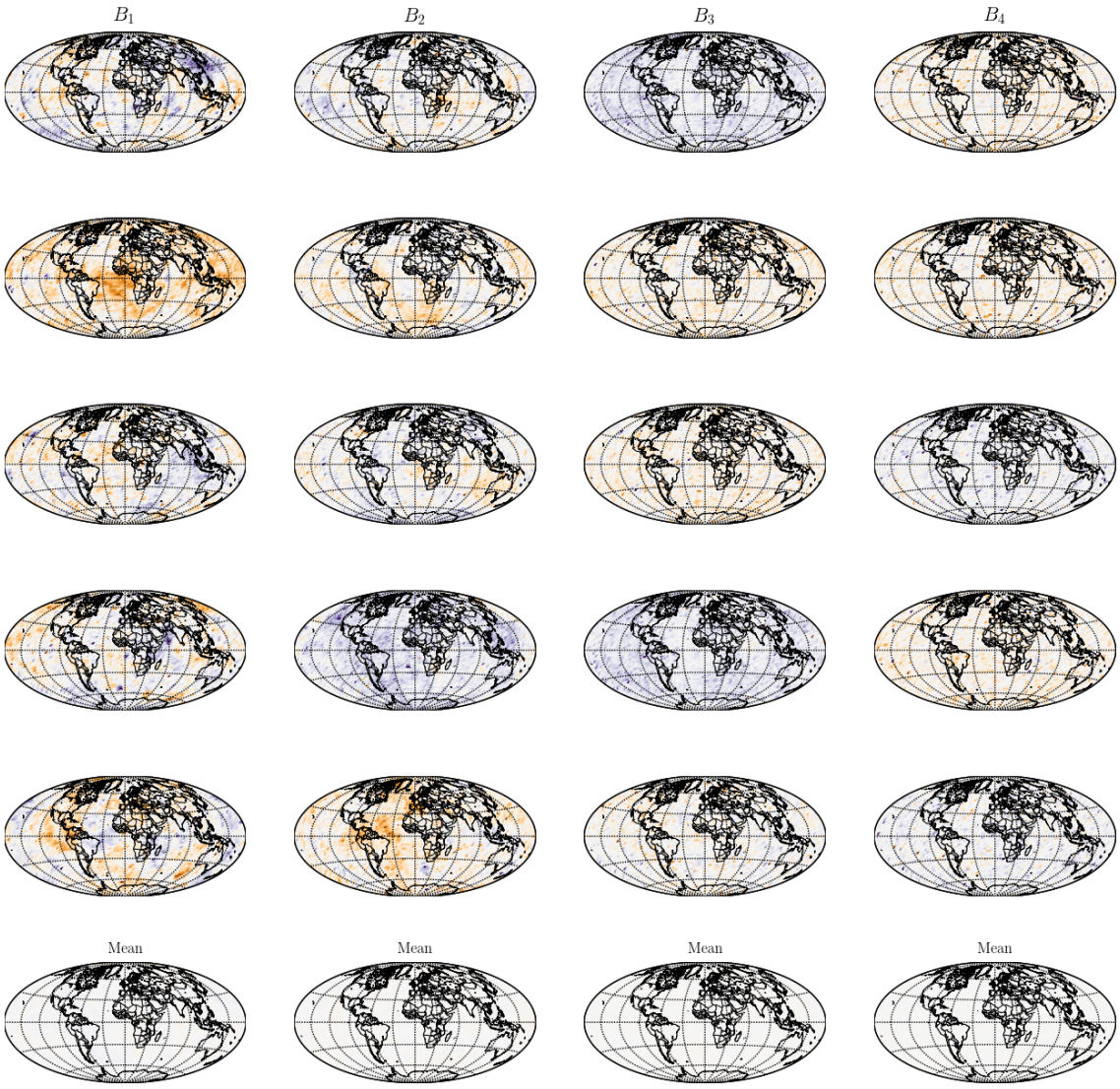
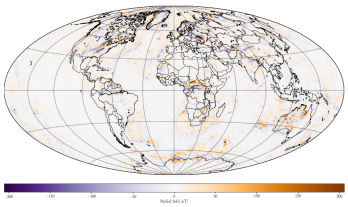


Figure 4.2.4: Stochastic simulation realizations and their statistics reproduction of the lithosphere training image. Source grid size is $N_S = 5000$ and each simulation consists of 100 realizations.

Test	**Mean** $[nT]$	**Std.dev.** $[nT]$	**Compute time** $[hrs]$
B_1	$-5.918e-01$	36.2	0.24
B_2	$6.541e-01$	31.1	0.245
B_3	$8.868e-01$	27.8	0.37
B_4	$4.444e-01$	27.7	0.66
Target	$-2.293e-02$	19.7	N/A



(a) Realizations



(b) Prior reference

Figure 4.2.5: **(a)** Sample realizations for each simulation run. **(b)** Lithosphere training image used to generate local conditional distributions.

4.3 Reproducing synthetic satellite observations

Here I describe tests showing the direct sequential simulation algorithm capability of reproducing magnetic field observations at satellite altitude. An overview of the tests carried out is found in table 4.3.1. In section 4.3.1, sizes of observation neighborhoods and their influence on ability to reproduce the training image statistics are scrutinized. Finally, in section 4.3.2 it is tested whether fitting the observations in a smooth least squares (LSQ) sense is possible using DSSIM, followed by an attempt at LSQ fitting observations with an approximate global coverage (AGC) neighborhood in section 4.3.3. In appendix D.2.1, an extra simulation of 1000 realizations is shown conditional to all synthetic observations. This test shows that there is convergence progress toward observation fit for increased amounts of realizations. In all the tests presented here, an observation error estimate of $1nT$ has been added to the ordinary Kriging system, through the error covariance matrix described in section 2.3.3.

Test overview for reproducing observations by estimation of the CMB field

Grid size, $N_S = 5000$. Total observations, $N_{obs} = 9998$

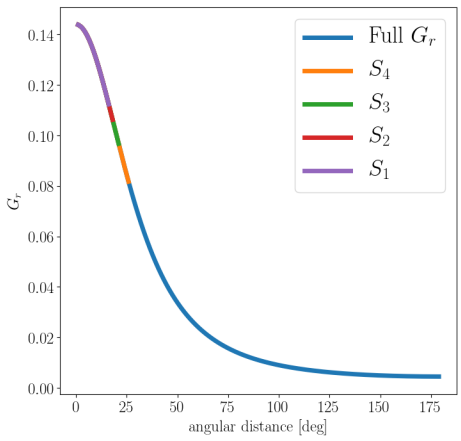
D.2.1 All synthetic observations		**4.3.2 LSQ**	
Semi-variogram type	exp + Gau	Semi-variogram type	exp + Gau
Realizations	1000	Realizations	1
Neighborhood size	all	Neighborhood size	all
Conditional dist. type	no dipole	Conditional dist. type	no dipole
4.3.1 Obs. neighborhood influence		**4.3.3 AGC LSQ**	
Neighborhood size		Semi-variogram type	exp + Gau
S_{01}	$N_{obs}/100$	Realizations	1
S_{02}	$N_{obs}/90$	Neighborhood size	$N_{obs}/3$
S_{03}	$N_{obs}/80$	Conditional dist. type	no dipole
S_{04}	$N_{obs}/70$		
S_{05}	$N_{obs}/60$		
S_1	$N_{obs}/50$		
S_2	$N_{obs}/40$		
S_3	$N_{obs}/30$		
S_4	$N_{obs}/20$		
Semi-variogram type	exp + Gau		
Realizations	100		
Conditional dist. type	no dipole		

Table 4.3.1: Parameters under consideration for direct sequential simulation of synethetic observations.

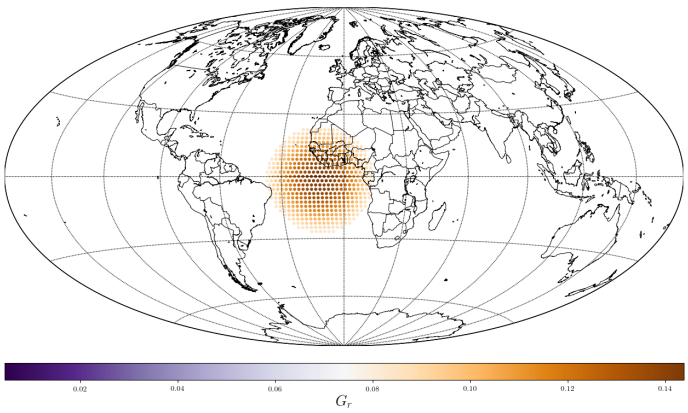
4.3.1 Influence of the chosen observation neighborhood

First I investigate whether it is feasible to reproduce the target statistics using less than global observation coverage. In this test, the range of observation neighborhoods investigated are from one hundredth of the total available observations, to one twentieth. I distinguish the observation neighborhoods through a naming convention as given in table 4.3.1, with S_{0X} denoting neighborhoods generally overestimating, and S_1 to S_4 generally underestimating the semi-variogram fit.
The neighborhoods can also be expressed as a fraction of the radial Green's function, previously defined in equation 2.1.2. This is most easily shown as a function of the angular distance. Neighborhood sizes S_1 to S_4 is plotted according to angular distance on figure 4.3.1 along with an example of the neighborhood surface grid size in test S_4. The training image statistics fit for the tests designated S_1 to S_4 are seen on figure 4.3.2 and the table below it, with tests S_{01} to S_{05} on figure 4.3.3 and the table below that. Observation reproduction and residuals for tests S_1 to S_4 are found on figure 4.3.4. The histogram mean is generally close to the target, with the largest neighborhood, S_4, being closest. The computational

time is naturally rising with neighborhood size, reaching $4.5 - 7$ hours for the 100 realizations in S_3 and S_4. The smaller neighborhood taking longer time is due to different CPU's used. In all test cases (S_1 to S_4 and S_{01} to S_{05}) the histogram shape follows the prior training image with the dipole included, this is despite the semi-variogram model and local conditional distributions being based on the training image without the dipole. Clearly this information is coming directly from the observations, showing the information they provide. From tests S_1 to S_4 the semi-variogram and standard deviation fit appear to change from slightly underestimating, to underestimating more, before finally estimating the semi-variogram with dipole. From the shape of the Green's function alone, this is unexpected as the function steadily decreases, suggesting an improving estimate with larger neighborhoods as structures further away are accounted for. The smaller neighborhoods of tests S_{01} to S_{05} show the estimation leading down to the near semi-variogram fit of S_1. Here we see the semi-variogram fit over-estimating for very small neighborhoods such as S_{01}, but improving for increasing neighborhoods until it is fitting well at one sixtieth the total surface area in S_{05}. However, note the small scale fit differences between S_{05} and S_4, the larger neighborhood of S_4 is reproducing the model fit at small lags, whereas S_{05} is not. The discrepancy is further seen on the observation reproduction of S_4 at the bottom of figure 4.3.4, and S_{05} in figure 4.3.5. Specifically, when reaching the largest neighborhood of S_4, there is a significant difference in the mean of the residuals becoming smaller as the number of realizations increases, compared to the smaller neighborhoods, except at S_{05}, which appear to converge faster than the surrounding neighborhood sizes, albeit not as sharply as S_4. It seems both neighborhoods may reproduce realizations that are valid in the prior sense, but once the size of neighborhood S_4 is reached, smaller structures are reproduced. Further evidence of this is seen in figure 4.3.6, where slightly sharper features stand out for the realizations and mean in neighborhood S_4.



(a)



(b)

Figure 4.3.1: **(a)** Fraction of Green's function considered for the test neighborhoods $S_1 ... S_4$ with respect to angular distance from the target location. Each colour shows the increase in angular distance considered from the previous test. **(b)** Radial Green's function for the observation neighborhood in test S_4 according to grid locations, the neighborhood is approximately one twentieth the total surface area.

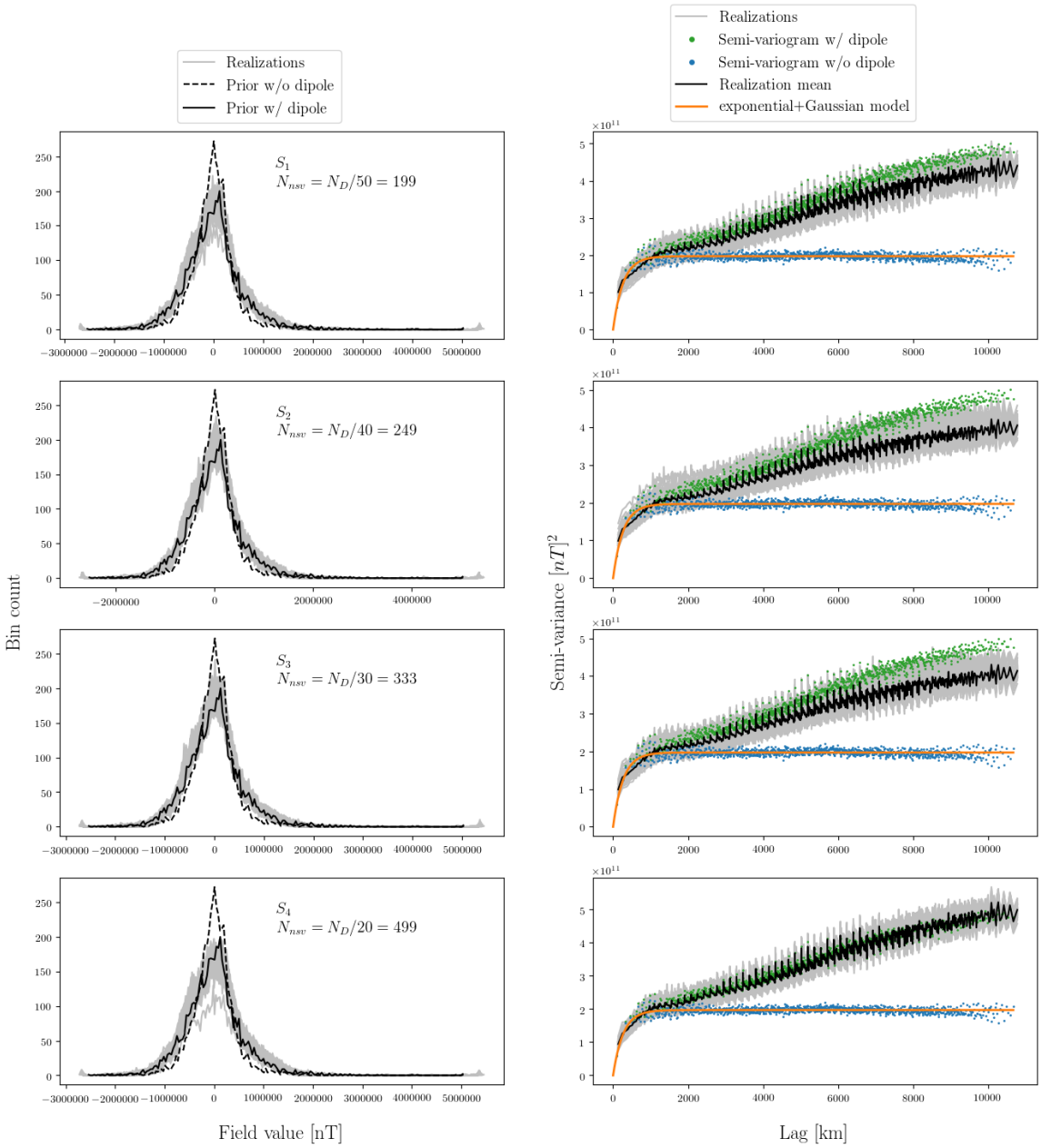


Figure 4.3.2: Test S_1 to S_4 simulation realizations conditional only to available observations and their statistics reproduction of the CMB training image. Source grid size is $N_S = 5000$ and total observations are $N_{obs} = 9998$, each simulation consists of 100 realizations.

Test	**Mean** $[nT]$	**Std.dev.** $[nT]$	**Compute time** $[hrs]$
S_1	-93.67	$5.64e+05$	0.97
S_2	169.5	$5.51e+05$	1.23
S_3	210.2	$5.54e+05$	6.65
S_4	-8.12	$5.86e+05$	4.5
Target	13.65	$5.9e+05$	N/A

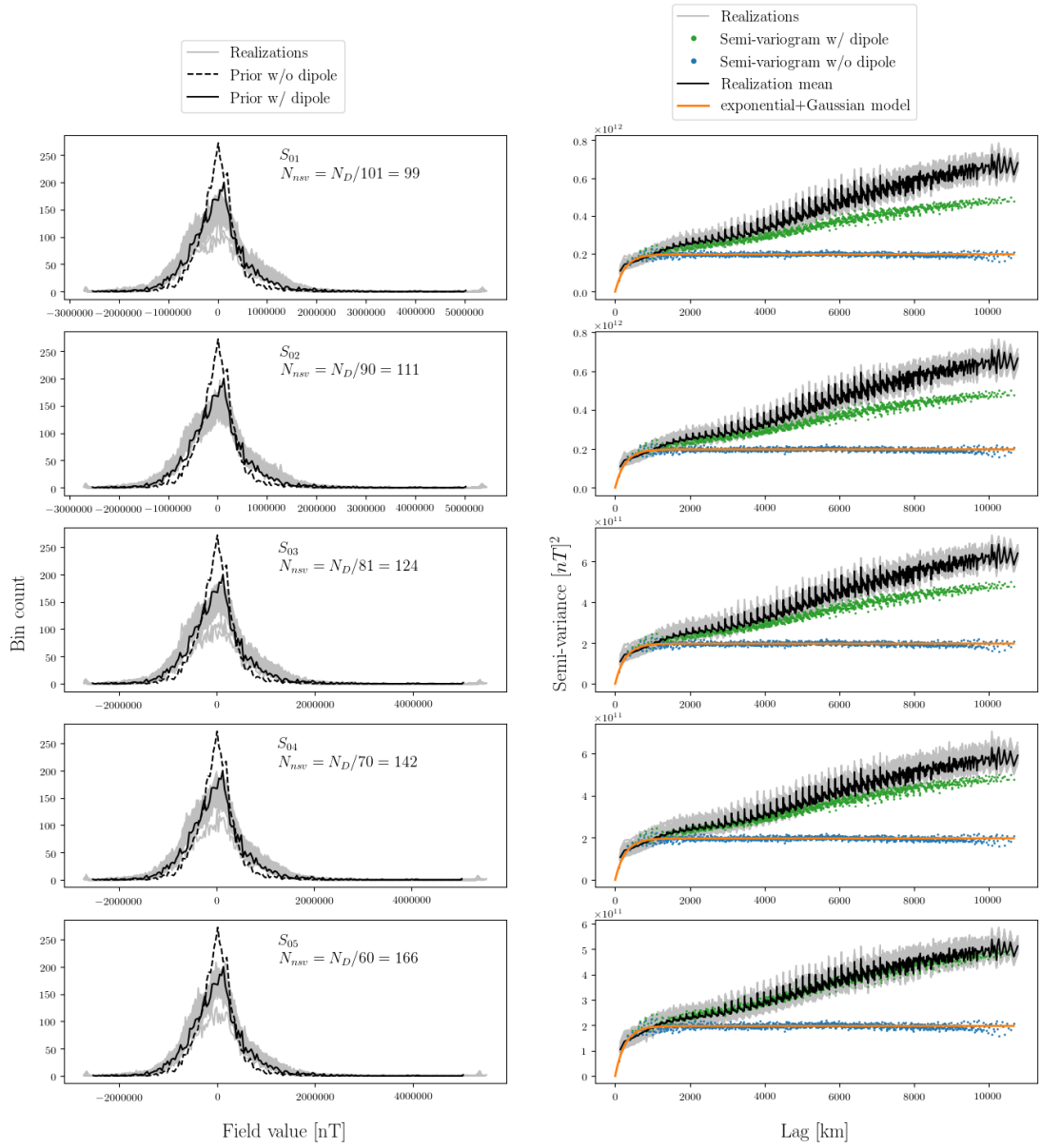


Figure 4.3.3: Test S_{01} to S_{05} simulation realizations conditional only to available observations and their statistics reproduction of the CMB training image. Source grid size is $N_S = 5000$ and total observations are $N_{obs} = 9998$, each simulation consists of 100 realizations.

Test	**Mean** $[nT]$	**Std.dev.** $[nT]$	**Compute time** $[hrs]$
S_{01}	341.7	$6.58e+05$	0.41
S_{02}	698.1	$6.52e+05$	0.44
S_{03}	-51.78	$6.43e+05$	0.49
S_{04}	-12.24	$6.23e+05$	0.57
S_{05}	-765.6	$5.92e+05$	0.67
Target	13.65	$5.9e+05$	N/A

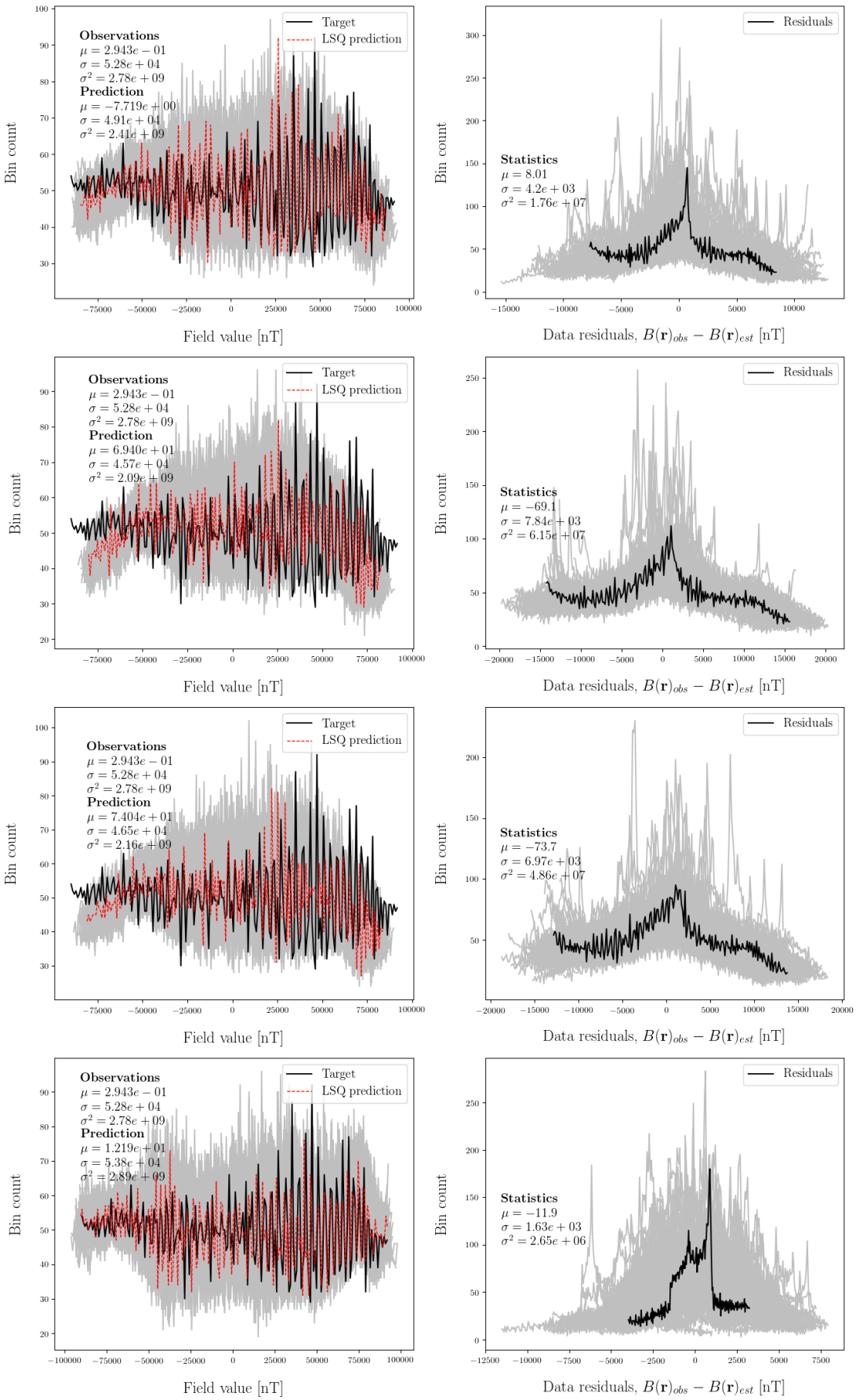


Figure 4.3.4: Observation reproduction and residuals for tests S_1 to S_4, top to bottom.

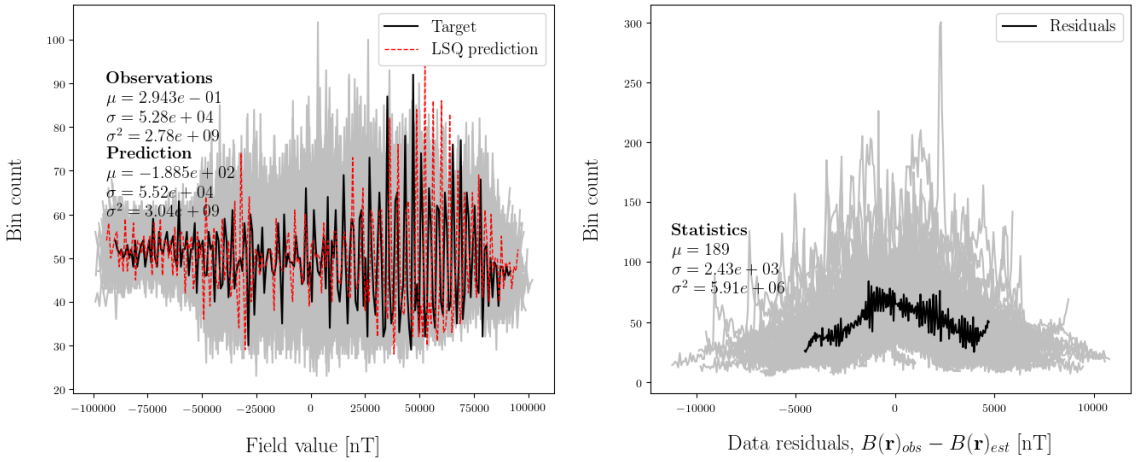
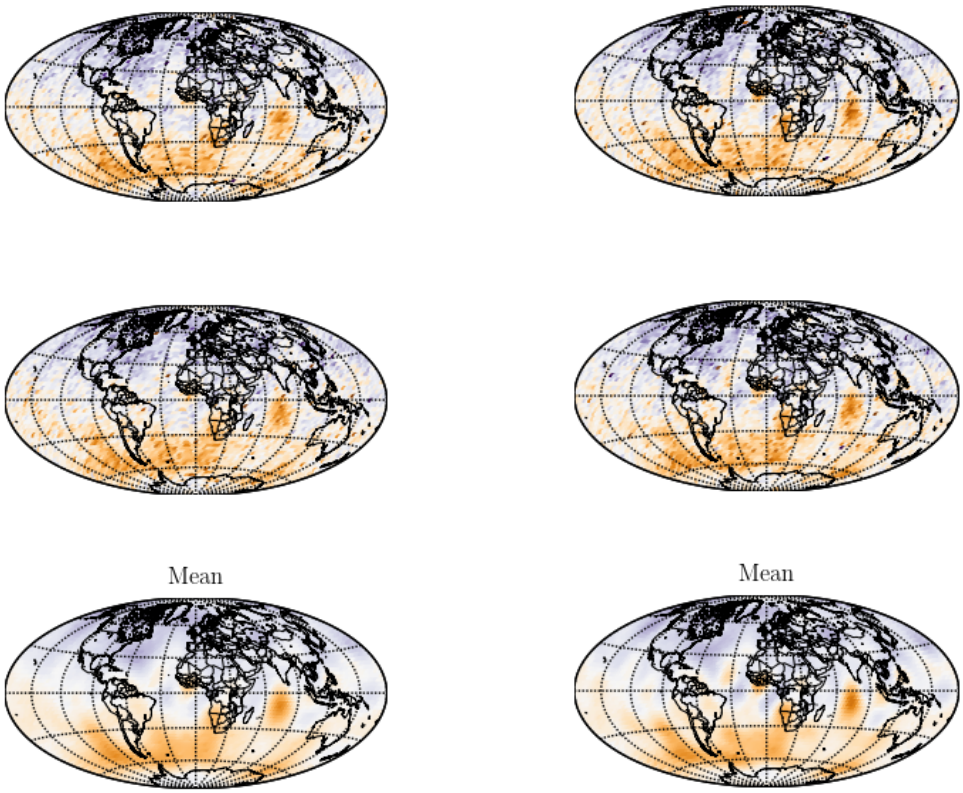


Figure 4.3.5: Observation reproduction and residuals for test S_{05}.



(a) S_{05} **(b)** S_4

Figure 4.3.6: Realization samples and the mean of all realizations for test S_4 and S_{05}.

4.3.2 Smooth least squares solution

The most important part of the simulation is ensuring adherence to the observations. This makes it important to test whether the system is capable of reproducing the observations without conditioning on previously simulated values. For direct sequential simulation, this can be done through the sequential least squares estimation method given by Hansen and Mosegaard (2006). In this implementation, the method is achieved by calculating the ordinary Kriging weights using all observations at each location in the CMB field grid, through solving the linear system in equation 2.3.9. Note that I solve these systems using SVD as described in section 3.7 and that there is no contribution from previously simulated values in

this case. Once the ordinary Kriging weights are computed, the Kriging mean and variance is found from equation 2.3.10, which is used as the estimate. This generates a realization that is equivalent to a smooth least squares solution. The SDSSIM implementation is shown in section 3.7.2. In a probabilistic sense related to conditioning on previously simulated values, this solution represents the most likely value of the local conditional distributions at each target source location. However, note that no information about local distribution shape is retained, this solution is also the most likely when assuming local Gaussian distributions. Figure 4.3.7 shows the observation reproduction and data residuals for such a sequential least squares estimation, and figure 4.3.8 shows the resulting radial core field estimation. The observation reproduction through forward modelling of the estimated core values clearly agree very well, with similar mean and variance as that of the target observations. This is more clearly shown through the data residuals, which are distributed around zero with a spread lower than the added synthetic noise of 1 nT. The estimated radial core field shows a smooth solution as expected.

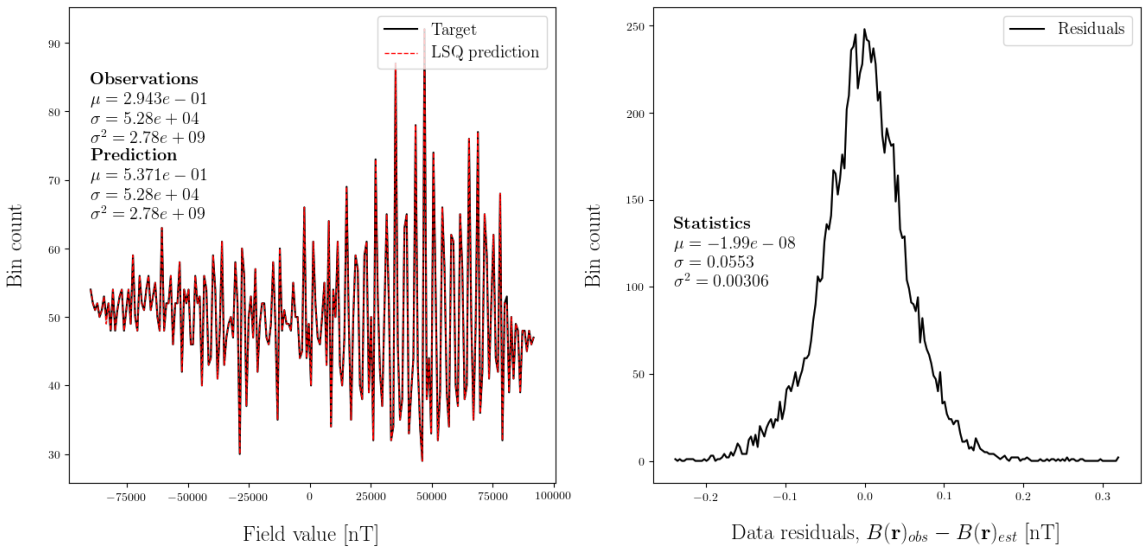


Figure 4.3.7: Observation reproduction and residuals for a sequential least squares estimation using 2,998 synthetic satellite observations at 300 km above Earth's surface with 1 nT noise added.

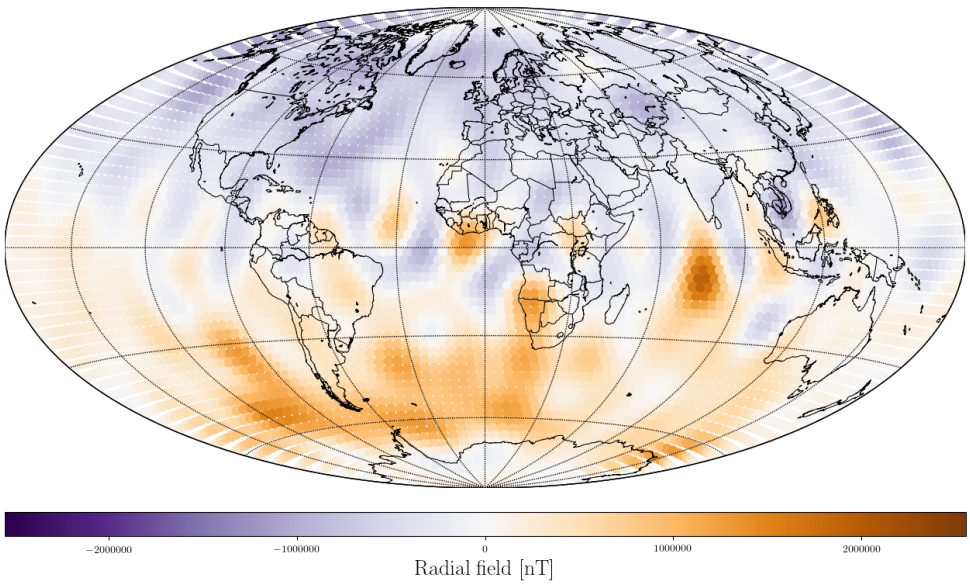


Figure 4.3.8: Sequential least squares estimation of the radial field at the core mantle boundary using 2,998 synthetic satellite observations at 300 km above Earth's surface. The estimated values are determined on an approximate equal area grid of 5,000 locations.

4.3.3 LSQ solution with approximate global coverage neighborhood

It has now been shown that LSQ estimation using all available observations produce an estimate resembling the smooth least squares solution. However, in order to utilize the simulated values such that conditioning increases throughout the simulation, smaller neighborhoods must be used to avoid very large Kriging systems and long computation. An approximate global coverage (AGC) neighborhood for the used observations was devised as an attempt at this. It is a neighborhood centered on the latitude and longitude of the target point at the core mantle boundary, with all nearby values considered out to a defined range, followed by a geometric progression of randomly sampled observations as distance to the target point increases. Figure 4.3.10 shows an example of such a neighborhood. The smooth least squares solution of using the AGC neighborhood is presented on figure 4.3.9 and 4.3.11. A fit within the synthetic data errors of 1 nT is not achieved using this method, however, it approaches reasonable values with a standard deviation of 16.6 nT for the residuals. Further optimizations through better integration approximation may be possible. Note that the CMB field estimation histogram is not reproduced, and neither is the observation semi-variogram. This illustrates that the least squares result, while being the maximum likelihood estimate, is an unlikely sample of the posterior.

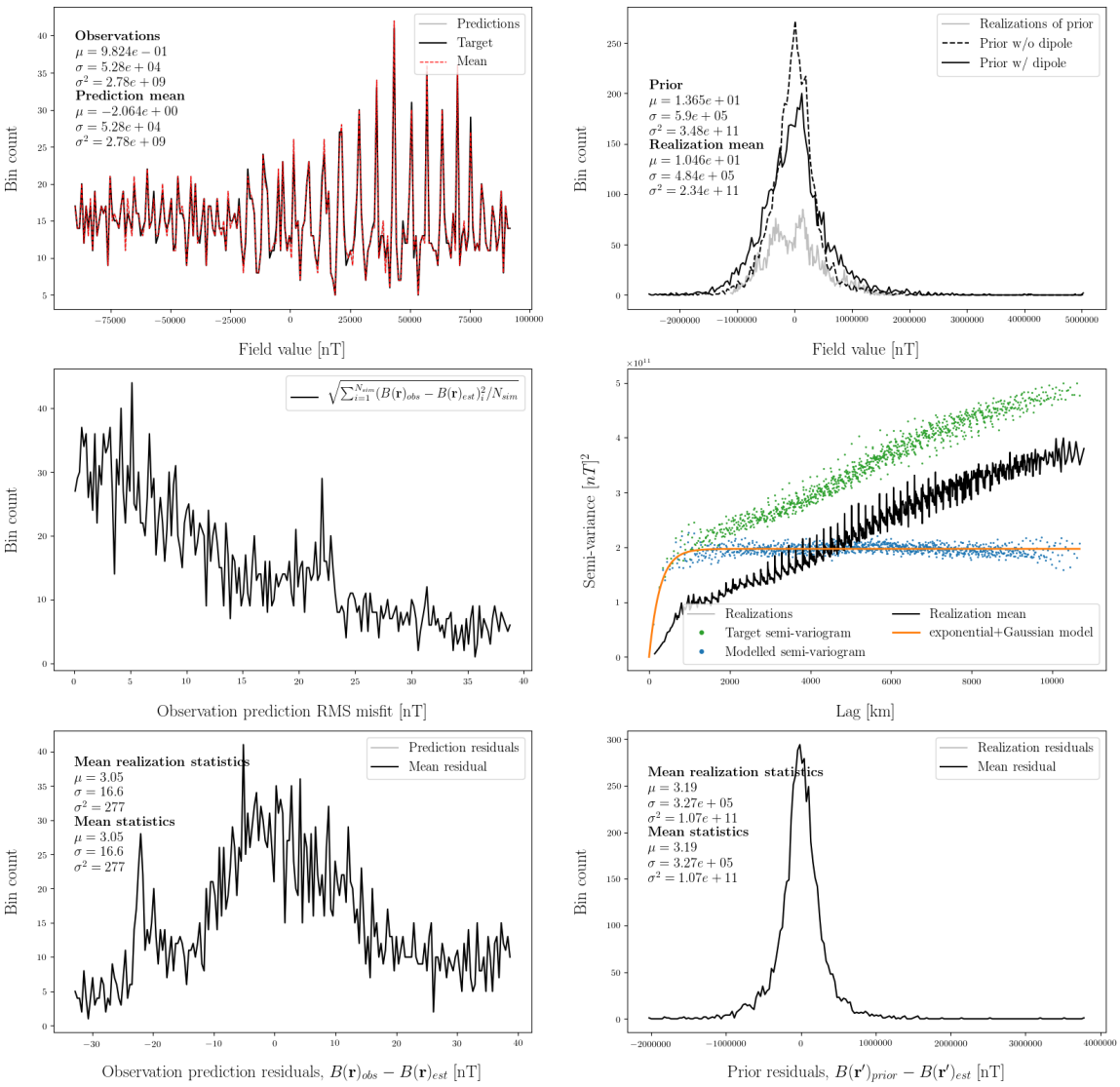


Figure 4.3.9: Diagnostics for the sequential least squares solution using the approximate global coverage neighborhood. Depicted are the observation reproduction histogram (upper left), the CMB field estimation histogram (upper right), the root-mean-square misfit to the observations (middle left), the semi-variogram fit (middle right), the observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

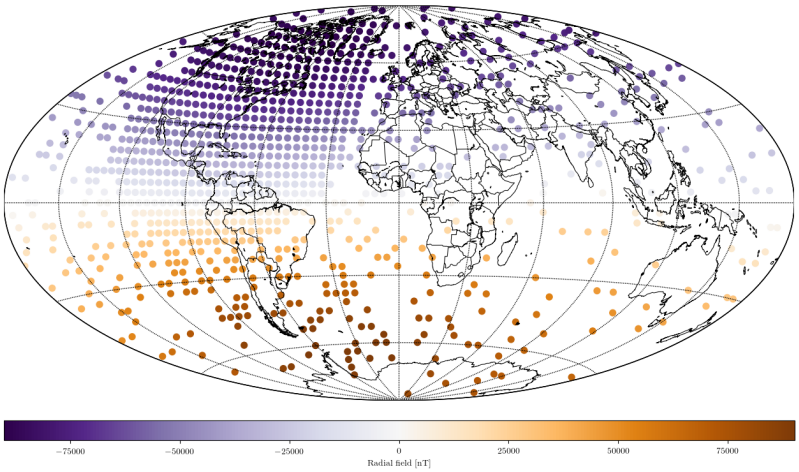


Figure 4.3.10: Example of approximate global coverage neighborhood for 2,998 observations, with approximately one third sampled as conditional data.

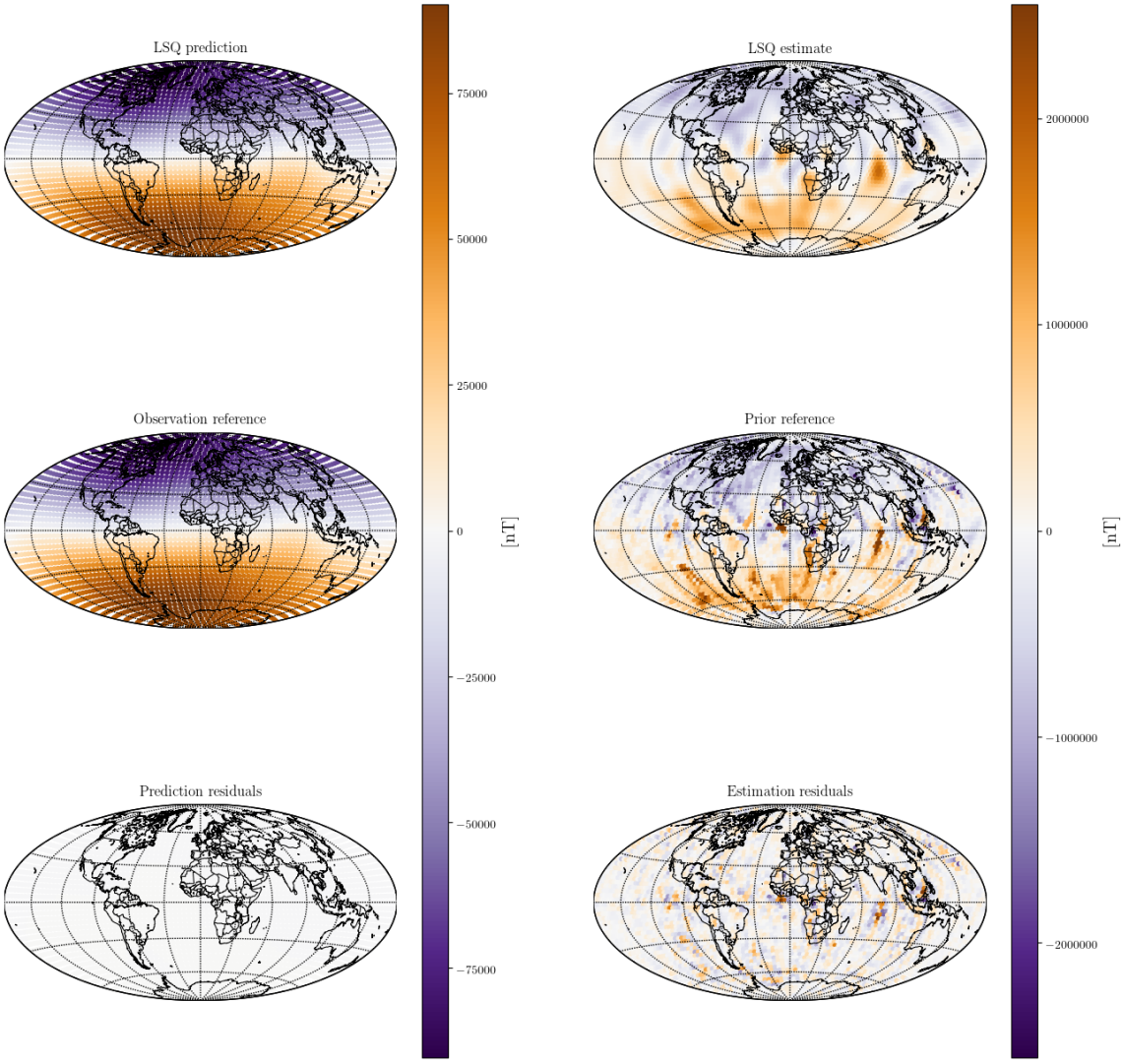


Figure 4.3.11: Realization of sequential least squares solution with approximate global coverage neighborhood, estimating the core mantle boundary field on a 5,000 location grid.

4.4 Observations + prior

In this section I present results of full spherical direct sequential simulation, where each source target estimate is conditional to both synthetic observations and the previously simulated values. In all the results an observation error estimate of $1nT$ has been added to the ordinary Kriging system, through the error covariance matrix described in section 2.3.3. I draw upon the test results from section 4.2 and 4.3 to configure the Kriging system. An overview is given in table 4.4.1. In addition, a small test has been made to ensure that neighborhood sizes are chosen such that histogram and semi-variogram reproduction is still accomplished, this is available in appendix D.3.1.

Parameter overview for posterior realizations at the CMB using synthetic observations		
4.4.1 AGC neighborhood observations + prior		
Grid sizes	CMB	Observations
	$N_S = 5000$	$N_D = 2998$
Neighborhood sizes	CMB	Observations
(A)	$N_S/50$	$N_D/3$
(B)	$N_S/50$	$N_D/3$
Semi-variogram type		
(A)		exp + Gau
(B)		double spherical
Conditional dist. type		
(A)		no dipole
(B)		dipole
Realizations		100
4.4.2 Increased core grid size		
Grid sizes	CMB	Observations
	$N_S = 15000$	$N_D = 2998$
Neighborhood sizes	CMB	Observations
	$N_S/50$	$N_D/3$
Semi-variogram type		exp + Gau
Realizations		10
Conditional dist. type		no dipole
4.4.3 All observations + prior		
Grid sizes	CMB	Observations
	$N_S = 15000$	$N_D = 2998$
Neighborhood sizes	CMB	Observations
	$N_S/50$	$N_D/1$
Semi-variogram type		exp + Gau
Realizations		5
Conditional dist. type		no dipole

Table 4.4.1: Parameters under consideration for direct sequential simulation of synthetic observations + prior information.

I use different algorithm configurations, all leading to posterior realizations of the core mantle boundary field, conditional to observations and previously simulated values. These configurations start with an AGC observation neighborhood based estimate of 2,998 synthetic observations, for 5,000 source location estimates of the CMB radial field. Here I show results from two different configurations, **(A)** and **(B)**,

wherein the difference lies in the used semi-variogram and conditional distribution type. **(A)** is based on the exponential + Gaussian semi-variogram model, with conditional distributions from the CMB field training image without dipole. **(B)** is based on the double spherical semi-variogram model, with conditional distributions from the CMB field training image with dipole. The semi-variogram types were shown in section 3.3 and conditional distribution generation in section 3.4. Following the test of systems **(A)** and **(B)**, the source location grid is increased in size to 15,000. Finally, posterior realizations are made while including all available synthetic observations. This is done for a system of 2,998 total synthetic observations. In the last two cases, these systems increase computation time through more, and in the second case, larger Kriging systems needed to be solved, resulting in fewer realizations available due to time constraints.
In the following sections I go over the results in detail, but a brief overview of the posterior observation prediction fits and residuals, are found in the table below as a comparison reference. As a final note, the neighborhood considered at the core mantle boundary is not approximate global coverage, but rather the nearest simulated values. This is a cut off at the neighborhood size value, determined as some fraction of the total available locations. This neighborhood method is the same as the one used for stochastic realizations in section 4.2. An example of such a neighborhood is seen in figure 4.4.1. Note the missing values of the cluster, these are field estimates yet to be determined.

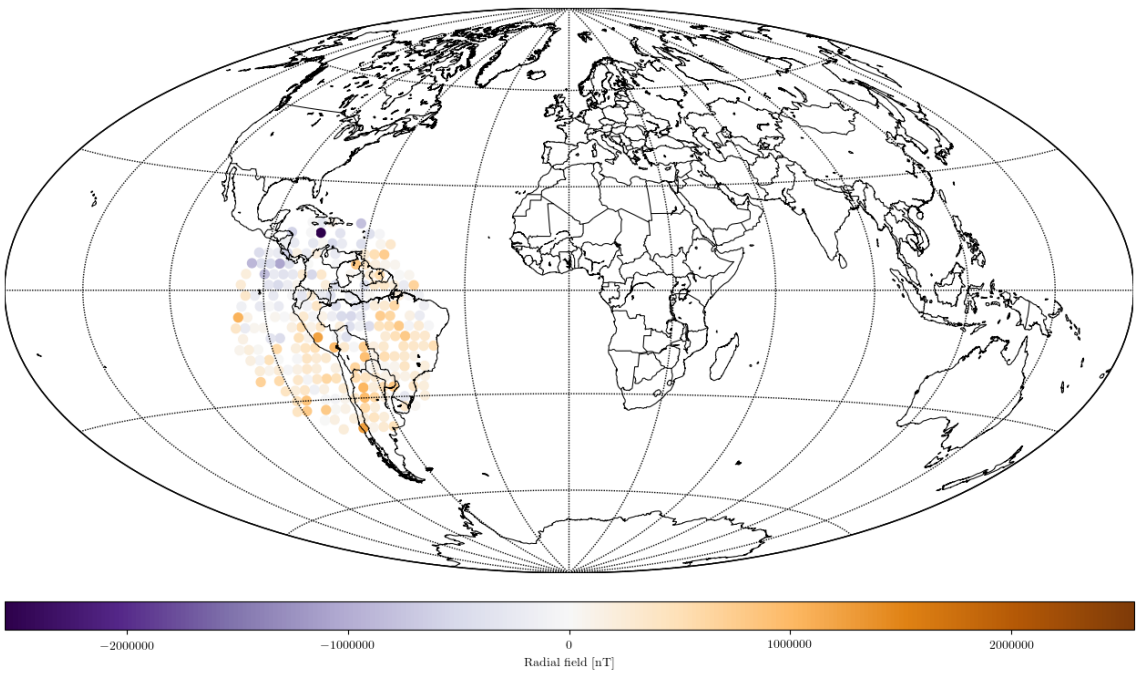


Figure 4.4.1: Source target neighborhood for the estimation of a single radial field value at the core mantle boundary during direct sequential simulation.

Result	Obs. mean $[nT]$	Obs. std.dev. $[nT]$	Res. mean $[nT]$	Res. std.dev. $[nT]$
(A)	139.8	$5.28e+04$	-139	113
(B)	146.5	$5.28e+04$	-146	113
4.4.2	-36.79	$5.28e+04$	37.8	223
4.4.3	13.12	$5.27e+04$	-12.1	129
Target	0.98	$5.28e+04$	N/A	N/A

4.4.1 AGC neighborhood observations + prior

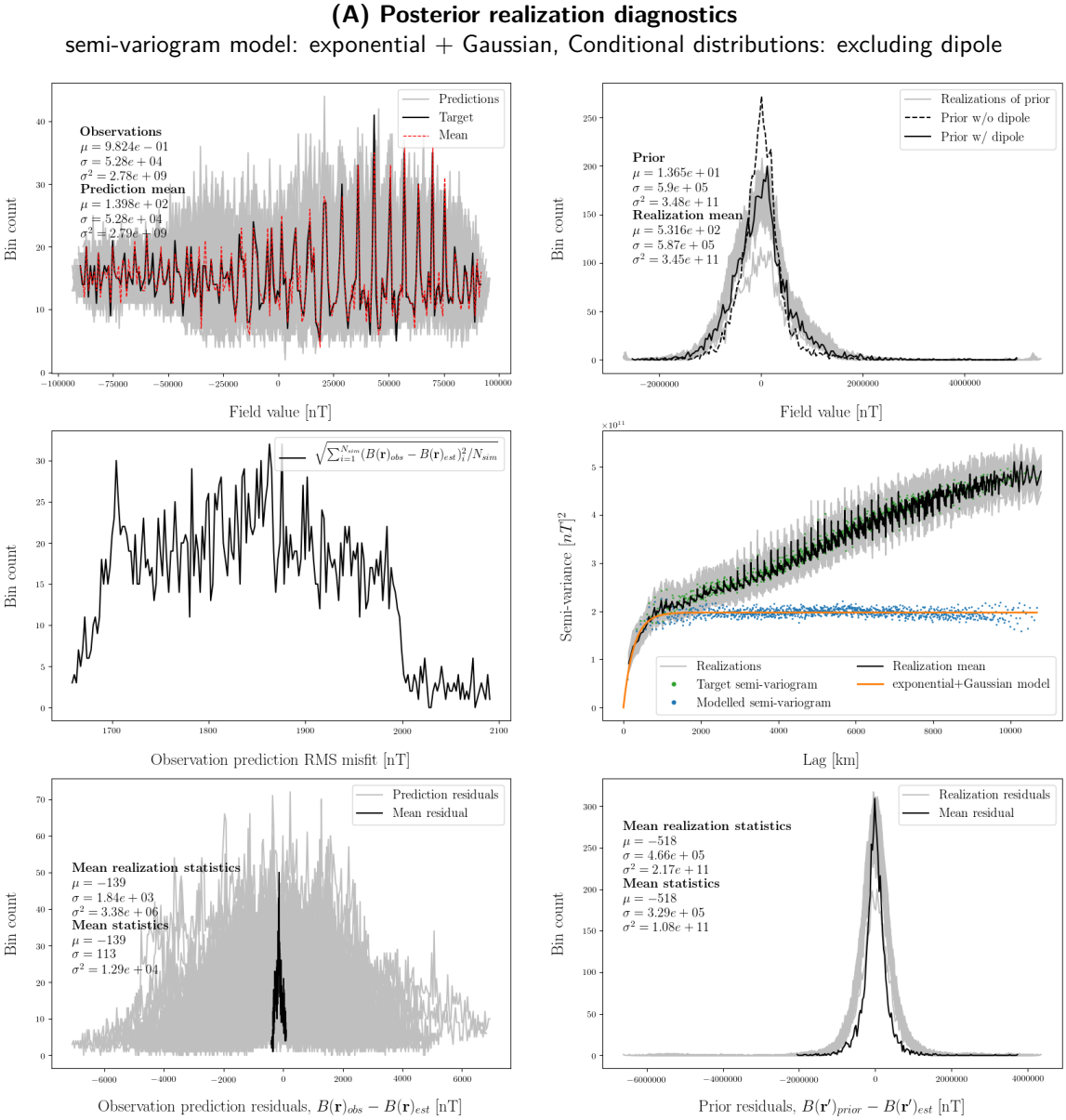


Figure 4.4.2: Diagnostics for **(A)** of 100 realizations using 2,998 synthetic observations with a target source grid of 5,000 locations. The simulation is conditional to previously simulated values and synthetic observations. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Depicted are an observation reproduction histogram (upper left), a CMB estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

Figure 4.4.2 shows the diagnostics for **(A)**. These are 100 realizations using an AGC neighborhood for one third of the total synthetic observations and a source neighborhood for 100 previously simulated values. Histogram and semi-variogram shapes are followed and the mean observation prediction is approaching a fit to the target. The observation residuals show the mean prediction at a mean of -139 nT with standard deviation 113 nT. This is larger than expected, possibly indicating 100 realizations may not be enough to characterize the solution fully. The computation time for these 100 realizations are ~ 80 hours.
In order to find a faster converging solution, the **(B)** system was set up, and the results are shown in figure 4.4.3. Here a double spherical semi-variogram model is used, as an attempt to include the dipole training image as prior for the local conditional distributions. This circumvents the issues present in reproducing

the prior histogram for stochastic simulations, as shown in appendix D.1.1. However, looking at the simulation in figure 4.4.3, it appears very similar to the previous system. One difference may be found with the small scale semi-variogram reproduction. This is not being reproduced as well in the double spherical case **(B)**. As a point of note, the residual mean and variance of both simulations are nearly the same. This may suggest that the presently used Kriging system won't converge on the LSQ solution or that the rate of convergence is the same.

(B) Posterior realization diagnostics

semi-variogram model: double spherical, Conditional distributions: including dipole

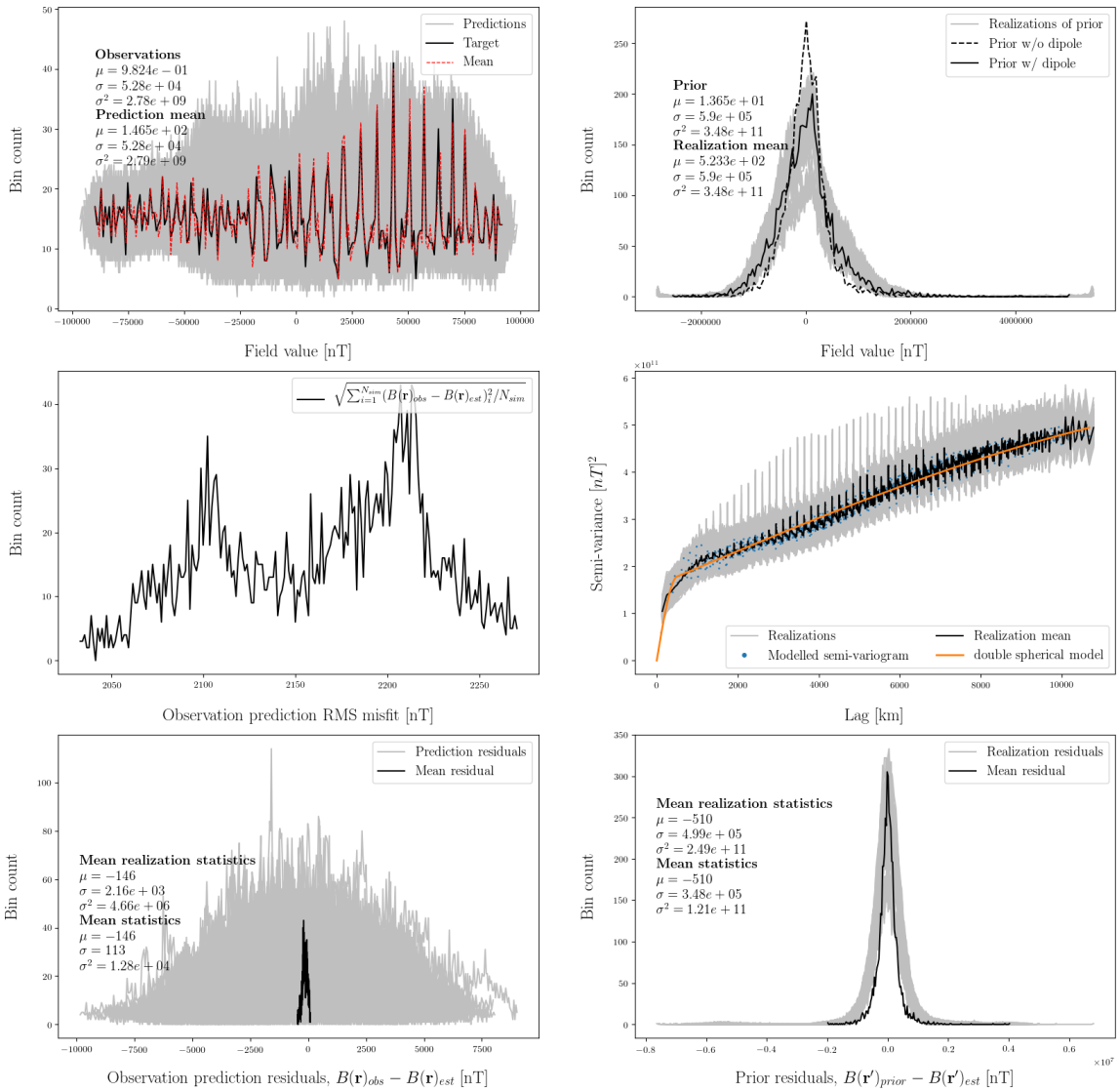


Figure 4.4.3: Diagnostics for **(B)** of 100 realizations using 2,998 synthetic observations with a target source grid of 5,000 locations. The simulation is conditional to previously simulated values and synthetic observations. A double spherical semi-variogram model has been used and the local conditional distributions are based on the prior training image with included dipole. Depicted are an observation reproduction histogram (upper left), a CMB estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

Figure 4.4.4 show examples from the posterior realizations of system **(A)**. From visual inspection of the results, they do appear to reproduce the structures as expected from the LSQ solution previously shown. This is most visibly seen from the triangular positive radial field structure located in the superimposed Indian Ocean. Figure 4.4.5 show samples of the ordinary Kriging weights, and resulting magnitude of the

estimating linear system, according to conditional data index. The observation neighborhood is visible, with ordered beginning followed by random geometrically ordered sampling, and lastly small weights for the target source neighborhood. Note how the conditional observations drop off in influence according to distance from the target, as well as the minor conditioning supplied by the previously simulated values. The conditioning from previously simulated values amount to a few nanotesla.

(A) Example realizations

semi-variogram model: exponential + Gaussian, Conditional distributions: excluding dipole

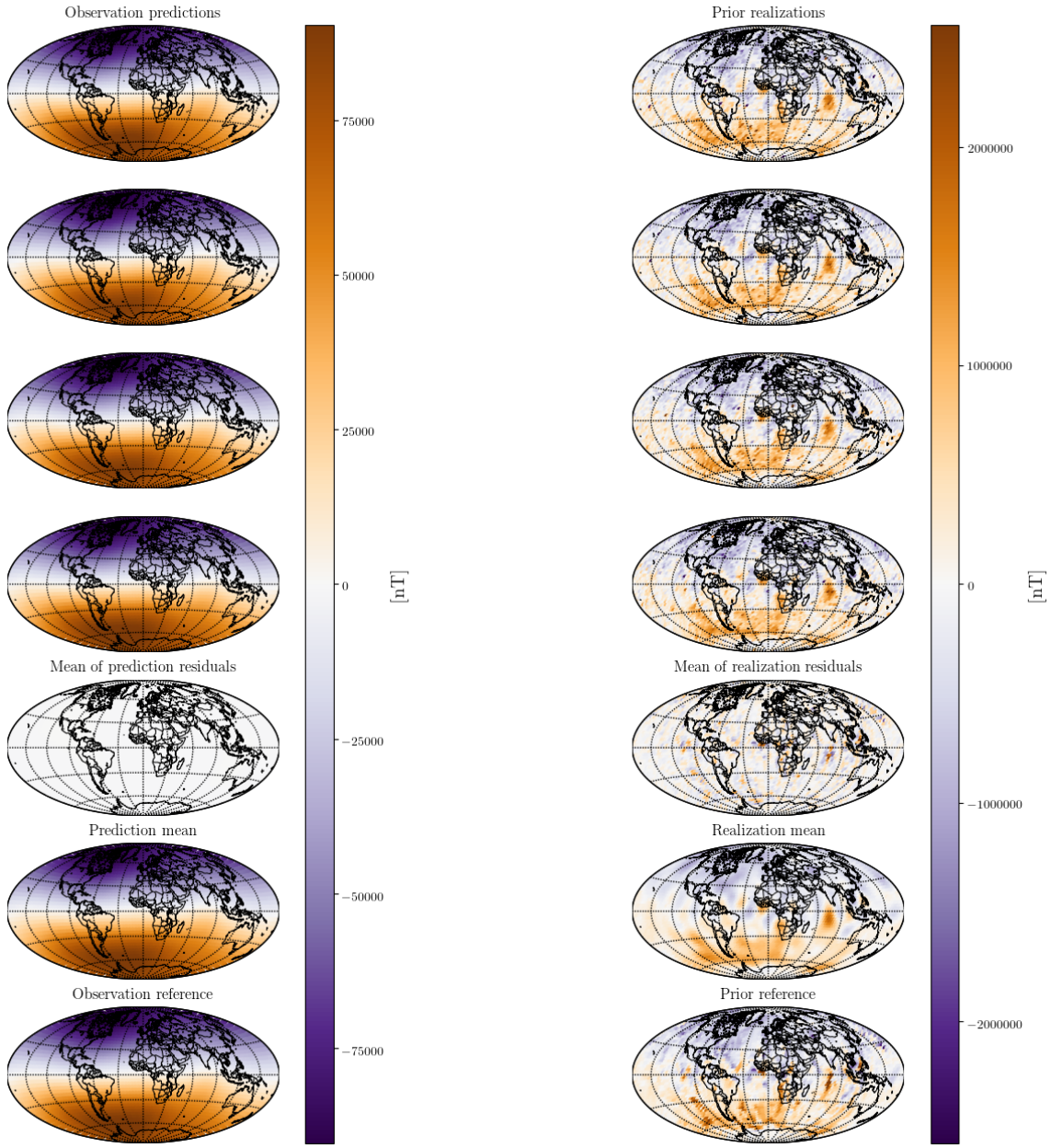


Figure 4.4.4: Example realizations for **(A)**. Included are a plot of the residuals at the CMB and for the observation reproduction at satellite altitude, as well as the mean of realizations, conditional observations, and the CMB training image.

(A) Example ordinary Kriging weights

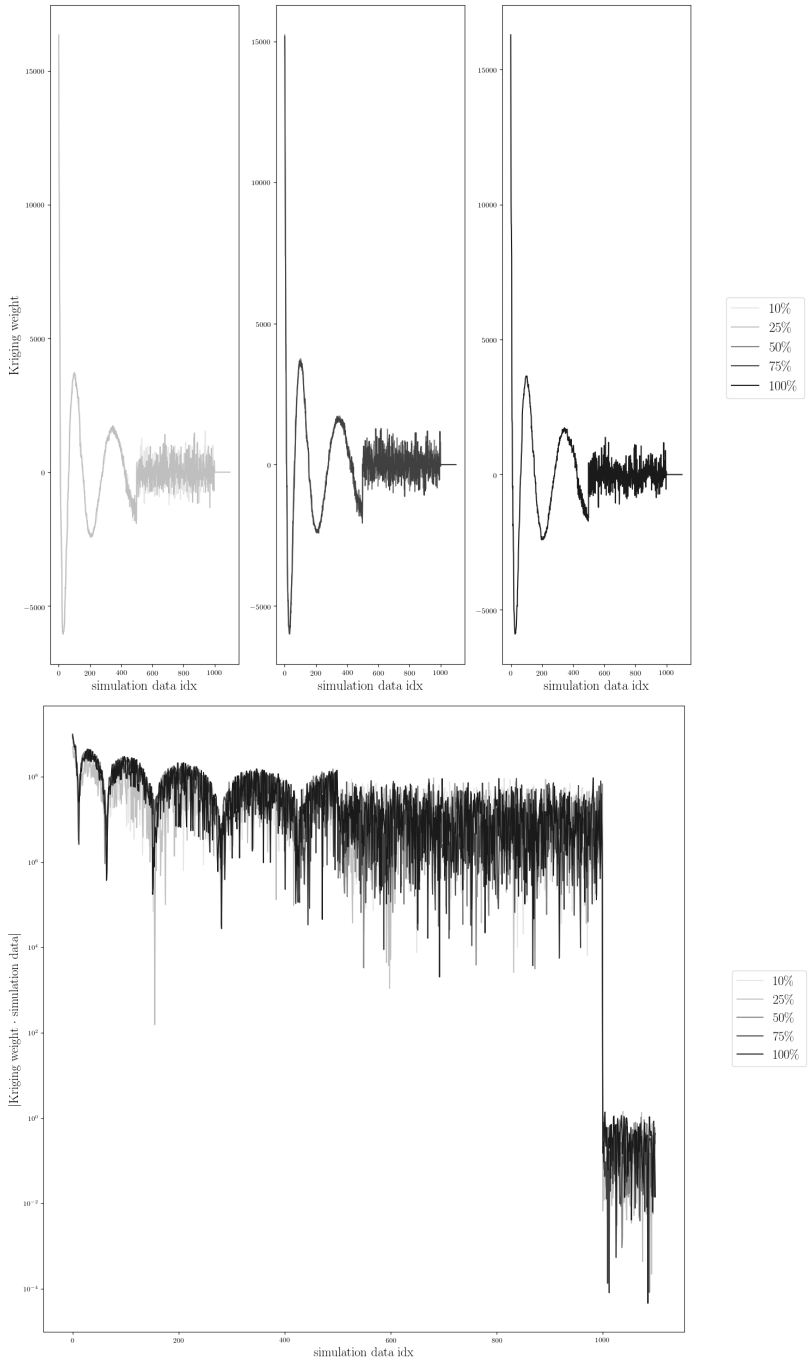


Figure 4.4.5: Sampled ordinary Kriging weights and resulting magnitude of the estimating linear system according to conditional data index, for **(A)**. The samples are picked at the shown simulation progression fraction [%] for one posterior realization.

4.4.2 Increased core grid size

The sorce grid size is now increased to 15,000 locations. Figure 4.4.6 shows 10 realizations using an AGC neighborhood for one third of the total synthetic observations, and a cut-off neighborhood for 300 previously simulated values. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Histogram and semi-variogram shapes are followed and the mean observation prediction is approaching a fit to the target. The observation residuals show the mean prediction at a mean of 37.8 nT with standard deviation 223 nT.

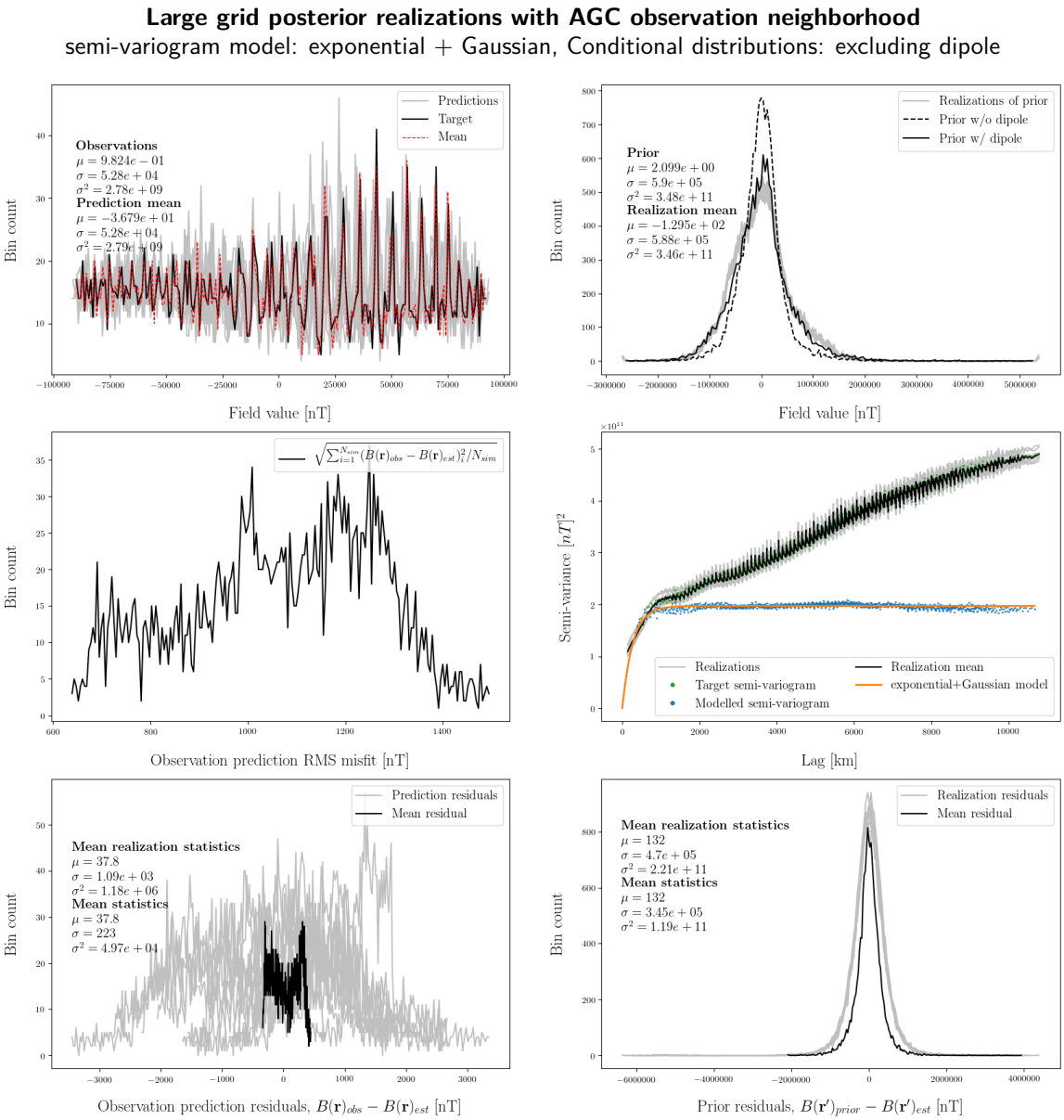


Figure 4.4.6: Diagnostics for DSSIM of 10 realizations using 2,998 synthetic observations with a target source grid of 15,000 locations. The simulation is conditional to previously simulated values and synthetic observations. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Depicted are an observation reproduction histogram (upper left), a CMB field estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

This larger grid approached a reasonable level of fit faster, reaching similar residual statistics at one tenth the realizations compared to **(A)** and **(B)**, while improving the observation prediction RMS fit. However, these ten realizations in a 15,000 source location grid, require similar computation time as 100 realizations using the 5,000 source location grid. In addition, note the small scale semi-variogram reproduction. Compared to the smaller grid, this is not being reproduced very well. In the previous section I indicated this being caused by a different semi-variogram model, but in this case, the simulation has been carried out with prior models previously displaying good fit at small scale variability. This may indicate a need for larger neighborhoods of previously simulated values, to allow more conditioning influence, when using larger grids, even when the neighborhood area is scaled.

4.4.3 All observations + prior

Here I show a simulation of five realizations conditional to all available observations. The simulation was made with a setup of 2,998 synthetic observations and a target source grid of 15,000 locations. The very few realizations make the statistics unsure, however, they do appear to fit the histogram and semi-variogram, as well as approach mean reproduction of the observations. Observation prediction residuals improve over previous methods, but computation time per realizations has increased significantly. Computationally, 100 realizations using AGC on a 5,000 CMB field grid also takes an equivalent amount of time as the 5 realizations shown here. A similar simulation of 15,000 target source locations and 998 synthetic observations can be found in appendix D, figure D.3.2. The results of that simulation are less well estimating than this one, showing that an increase in observation density while using global coverage increases precision.

Large grid posterior realizations using all observations

semi-variogram model: exponential + Gaussian, Conditional distributions: excluding dipole

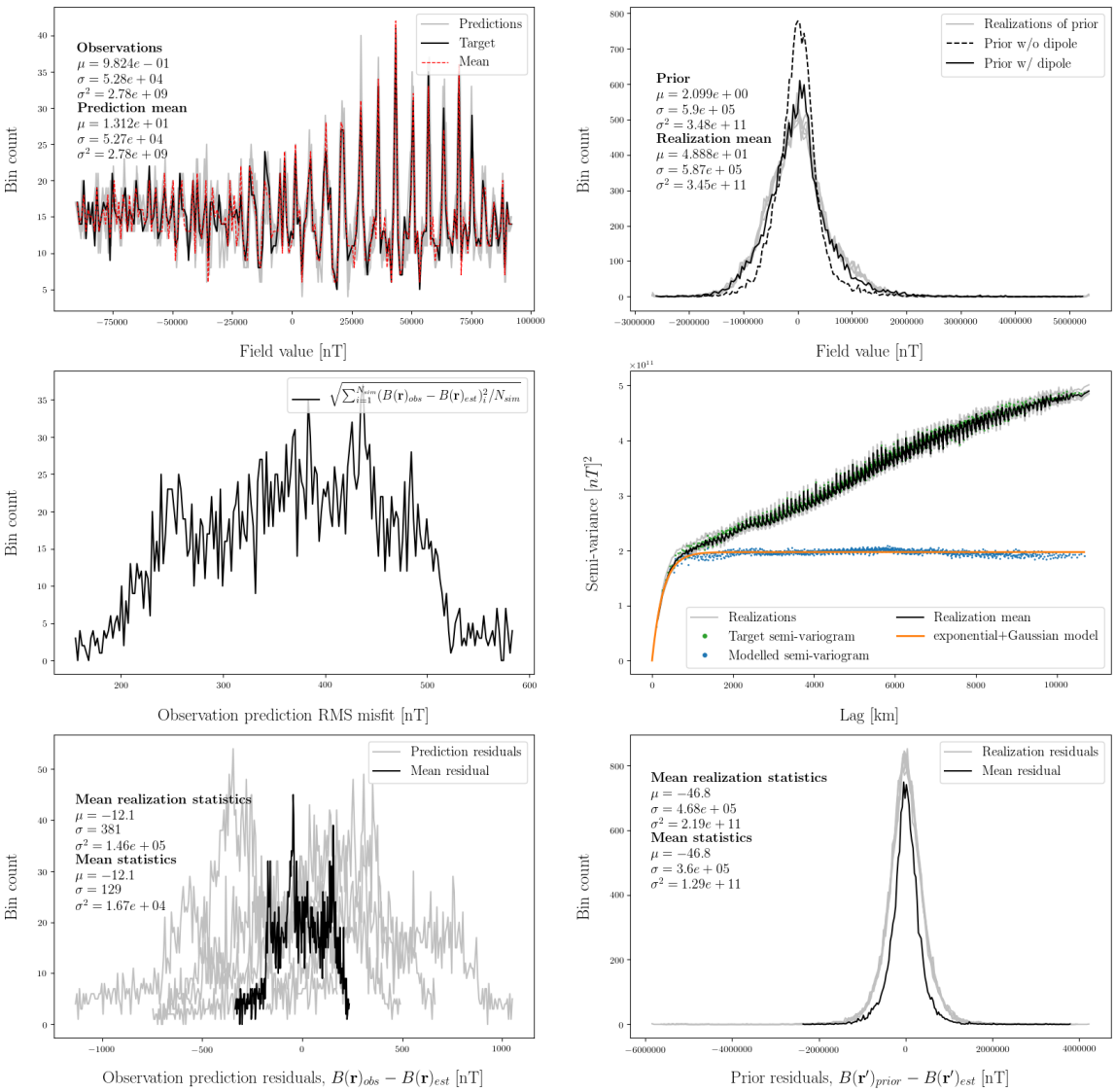


Figure 4.4.7: Diagnostics for DSSIM of 5 realizations using 2,998 synthetic observations with a target source grid of 15,000 locations. All 2,998 synthetic observations have been used as conditional data. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Depicted are an observation reproduction histogram (upper left), a CMB field estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

4.5 SDSSIM with Swarm satellite observations

I now present the results of applying my spherical direct sequential simulation tool to real satellite observations from Swarm alpha, one out of the three available data sets as described in section 3.6. To begin, a sequential least squares estimate is shown, using all Swarm alpha observations. Following that, I present two posterior realizations **(C)** and **(D)**. **(C)** uses a nearest simulated value neighborhood for estimates at the core mantle boundary, and an approximate global coverage neighborhood for observations. **(D)** uses all available satellite observations, and is not conditional to simulated values at the core mantle boundary. In all these results, a naive observation error estimate of $10nT$ has been added to the ordinary Kriging system, through the error covariance matrix described in section 2.3.3. An overview of all the parameters used to compute the following results, can be seen in table 4.5.1, as well as an overview of the resulting observation prediction fit and residuals in table 4.5.2.

Overview for estimates of the CMB field using Swarm alpha observations		
4.5.1 Sequential LSQ		
CMB grid size		$N_S = 5000$
No. of observations		$N_D = 2773$
Semi-variogram type		exp + Gau
Observation neighborhood		all
Conditional dist. type		no dipole
Realizations		1
4.5.2 Posterior realizations		
CMB grid size		$N_S = 5000$
No. of observations		$N_D = 2773$
Neighborhood sizes	CMB	Observations
(C)	$N_S/25$	$N_D/3$
(D)	0	all
Semi-variogram type		
(C)		exp + Gau
(D)		exp + Gau
Conditional dist. type		
(C)		no dipole
(D)		no dipole
Realizations		100

Table 4.5.1: Parameters used to compute results for direct sequential simulation of Swarm alpha satellite observations.

Result	**Obs. mean** $[nT]$	**Obs. std.dev.** $[nT]$	**Res. mean** $[nT]$	**Res. std.dev.** $[nT]$
Seq. LSQ	$1.096e+04$	$3.04e+04$	$-3.22e-05$	4.93
(C)	$1.101e+04$	$3.06e+04$	-56.5	213
(D)	$1.095e+04$	$3.04e+04$	5.68	181
Target	$1.096e+04$	$3.04e+04$	N/A	N/A

Table 4.5.2: Observation prediction fit and residual results for the estimates determined from Swarm alpha satellite observations.

4.5.1 Sequential LSQ with Swarm alpha observations

Figure 4.5.1 and 4.5.2 show the results of sequential least squares estimation on the observations from Swarm alpha. The estimation is carried out by use of SDSSIM as shown in flowchart 3.7.3. As can be seen on the plot as well as table 4.5.2, the data fit is well within the error estimate of $10nT$. This is a good indication that the implementation works well for noisy non-synthetic observations.

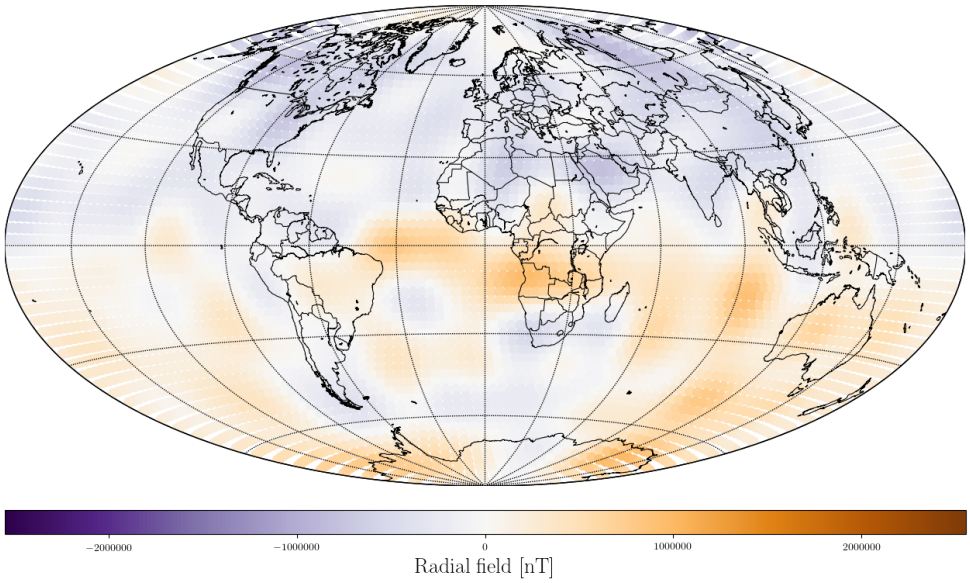


Figure 4.5.1: Radial field plot of estimates from sequential least squares at the core mantle boundary using Swarm alpha satellite observations. The estimated values are determined on an approximate equal area grid of 5,000 locations, using 2,773 satellite observations. $10nT$ noise has been added to the system prior to computing the solution.

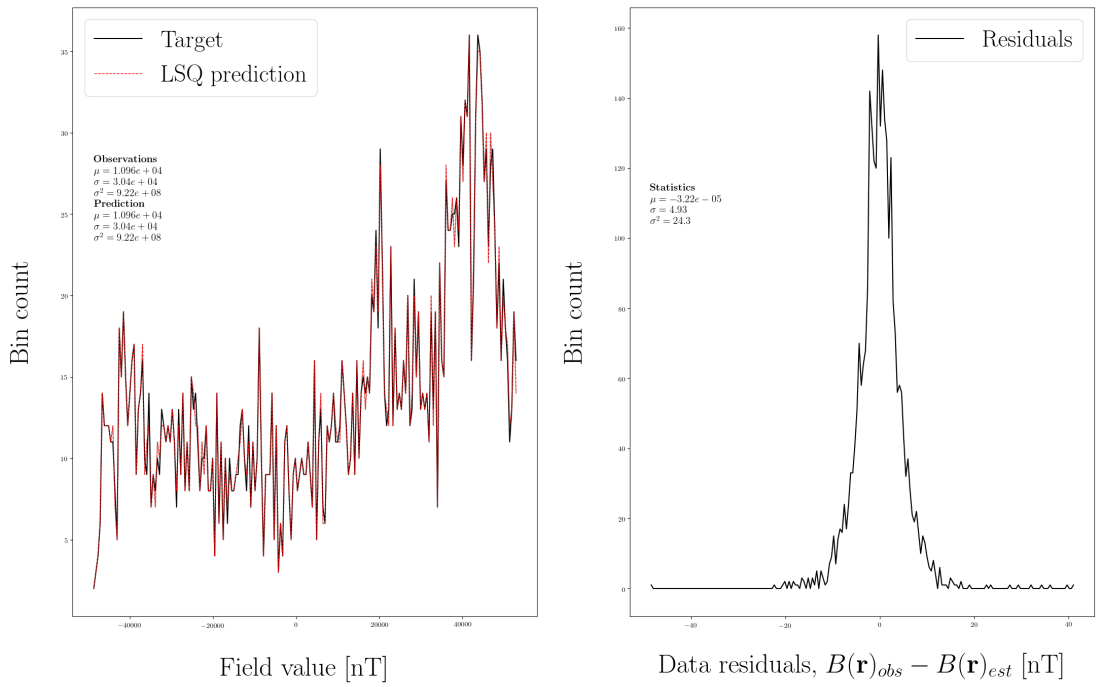


Figure 4.5.2: Prediction fit for sequential least squares estimation of the radial field at the core mantle boundary using Swarm satellite observations.

4.5.2 Posterior realizations from Swarm alpha observations

The final results of this thesis are posterior realizations using Swarm alpha observations. In the following I present two such cases, **(C)** and **(D)**. **(C)** computes the posterior realizations identically to the method used for the synthetic case, with AGC and nearest simulated value neighborhoods, while **(D)** uses all observations and no previously simulated values. As such, **(D)** may not strictly be a realization of the posterior. However, the result in the previous section showing that previously simulated values contribute very little to each estimate, may lead to the use of more observation data being a better approximation than estimation with approximate global neighborhoods.

(C) Example realizations

semi-variogram model: exponential + Gaussian, Conditional distributions: excluding dipole

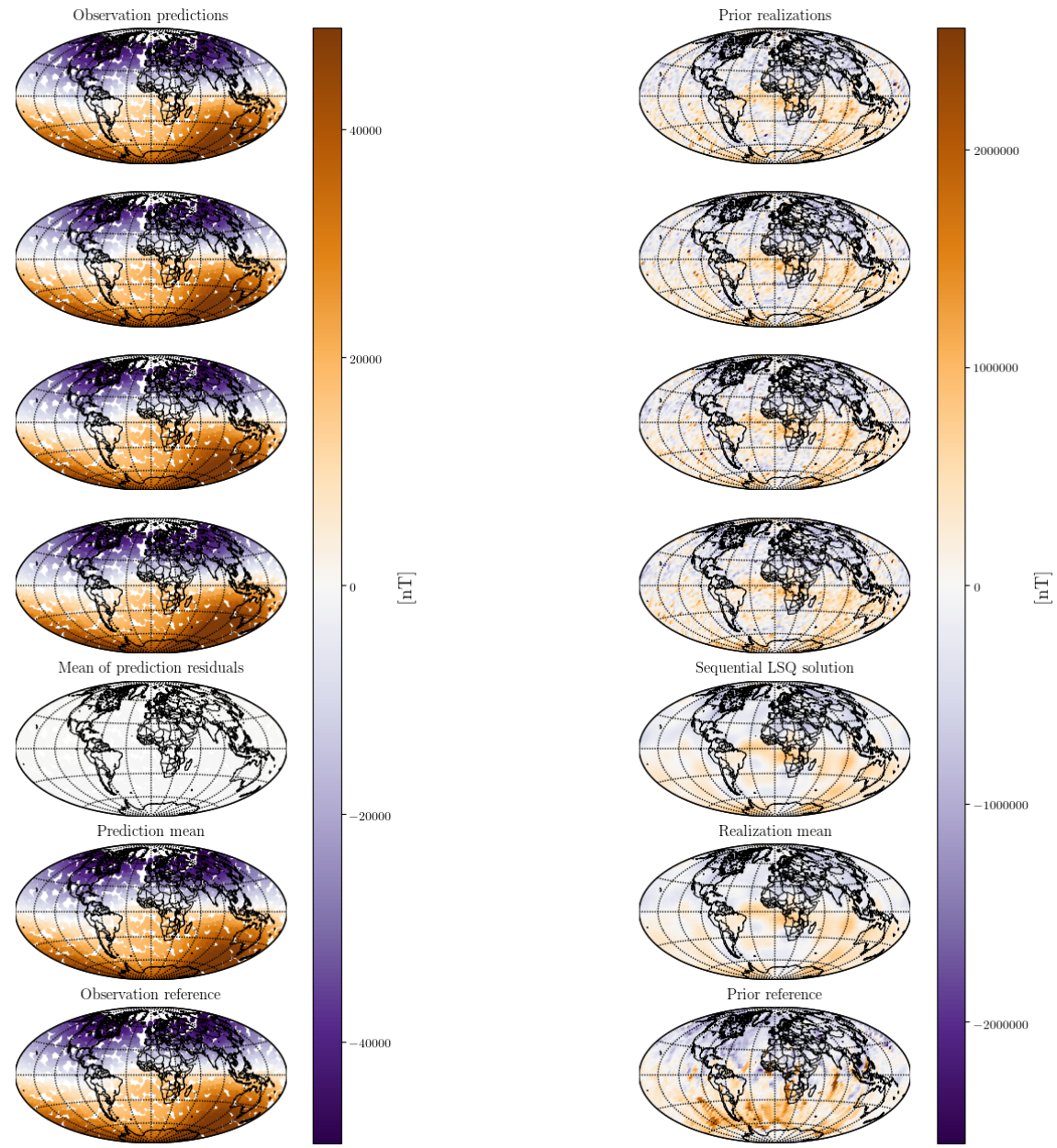


Figure 4.5.3: Example realizations for **(C)**. Included are a plot of the residuals at the CMB and for the observation reproduction at satellite altitude, as well as the mean of realizations, conditional observations, and the CMB training image.

Figure 4.5.3 shows examples of realizations from **(C)**. Clearly it is possible to generate posterior realizations

of real satellite observations, and as expected, the realization structure is similar to the sequential least squares estimate. However, there is not a lot of structure from the prior CMB field training image being reproduced, and the latitudinal trend (dipole) is less pronounced.

(C) Posterior realizations conditional to previously simulated values

semi-variogram model: exponential + Gaussian, Conditional distributions: excluding dipole

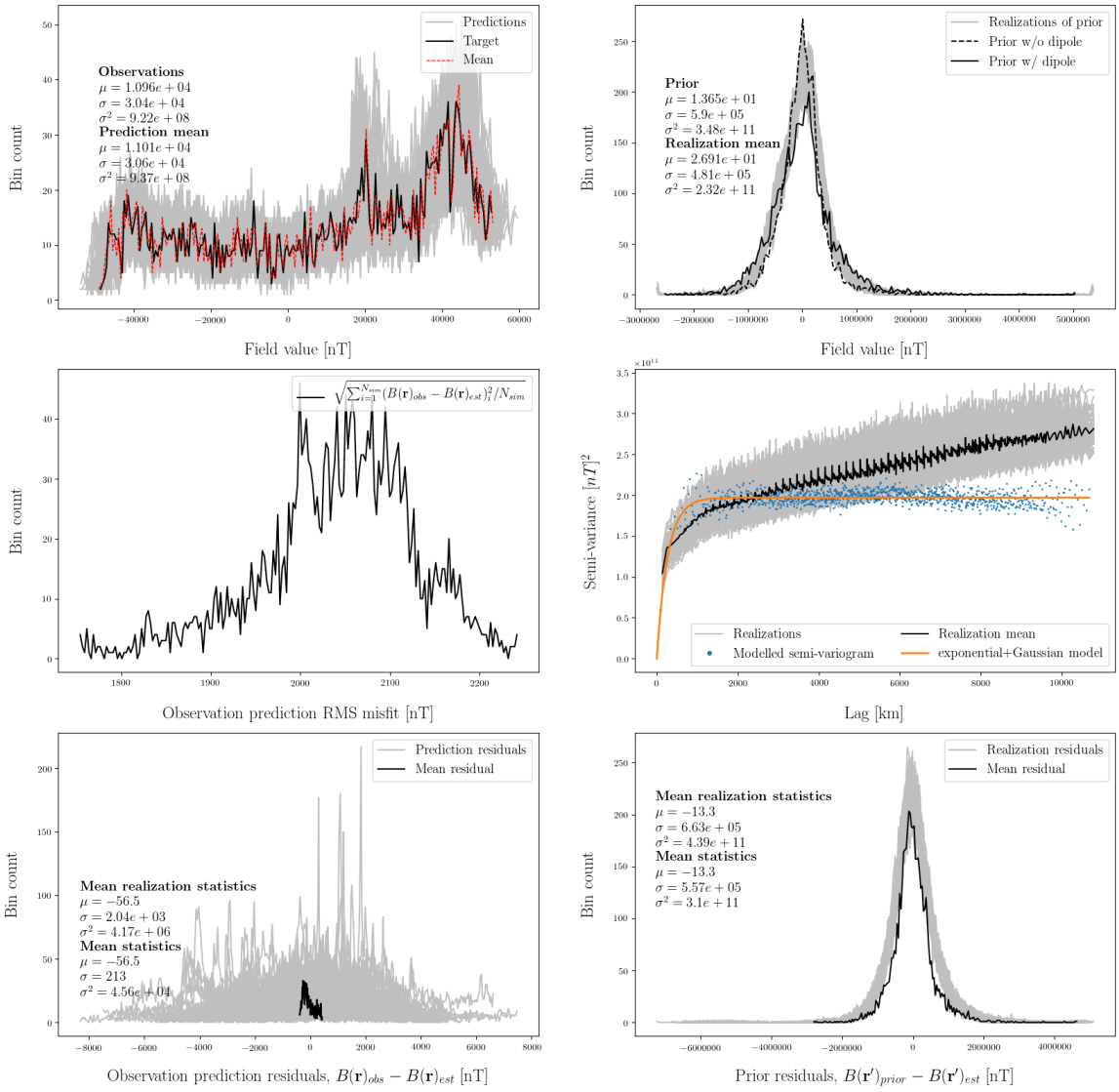


Figure 4.5.4: Diagnostics for **(C)** of 100 realizations using an approximate global coverage neighborhood on available Swarm alpha observations with a target source grid of 5,000 locations. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Depicted are an observation reproduction histogram (upper left), a CMB field estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

Figure 4.5.4 and 4.5.5 are the statistics diagnostics for **(C)** and **(D)** respectively. It is immediately apparent that these two simulations are very similar. Some differences are found in the statistics, where **(C)** is fitting slightly better to the observations. However, the CMB field training image statistics fit is roughly the same, indicating both are equally well conditioned on the prior. This is despite **(D)** not being conditioned on previously simulated values. It is possible that the prior covariance information in the Green's kernel as shown in section 2.2.2 and implemented through the semi-variogram, combined with global observation coverage, make up for any information gained from previously simulated values. In

this case, the only information gain possible, besides possible implementation changes, may be to use all available observations in conjunction with the previously simulated values, as seen for the synthetic case in section 4.4.3.

(D) Posterior realizations unconditional to previously simulated values

semi-variogram model: exponential + Gaussian, Conditional distributions: excluding dipole

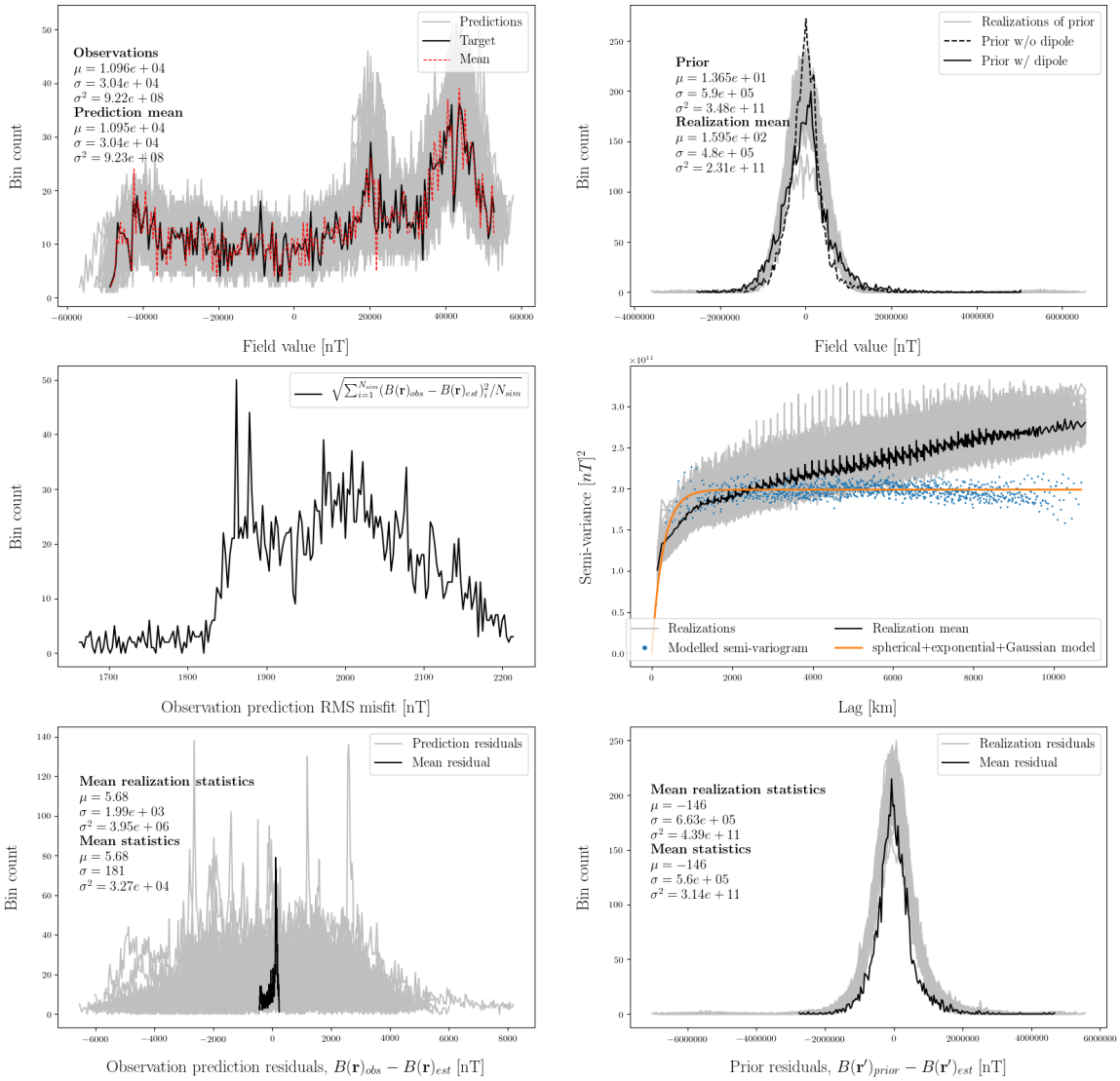


Figure 4.5.5: Diagnostics for SDSSIM of 100 realizations using all available Swarm alpha observations with a target source grid of 5,000 locations. An exponential + Gaussian semi-variogram model has been used and the local conditional distributions are based on the prior training image without dipole. Depicted are an observation reproduction histogram (upper left), a CMB field estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

This concludes the results. I now move on to a discussion where I give my thoughts on the preceding findings and possible improvements to achieve better posterior realizations.

Chapter 5

Discussion

This chapter contains a discussion of my implemented spherical direct sequential simulation tool and the results achieved through it. First a comparison is made with simple inversion of the linear systems, upon which my results are based. From this, ideas related to inversion in the implementation are explained. This is followed by a general critique of the implementation's shortcomings and limitations, and my ideas for a path to be taken in order to improve the current implementation further.

Improving and expanding the Kriging system solution

Consider the two least squares estimates of the core mantle boundary radial field in figure 5.0.1. These have been computed by solving the linear system of equations shown in equation 5.0.1, by the same SVD method as used for my Kriging system. Here $\sin\theta'_m \Delta\theta'_m \Delta\phi'_m$ has been absorbed into $\boldsymbol{G_d(r, r')}$.

$$\begin{aligned} B_r(\boldsymbol{r}) &\approx \sum_{m=1}^{N_S} G_r(\boldsymbol{r}, \boldsymbol{r'_m}) B_r(\boldsymbol{r'_m}) \sin\theta'_m \Delta\theta'_m \Delta\phi'_m \\ \boldsymbol{B_r(r)} &\approx \boldsymbol{G_d(r, r') B_r(r')} \end{aligned} \tag{5.0.1}$$

From the estimate, this looks to be readily solved by least squares in the case of synthetic observations, but in the noisy case of real satellite observations, the system is ill-conditioned leading to instability. In the case of my implemented ordinary Kriging system, the covariance expression shown in equation 5.0.2, is the essential kernel being inverted.

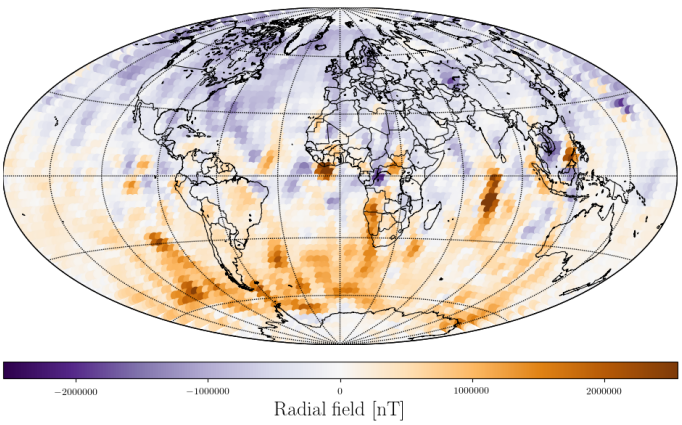
$$C_{i,j} = \sum_{m=1}^{N_S} \sum_{n=1}^{N_S} G_k(\boldsymbol{r_i}, \boldsymbol{r'_m}) G_k(\boldsymbol{r_j}, \boldsymbol{r'_n}) \Delta_m \Delta_n C(\boldsymbol{r'_m}, \boldsymbol{r'_n}) \tag{5.0.2}$$

Note the parallels of the double Green's function kernel product to normal equations, as that is essentially what it is. Inversion of this Green's kernel product by the same SVD system solver is unstable, even for the synthetic case. However, noise estimates are added to get around this issue, leading to the well fitting sequential LSQ results shown in section 4.3.2 and 4.5.1. This is a rather simple solution that I haven't fully explored, and I find it plausible that related methods such as regularization may improve upon the current noise + SVD based solution. However, an immediate issue here may be computational speeds, as the Python SVD solver is very efficient by acceleration through the Intel Math Kernel Library (MKL). Finally, the motivation for this thesis has been to work toward separation of core and lithosphere field sources, and the current implementation seems open to expansion in aid of this endeavour. For instance, the ordinary Kriging system could be expanded through observations from sources other than satellites, e.g. ground observations, as seen in equation 5.0.3. Where C_G is a Green's function kernel covariance expression, as currently in use for the satellite observation implementation. Of course what is really desired, is a system which estimates both the core and lithosphere simultaneously, conditional to each other. A naive expression of such a system may be in an expansion of the Kriging linear weighting scheme

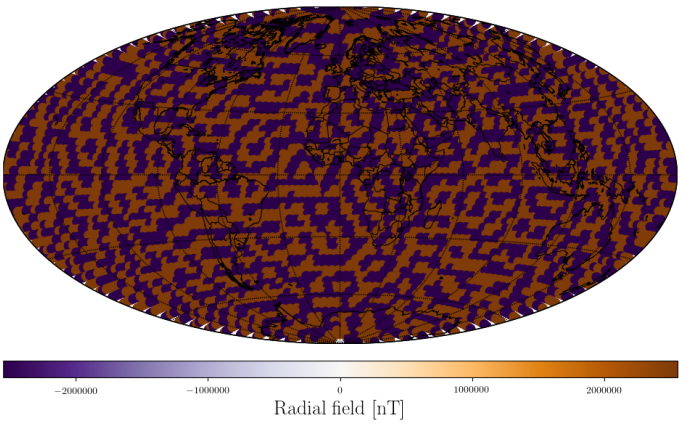
as illustrated in equation 5.0.4. Whether this is feasible as a Kriging system requires more thought, as I have been unable to find applications of such a system in literature.

$$\boldsymbol{K\lambda} = \boldsymbol{k} \rightarrow \begin{bmatrix} \boldsymbol{C_{sat}} + \boldsymbol{C_{E1}} & \boldsymbol{C_{crg}} & \boldsymbol{C_{crs}} & \mathbf{1} \\ \boldsymbol{C_{crg}^T} & \boldsymbol{C_G} + \boldsymbol{C_{E2}} & \boldsymbol{C_{crgs}} & \mathbf{1} \\ \boldsymbol{C_{crs}^T} & \boldsymbol{C_{crgs}^T} & \boldsymbol{C_S} & \mathbf{1} \\ \mathbf{1}^T & \mathbf{1}^T & \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega_{sat}} \\ \boldsymbol{\omega_G} \\ \boldsymbol{\omega_S} \\ \boldsymbol{\Lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c_{sat}} \\ \boldsymbol{c_G} \\ \boldsymbol{c_S} \\ 1 \end{bmatrix} \tag{5.0.3}$$

$$\begin{bmatrix} \hat{B}_r(\boldsymbol{r'_{t1}}) \\ \hat{B}_r(\boldsymbol{r'_{t2}}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega_1^T B_r(r_1)} \\ \boldsymbol{\omega_2^T B_r(r_2)} \end{bmatrix} \tag{5.0.4}$$



(a)



(b)

Figure 5.0.1: Radial field plots of estimates from singular value decomposition least squares at the core mantle boundary. **(a)** is inversion of synthetic observations and **(b)** is inversion of Swarm alpha observations.

Limits and optimizations

In my eyes, this SDSSIM implementation is a struggle between needing to use as many observations as possible, as dictated by the Green's function implementation, and wanting to use as little as possible to reach good computational speed. This dichotomy is an issue, since no good balance has been found for small observation neighborhoods. In addition, small source grids are also desired for computational efficiency, limiting the effectiveness of the integration approximation.

It is possible that the solution is to brute force realizations using neighborhoods and grids as large as possible, but a more wanted solution would be one that always produces posterior realizations close to the observation fit. In previous sequential simulation, one method has been to use likelihood functions in a Monte Carlo framework, however, the current implementation does not produce model realizations very fast, rendering this option unlikely. Other methods will probably have to be used if this is to be achieved.
With regards to the results, no presented posterior realizations in this thesis, have a mean solution converged on the expected LSQ result. I present in table 5.0.1 an outline of the simulation structures I think may be useful in reaching such a result, given more time for research and implementation.
First is a continuation of what I consider the best result presented here. The Green's function solution appear to only improve with more observations, and in the available result of a simulation using all observations in section 4.4.3, only 5 realizations have been generated. An easy and immediate step is to allow such a solution to run for a longer time, e.g. 10 realizations, to see if the mean converges significantly. If it does, an even longer simulation could be run.
Another solution readily present in literature, is updating the implementation to modelling based on anisotropic semi-variograms. This would allow a better coding of the training image information into the system. Considering the CMB field training image in figure 3.2.1, I think this might partly alleviate the issue of previously simulated values having a very small conditional effect, as there are clearly large latitudinal structures present. Currently these structures are most likely underestimated, or not present in a proper geometric fashion, in the isotropic semi-variogram models.
Another shorter term improvement might be found through investigating the stochastic realizations of the prior. These realizations are an image of how the previously simulated values work, when no conditional observations are present. Currently, these realizations do tend to the prior statistics for the mean of realizations, but single realizations fluctuate less than expected. I.e. each realization appears to be realized in a way that closely follows the shape of the semi-variogram model, only at different scales. If there is an issue here, I'm not certain where it is introduced. Expert opinions may be needed to clarify this possible issue.
Finally, implementation of different system solutions are entirely possible, either through the ideas expressed in this discussion, or some other method of implementation to be chosen. Such changes to the implementation may naturally lead to better individual realization fit and/or mean convergence of all the realizations, which will hopefully lead to a new source separation technique eventually.

Roadmap for producing posterior realizations that fit to observations

1. All obs. + prior		**3. Prior**	
CMB grid size	$N_S = 15000$	CMB grid size	$N_S = 15000$
Semi-variogram type	exp + Gau	Semi-variogram type	exp + Gau
Realizations	10-100	Realizations	100
Obs. neighborhood size	all	Obs. neighborhood size	0
Conditional dist. type	no dipole	Conditional dist. type	no dipole
Estimated timeline	1 month	Estimated timeline	1 month
2. Anisotropic SV		**4. Alternative Kriging**	
CMB grid size	$N_S = 15000$	CMB grid size	$N_S = 15000$
Total observations	$N_{obs} = 2773$	Total observations	$N_{obs} = 2773$
Obs. neighborhood size	$N_{obs}/3$	Obs. neighborhood size	$N_{obs}/3$
Semi-variogram type	NEW	Semi-variogram type	exp + Gau
Realizations	100	Realizations	100
Conditional dist. type	unknown	Conditional dist. type	no dipole
Estimated timeline	1 month	Estimated timeline	unknown

Table 5.0.1: Possible simulations to be conducted in the future, with the aim of fitting to observations.

Chapter 6

Conclusion

This thesis has documented a practical implementation of probabilistic inversion of satellite magnetic data. The forward scheme being inverted is a geomagnetic vector field description, using Green's functions for Laplace's equation in spherical geometry, with Neumann boundary conditions. The inversion itself is accomplished through ordinary Kriging based spherical direct sequential simulation with histogram reproduction. This is carried out as an approximate integration solution on an approximate equal area grid. With semi-variogram analysis, and generation of local conditional distributions through normal score transformation, statistical prior information from training images have been implemented as part of the solution. The prior information used consists of training images of the core mantle boundary and lithosphere field. These training images originate from core dynamo simulations for the core mantle boundary, and models of remanent magnetization of the oceans in combination with full Earth models of induced magnetization for the lithosphere. The implementation has resulted in a geostatistical Python tool, currently called Spherical Direct Sequential Simulation (SDSSIM), which is planned to be made publicly available. Using this tool, realizations of the radial geomagnetic field have been generated from the prior, with synthetic observations from core dynamo simulation, and with observations from the Swarm satellite constellation, ranging from April-June 2018.

Extensive results have been documented. Stochastic behaviour in realizations of different sizes of simulation neighborhoods, are tested for the prior core mantle boundary and lithosphere field. These tests show approximate reproduction of the statistical prior information for the mean of realizations, with more realizations needed to reach convergence. For the core mantle boundary field, synthetic and Swarm satellite observations are shown to be reproduced in a smooth least squares sense, using a sequential least squares method. This is a previously unexplored method of inversion in geomagnetic field modelling. It is demonstrated that computation of posterior realizations of the core mantle boundary field is possible. The realizations are generated using global observation coverage and an approximate global observation coverage method. The results indicate that the use of global observations generate better posterior realizations, showing approximate global observation coverage, as being a poor approximation in the implementation of the geomagnetic vector field description using Green's functions. All posterior realizations display a mean converging toward fitting the observations, but a fit is not reached, and will require more realizations. As such, longer simulations should be carried out to display proper convergence on the target statistics. Having said that, the produced results look promising.

Some issues are found in the prior implementation, visible through very little conditional effect of previously simulated values in the simulations. This issue may be alleviated by further development of anisotropic semi-variogram modelling or alternatives to the current Kriging system solution. Finally, the systems developed here offer future possibilities for including prior information from more than one source, and possible expansion of the estimation to two simultaneous estimation locations. Such implementation may open the door to new source separation techniques.

Bibliography

Gastine T. Fournier A. Aubert, J. Spherical convective dynamoes in the rapidly rotating asymptotic regime. *Journal of Fluid Mechanics*, 2017.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

Clayton V. Deutsch and André G. Journel. *GSLIB: Geostatistical Software Library and User's Guide*. Oxford University Press, 1998.

Chris Finlay, Nils Olsen, Stavros Kotsiaros, Nicolas Gillet, and Lars Tøffner-Clausen. Recent geomagnetic secular variation from Swarm and ground observatories as estimated in the chaos-6 geomagnetic field model. *Earth Planets and Space*, 68(1):112, 2016. ISSN 18805981, 13438832. doi: 10.1186/s40623-016-0486-1.

Magnus Danel Hammer. *Local Estimation of the Earth's Core Magnetic Field*. PhD thesis, Technical University of Denmark (DTU), 2018.

Cordua K.S. Looms M.C. Hansen, T.M. and K. Mosegaard. Sippi: A matlab toolbox for sampling the solution to inverse problems with complex prior information part 1 — methodology. *Elsevier, Computers & Geosciences*, 52(470–480), 2013a.

Cordua K.S. Looms M.C. Hansen, T.M. and K. Mosegaard. Sippi: A matlab toolbox for sampling the solution to inverse problems with complex prior information part 2 — application to crosshole gpr tomography. *Elsevier, Computers & Geosciences*, 52(470–480), 2013b.

Journel A.G. Tarantola A. Hansen, T.M. and K. Mosegaard. Linear inverse gaussian theory and geostatistics. *Geophysics*, 71(6), 2006.

T.M. Hansen and K. Mosegaard. Visim: Sequential simulation for linear inverse problems. *Computers and Geosciences*, 34(53-76), 2008.

G. Hulot, T. J. Sabaka, N. Olsen, and A. Fournier. The present and future geomagnetic field. *Treatise on Geophysics: Second Edition*, 5:33–78, 2015. doi: 10.1016/B978-0-444-53802-4.00096-8.

A.G. Journel. Modelling uncertainty: some conceptual thoughts. *Quant Geo G*, Volume 6(30-43), 1994.

A.G. Journel and Ch.J. Huijbregts. *Mining Geostatistics*. Academic Press, 1978.

Masaru Kono. Geomagnetism: An introduction and overview. *Treatise on Geophysics: Second Edition*, pages 1–31, 12 2015. doi: 10.1016/B978-0-444-53802-4.00095-6.

P. Leopardi. A partition of the unit sphere into regions of equal area and small diameter. *Applied Maths Report*, AMR05/18, 2005.

Gubbins D. Müller R.D. Singh K.H. Masterton, S.M. Forward modelling of oceanic lithospheric magnetization. *Geophysical Journal International*, 2013.

A. A. Nielsen. Kriging. http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3479/pdf/imm3479.pdf, 2004.

Nils Olsen. GMT tools for Python, 2018. Toolbox for Python, used to generate design matrices for use with (Schmidt-normalized) spherical harmonic expansion coefficients.

P. Olson. Core dynamics: An introduction and overview. *Treatise on Geophysics: Second Edition*, 8: 1–25, 2015. doi: 10.1016/B978-0-444-53802-4.00137-8.

Deutsch C.V. Tran T.T. Oz, B. and Y. Xie. Dssim-hr: A fortran 90 program for direct sequential simulation with histogram reproduction. *Computers and Geosciences*, 29(39-51), 2003.

A. D. Richmond. Ionospheric electrodynamics using magnetic apex coordinates. *Journal of Geomagnetism and Geoelectricity*, 47(2):191–212, 1995. ISSN 21855765, 00221392. doi: 10.5636/jgg.47.191.

Erwan Thébault, Chris Finlay, Ciarán D. Beggan, Patrick Alken, Julien Aubert, Olivier Barrois, Francois Bertrand, Tatiana Bondar, Axel Boness, Laura Brocco, Nils Olsen, and Lars Tøffner-Clausen. International geomagnetic reference field: the 12th generation. *Earth, Planets and Space*, 67(1):1–19, 2015. ISSN 18805981, 13438832. doi: 10.1186/s40623-015-0228-9.

Appendices

Appendix A

Initial project plan

Project plan for Master's thesis titled:

Probabilistic inversion of satellite magnetic data using geostatistical simulation in spherical geometry

Overview

Our current understanding of the geomagnetic field is that it has internal and external sources. Further, the internal sources are separated in the primary field, arising from dynamo processes in Earth's fluid core, and the smaller magnitude lithospheric field, from long-timescale cooling of heated magnetized rocks.

In geomagnetic field modelling, internal field separation is a well known problem. Large scales are dominated by the core field, small scales by the lithospheric field. The issue is, that they might both have significant structures at the scale they don't dominate. It is currently hard to distinguish the overlap, leading to issues with interpretation of large-scale lithospheric and small-scale core tendencies. The aim of this master's thesis is a better separation of the core and lithospheric geomagnetic field.

The thesis will be based on a recently developed geostatistical simulation tool suitable for spherical geometry. The tool is capable of generating stochastic model realizations, based on semi-variogram analysis and direct sequential simulation. It is currently tested on stochastic model simulations of a synthetic core mantle boundary field. This project will continue from these results, with the goal of including satellite magnetic data to generate well separated models of the global core and lithospheric fields. The project outline is shown below.

1. Simulation of synthetic lithospheric field **[September 2018]**
2. Inclusion of measured satellite data **[October 2018]**
3. Realization of posterior probability density function **[October/November 2018]**
4. Comparison with Markov chain Monte Carlo methods **[November 2018]**
5. Modelling **[November/December 2018]**
    - Separate core and lithosphere
    - Joint model, core+lithosphere
6. Comparison with traditional models **[December/January 2018]**
7. Documentation **[Full project period]**

Thesis project details

The first part of the project is a direct continuation of generating stochastic results with synthetic fields. A synthetic lithospheric field model is available and will be used to further test the capabilities of the previously developed tool. The additional synthetic testing will hopefully give an idea of the modelling behaviour when generating results for the lithosphere. This is important as I intend to generate models of both the core and the lithosphere. In addition, I will also look into including anisotropy in the developed tool. Currently it is assumed that the field realizations are directionally independent, while the synthetic models suggest dependence.

Once this is accomplished, real data will be included. This will be geomagnetic measurement data originating from the satellites CHAMP and Swarm. In comparison to the synthetic fields, real data will not be available at the exact grid locations of the model. It is thus necessary to include them as "soft" data. In this case, soft means inferring the data at grid locations using physical principles. This will be done using Green's functions in relation to boundary conditions, when solving Laplace's equation for the magnetic scalar potential in current free regions of the outer atmosphere.

Once data is included, it will be possible to generate conditional prior realizations, which in combination with Bayes' theorem, can be used to generate posterior realizations. Essentially, these will be collections of possible field models making out a probability density function for the field value at each grid location.

The intention is then to investigate whether the computational efficiency is fast enough, to allow the use of Markov chain Monte Carlo (MCMC) methods as a possible way to generate better posterior realizations. The modelling will be attempted for the core, the lithospheric, and the joint field.

Finally, I will scrutinize the resulting models in comparison with traditional geomagnetic field models. Here I hope to see some indication that the simulation step can aid in generating better geomagnetic field models in the future. The process and results will naturally be documented in the final thesis paper.

Appendix B

Thesis project agreement

Projektaftale, Kandidatspeciale

Aftalen indgås mellem

Institut for Rumforskning og Rumteknologi
Intet samarbejdsinstitut
og
134400, Mikkel Otzen

Vejleder(e)	Chris Finlay

Detaljer om projekt:

Dansk titel	Probabilistisk inversion af magnetisk data fra satellitter ved brug af geostatistisk simulation i sfærisk geometri
Engelsk titel	Probabilistic inversion of satellite magnetic data using geostatistical simulation in spherical geometry
Startdato	27. aug 2018
Afleveringdato	17. feb 2019
Begrundelse for overskridelse af projekttid	Has 5 ECTS course in addition to thesis. This permits 3 extra weeks
ECTS Point	30
Projekt udføres i	Danmark
Dato for aflevering af projektplan	26. sep 2018

Underskrifter

22/08/18	
Dato	Mikkel Otzen, 134400
22-08-2018	Christopher C. Finlay
Dato	For Institut for Rumforskning og Rumteknologi

Appendix C

VISIM: Creating a conditional distribution table

This section describes the core process of generating a conditional distribution table in VISIM, based on Fortran scripts from GSLIB. First a target data histogram is normal score transformed through the function *nscore.f*, then a selection of quantile functions are back transformed through the function *backtr.f*. The details of these two functions are explained below.

nscore.f

The normal transform function has input and output as given by the following box.

Input

1. Number of data [nd]
2. Data values to be transformed [vr(nd)]
3. Trimming limits [tmin/tmax]
4. Weighting to be equal or specified [iwt]
5. Weight for each data [wt(nd)]
6. Temporary storage space [tmp(nd)]
7. Write transform table file on/off [lout]
8. Use method for discrete data on/off [disc]

Output

1. Normal scores [vrg(nd)]
2. Error flag [ierror]

Given the input the function proceeds with the following computations.

Sort data values

First, data values are sorted in ascending order using a sorting algorithm.

Gaussian inverse

Running through indices *1* to *nd*, the input weights are defined by the index value and number of data.

$$wt(i) = i/nd \tag{C.0.1}$$

While running through indices, the defined weight is used as input to *gauinv.f*. The input to this function is a cumulative probability value, hence the additive weight definition. The function *gauinv.f* computes the inverse of the standard normal cumulative distribution. The computation is based on the following equations and originate from Statistical Computing, by W.J. Kennedy, Jr. and James E. Gentle, 1980, p. 95.

$$y = \sqrt{\log\left(\frac{1.0}{wt(i)^2}\right)} \tag{C.0.2}$$

$$vrg(i) = y + \frac{((((y \cdot p_4 p_3) \cdot y + p_2) \cdot y + p_1) \cdot y + p_0)}{((((y \cdot q_4 q_3) \cdot y + q_2) \cdot y + q_1) \cdot y + q_0)} \tag{C.0.3}$$

Here p_n and q_n are specific numerically determined constants used in the original algorithm.

Back-sort data values

Created arrays are sorted back such that they follow the input structure using the same sorting algorithm.

backtr.f

Back-transformation of generated quantile functions are handled by *backtr.f*. The back transform function has input and output as given by the following box.

Input

1. Normal score value to be back-transformed [vrgs]
2. Number of values [nt]
3. Data values [vr(nt)]
4. Normal transformed data values [vrg(nd)]
5. Limits for tail models [zmin/zmax]
6. Lower tail model specification [ltail]
7. Parameter for lower tail model [ltpar]
8. Upper tail model specification [utail]
9. Parameter for upper tail model [utpar]
10. Use method for discrete data on/off [discrete]

Output

1. Back transformed normal score value [vbt]

Given the input, the function proceeds with the following computations. The function only handles single values, so in order to get a full back-transform, it is iterated through the normal distribution values. In the function there are options to handle tail values with linear or power models, however, the most straightforward solution is to disable them. In that case, the back-transformation is handled solely through the power interpolation function, *powint.f*.

Power interpolation

Prior to using power interpolation, an index, j, representing the value closest to the value to be back-transformed, is located in the normal transformed data distribution, vr(nt). Then power interpolation is computed based on the following criteria.

$$vrg(j+1) - vrg(j) < \epsilon = 1.0e - 20 \tag{C.0.4}$$

If the criteria is true, the following interpolation is computed.

$$vbt = \frac{(vr(j+1) + vr(j))}{2.0} \tag{C.0.5}$$

If the criteria is not true, the following interpolation is computed.

$$vbt = vr(j) + \Big(vr(j+1) + vr(j)\Big) \cdot \left(\frac{vrgs - vrg(j)}{vrg(j+1) - vrg(j)}\right) \tag{C.0.6}$$

This continues until all the desired conditional probability distributions have been generated.

Appendix D

Extra results and figures

D.1 Sampling the prior

D.1.1 CMB field with dipole

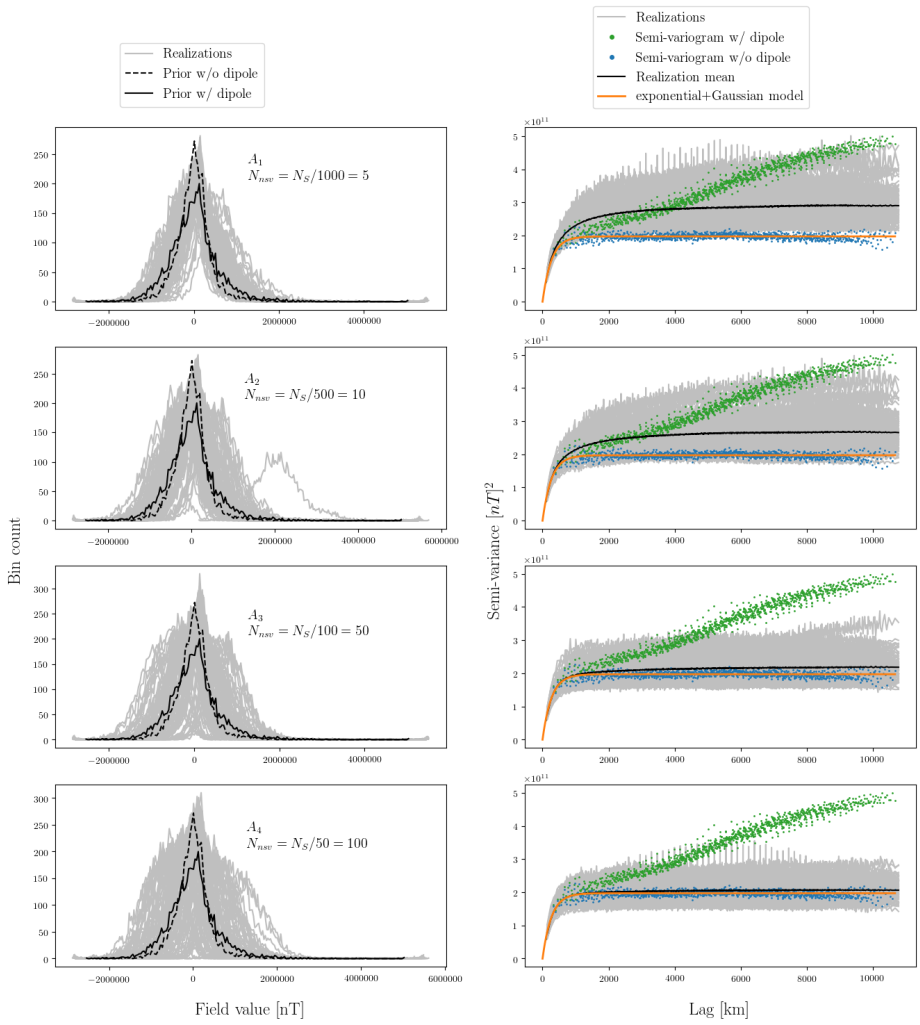
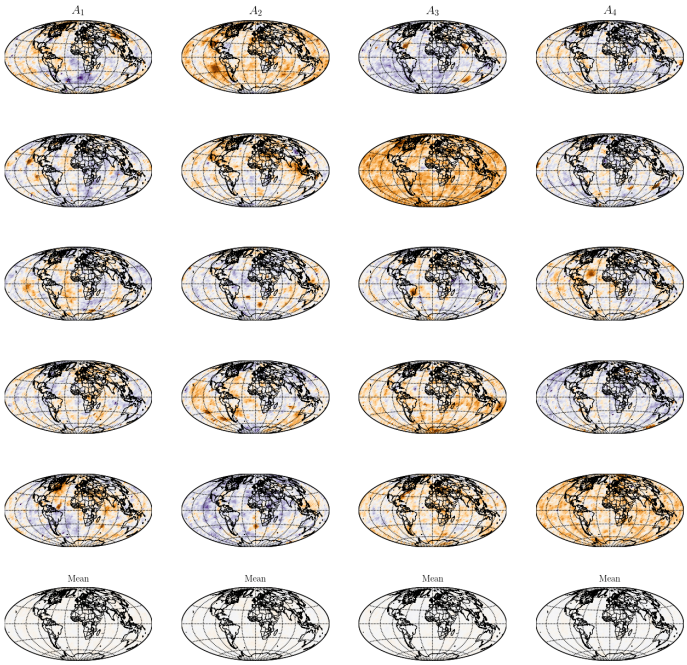
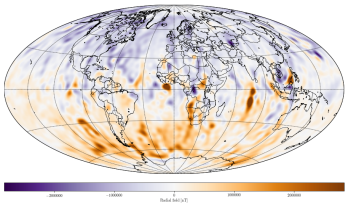


Figure D.1.1: Stochastic simulation realizations and their statistics reproduction of the full CMB prior training image. Source grid size is $N_S = 5000$ and each simulation consists of 100 realizations.

In order to test the core mantle boundary source neighborhood, four stochastic simulations are run with increasing neighborhood size. This test is carried out with the CMB training image with dipole included as target histogram. The target statistic fit results are shown in figure D.1.1 and the table below it, with examples of the generated realizations in figure D.1.2. From the fit statistics it is immediately clear that the mean of semi-variogram realizations move toward the prior model with increasing neighborhood size. However, the shape of the realization histograms seem to better follow that of the target with smaller neighborhoods. For these neighborhoods the mean of semi-variogram realizations also fit the semi-variogram w/ dipole better, though unintentionally. The histogram mean in all cases is orders of magnitude above the target, with the standard deviation being closer. As expected, computation time increases with a larger neighborhood size. From the sample realizations it is seen that these CMB field estimates often result in realizations with a large mean toward either end of the prior reference value range.



(a) Realization examples



(b) Prior reference

Figure D.1.2: **(a)** Sample realizations for each simulation run. **(b)** CMB field training image used to generate local conditional distributions.

Test	**Mean** $[nT]$	**Std.dev.** $[nT]$	**Compute time** $[hrs]$
A_1	$4.062e+04$	$5.75e+05$	0.25
A_2	$2.962e+04$	$6.24e+05$	0.26
A_3	$1.688e+04$	$5.97e+05$	0.375
A_4	$2.114e+03$	$6.54e+05$	0.64
Target	13.65	$5.9e+05$	N/A

D.2 Reproducing synthetic satellite observations

D.2.1 Test simulation using all synthetic observations

Given enough realizations of the posterior, the mean should converge to the smooth least squares solution (Hansen and Mosegaard, 2006). In this section I show the results of a simulation with 1,000 realizations conditional to all available synthetic observations, but no simulated values. This is only an approximation of the posterior, as posterior realizations require conditioning to the previously simulated values. This may be a good approximation if conditioning from previously simulated values is small. Figure D.2.1 show the relevant simulation diagnostics. The observation reproduction histogram indicates some convergence toward the target for the mean of realizations. However, it is not completely converged, indicating either a requirement for more than 1,000 realizations, that the observation information alone is not enough, or that the target can't be reached with the current system configuration (grid sizes, Kriging method, etc.). The histogram shape is reproduced, but not very tightly for individual realizations, with the same being the case for the semi-variogram fit. The integration residuals expresses the residuals of the observation prediction, if the CMB training image (with dipole) is used in the forward problem directly. As such I consider it a measure of how well the numerical integration is approximated given the geometry of the core grid used. In this case it is not a perfect approximation and better results would be gained from a larger core grid. The observation reproduction residuals are in accord with the observation reproduction histogram. The mean converges toward fitting the observations, but hasn't been reached yet.

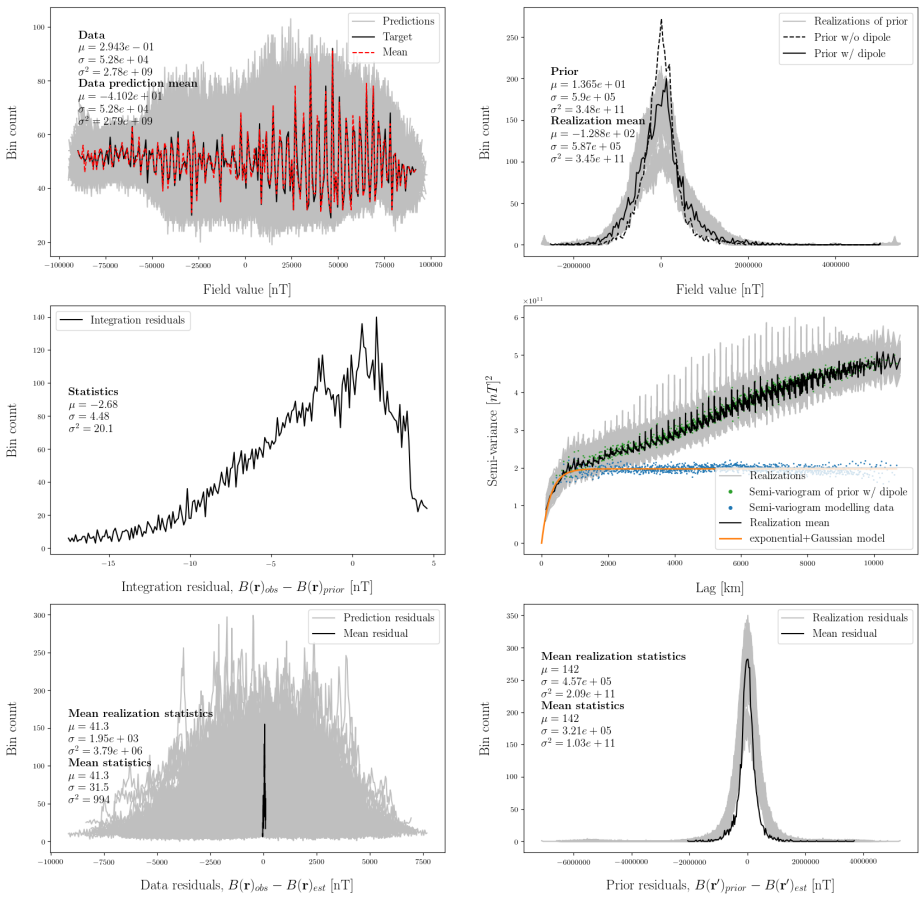


Figure D.2.1: Diagnostics for simulation of 1,000 realizations conditional to 9,998 synthetic observations, estimating the magnetic field at the core mantle boundary on a 5,000 location grid. Depicted are the observation reproduction histogram (upper left), the CMB field estimation histogram (upper right), the integration residuals (middle left), the semi-variogram fit (middle right), the observation reproduction residuals (lower left), and the residuals of the CMB field of the realizations to the synthetic true field (lower right).

Figure D.2.2 shows sample realizations and their equivalent observation predictiond when using the CMB estimate to compute the forward problem. One interesting feature points toward the CMB estimates being posterior realizations that reproduce the smooth least squares solution for the mean of realizations. The smooth least squares solution later shown in figure 4.3.7 (the solution based on dipole removal) has a distinct triangular feature with positive radial field value, depicted in the superimposed Indian Ocean. This triangular feature is present in the estimates shown in figure D.2.2, whereas a smooth rounded feature is found on the small neighborhood based estimates of figure 4.3.6, as well as on the dipole inclusion LSQ estimate at the bottom of figure 4.3.7.

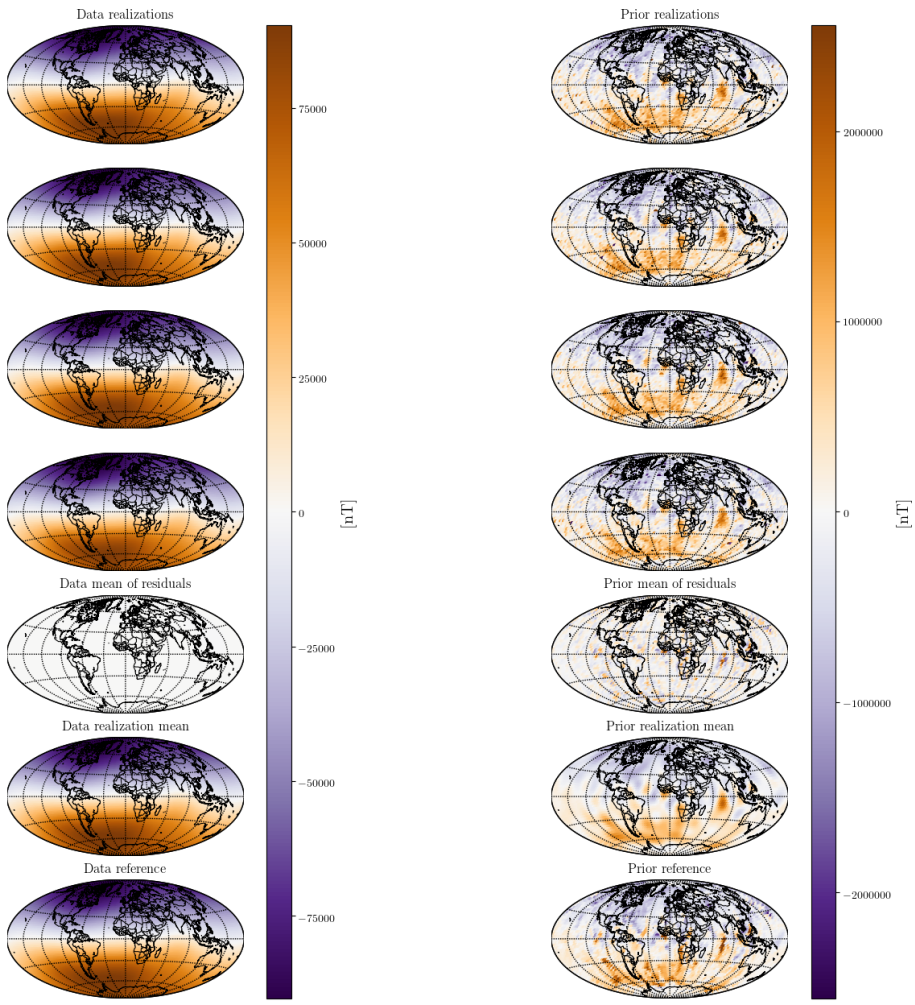


Figure D.2.2: Sample realizations for simulation of 1,000 realizations conditional to 9,998 synthetic observations, estimating the core mantle boundary on a 5,000 location grid. Included are a plot of the residuals at the CMB and for the observation reproduction at satellite altitude, as well as the mean of realizations, conditional observations, and the CMB training image.

D.2.2 Smooth least squares solution comparison for target histogram choice

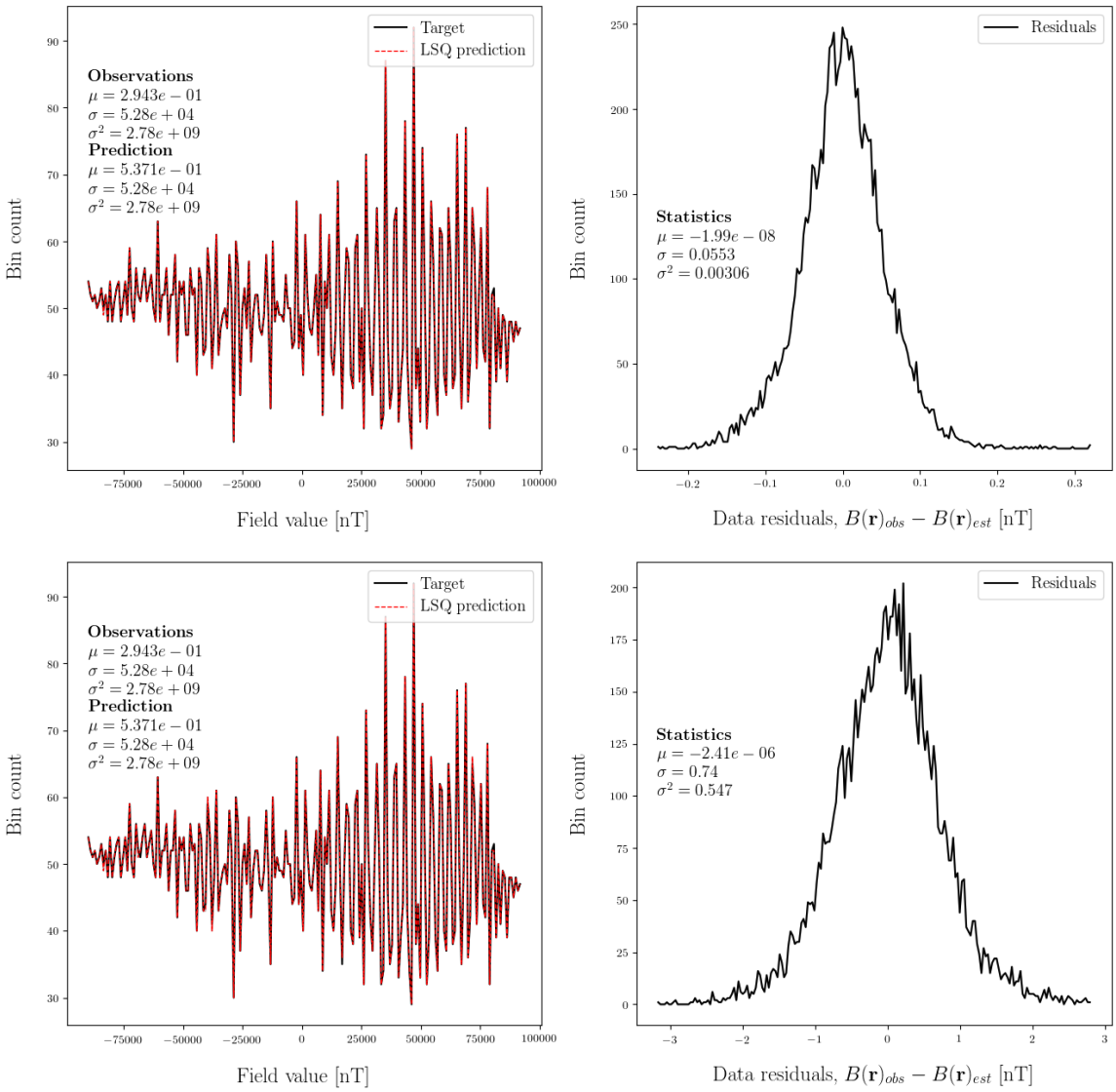


Figure D.2.3: Observation reproduction and residuals for a sequential least squares estimation using 3,000 synthetic satellite observations at 300 km above Earth's surface with 1 nT noise added. The top plots show the result for a prior histogram without the dipole and the bottom plots for the full prior histogram.

D.3 Observations + prior

D.3.1 Reproducing target statistics

Using both synthetic observations and previously simulated values require the combined neighborhoods explored in section 4.2 and 4.3. Figure D.3.1 show reproduction of the target statistics using similar neighborhoods as previously, in a combined simulation. Histogram reproduction is similar independent of synthetic observation neighborhood size, but semi-variogram reproduction still require a certain threshold to be reached. Previously investigated was a synthetic observation neighborhood of $N_D/20$, where N_D is the total observations. Figure D.3.1 show that reproduction is still ensured for the larger neighborhood of size $N_D/10$. Small scale semi-variogram reproduction still seems insured by smaller neighborhood sizes, such as the shown $N_S/100$.

Test overview for reproducing target statistics at the CMB

Neighborhood sizes	CMB	Observations
A_1	$N_S/100$	$N_D/50$
A_2	$N_S/100$	$N_D/10$
Semi-variogram type		exp + Gau
Realizations		100
Conditional dist. type		no dipole

Table D.3.1

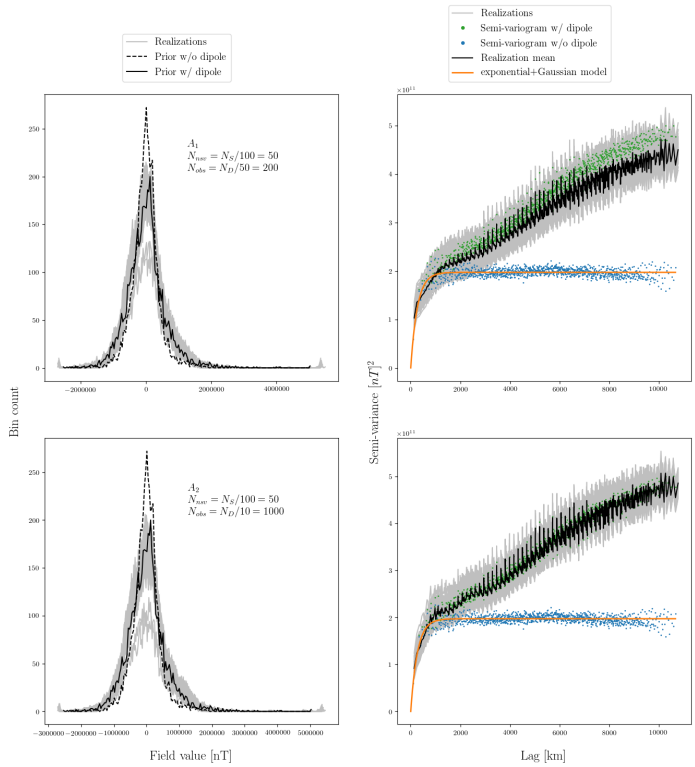


Figure D.3.1: Training image statistics fit for posterior realizations using both synthetic observations and previously simulated values.

D.3.2 All observations + prior

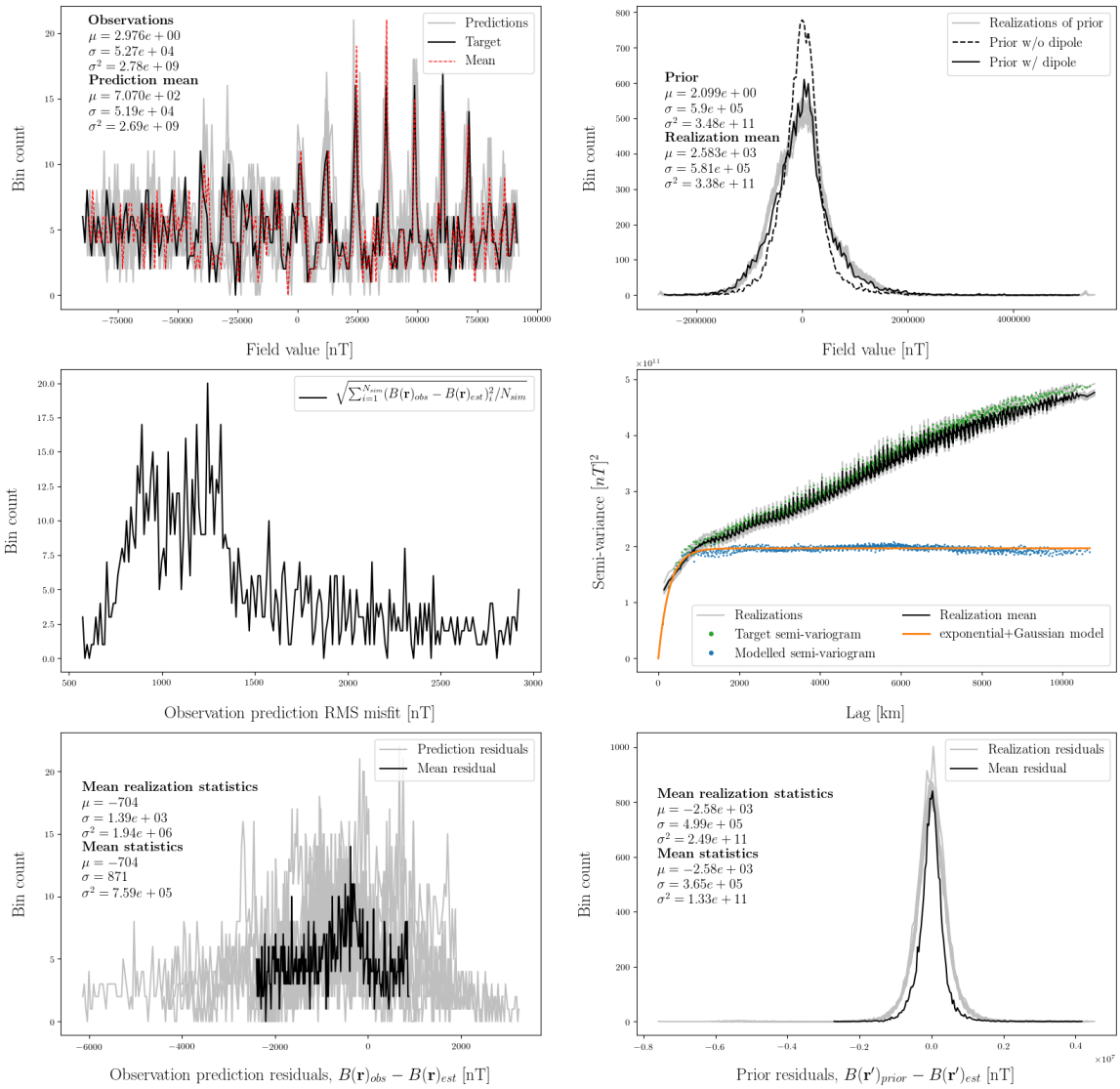


Figure D.3.2: Diagnostics for DSSIM of 10 realizations using 1,000 synthetic observations with a target source grid of 15,000 locations. The simulation is conditional to previously simulated values and all available synthetic observations. Depicted are an observation reproduction histogram (upper left), a CMB estimation histogram (upper right), root-mean-square misfits to the observations (middle left), the semi-variogram fit (middle right), observation reproduction residuals (lower left), and the CMB estimation residuals (lower right).

Appendix E

SDSSIM_1.2: geostatistics Python tool

E.1 SDSSIM_setup

```
def SDSSIM_setup(*args, setup_type = 'core', N = 'default', condtab_compiler = ...
    'Python', savefig = 'no', dpi = 100, figsize=(16,16), gensize=18, fontsize=18, ...
    fontsize_sub = 14):
    import matplotlib.pyplot as plt
    cmap = plt.cm.PuOr_r
    #cmap = plt.cm.Spectral_r

    if setup_type == 'core':
        grid_radius = 3480.0 # Earth core mantle boundary
        type_model = 'Julien_core'
        unit = '[nT]'
        n_sh = 60
        if N == 'default':
            N = 30000
    elif setup_type == 'surface':
        grid_radius = 6371.2
        type_model = 'Masterton_surface'
        unit = '[nT]'
        n_sh = 100
        if N == 'default':
            N = 30000
    elif setup_type == 'sat':
        grid_radius = 6371.2 + 300.0
        type_model = 'Julien_synth_dat_300km'
        unit = '[nT]'
        n_sh = 60
        if N == 'default':
            N = 20000
    elif setup_type == "swarm":
        grid_radius = None
        type_model = "swarm"
        unit = '[nT]'
        n_sh = None

    sdssim_setup = {"N":N, "type":setup_type, "type_model":type_model, ...
        "unit":unit, "n_sh":n_sh, "grid radius":grid_radius, "figsize":figsize, ...
        "gensize":gensize, "fontsize":fontsize, "dpi":dpi, "savefig":savefig, ...
        "cmap":cmap, "condtab_compiler":condtab_compiler, "fontsize_sub":fontsize_sub}

    return sdssim_setup
```

E.2 SDSSIM_grid

```
def calc_spherical_distances(grid_core, setup_core):
    from SDSSIM_utility import haversine
    import numpy as np
    lat_mesh, lon_mesh = np.meshgrid(grid_core["grid latitude"],grid_core["grid ...
        longitude"])
    grid_core["grid spherical distances"] = haversine(setup_core["grid radius"], ...
        lon_mesh, lat_mesh, lon_mesh.T, lat_mesh.T)
    return grid_core

def eqsp(N,dim):

    """
    Calculate equal area spherical partitioned grid and plot

    """

    """
    Initialization
    """
    import matlab
    import matlab.engine
    #from mpl_toolkits.basemap import Basemap
    #import matplotlib.pyplot as plt
    import numpy as np
    #import scipy.io as sio
    N = N
    dim = dim

    """
    Start MATLAB Engine API for Python
      -This enables calls to Matlab functions through Python.
    """

    eng = matlab.engine.start_matlab()

    """
    Set up floats suitable for inputs to Matlab functions
      dim: dimension of S^d unit sphere being embedded into R^(d+1).
      N: number of wanted regions of equal area and on the unit sphere.
    """

    if 'dim' not in locals():
        print('Dimension (dim) not specified')
        return

    if 'N' not  in locals():
        print('Grid size (N) not specified')
        return

    if not isinstance(dim, float):
        print('Dimension (dim) must be float type')
        return

    if not isinstance(N, float):
        print('Grid size (N) must be float type')
        return

    """
    Calling Matlab functions from the eq_sphere_partitions toolbox:
      eng.double(Matlab double output): Python class to hold array of MATLAB type ...
          double.
      eng.eq_point_set_polar(dim,N): Coordinates of central point for each ...
          partitioned area in polar coordinates
      eng.eq_point_set(dim,N): Coordinates of central point for each partitioned ...
          area in cartesian coordinates
    """

    points_polar = eng.double(eng.eq_point_set_polar(dim,N))
    points_cart = eng.double(eng.eq_point_set(dim,N))
```

```
    """
    Import numpy library for matrix manipulation
       np.asmatrix(Python array): Converts Python array to matrix.
    """

    points_polar = np.asmatrix(points_polar)
    points_cart = np.asmatrix(points_cart)


    longitude = points_polar[0,:]*180/np.pi
    latitude = 90 - points_polar[1,:]*180/np.pi

    grid = np.asmatrix(np.array([np.asarray(longitude), ...
        np.asarray(latitude)]).reshape(2,int(N)))

    lon_s = np.asarray(grid[0,:]).ravel()
    lat_s = np.asarray(grid[1,:]).ravel()

    s_cap, n_regions = eng.eq_caps(dim,N,nargout=2) # nargout = number of outputs ...
        (for when you know a function returns more than one output)
    s_cap = np.asmatrix(eng.double(s_cap))
    n_regions = np.asmatrix(eng.double(n_regions))



    #eqsp_cap_region = eng.double(eng.eq_caps(dim,N))
    #eqsp_cap_region  = np.asmatrix(eqsp_cap_region)

    """
    Spherical distance between all points
      -Each row in sph_d_all is spherical distance from one point to all others,
       going through the points as ordered in points_cart.
    """

    # Runtime is very long for 30.000 points

    #sph_d_all = np.zeros([int(N),int(N)])

    #for x in range(0, int(N)):
    #    points_cart = np.asmatrix(points_cart)
    #    points = np.multiply(points_cart[:,x], np.mat(np.ones([3,int(N)])))

    #    points_cart = matlab.double(np.asarray(points_cart).tolist())
    #    points = matlab.double(np.asarray(points).tolist())

    #    spherical_distance = eng.double(eng.spherical_dist(points,points_cart))
    #    spherical_distance = np.asmatrix(spherical_distance)
    #    print(x)
    #    sph_d_all[x,:] = spherical_distance

    """
    Stop MATLAB Engine API for Python
    """
    eng.quit()


    EQSP = {'N':N, 'dim':dim, 'Polar coords':points_polar, 'Latitude':lat_s, ...
        'Longitude':lon_s, 's_cap':s_cap, 'n_regions':n_regions}
    #EQSP = {'N':N, 'dim':dim, 'Polar coords':points_polar, 'Latitude':lat_s, ...
        'Longitude':lon_s, 'eqsp_cap_region':eqsp_cap_region}

    #return EQSP, sph_d_all
    return EQSP

def eqsp_grid(N, grid_radius, sph_d_loadmat=False, sph_d_loadpy=True, ...
    latlon_load=True):
    import hdf5storage
    import numpy as np

    grid_sph_d = None
    s_cap = None
    n_regions = None
```

```
    #eqsp_cap_region = None

    if sph_d_loadmat == True:
        mat = hdf5storage.loadmat('saved_variables/sph_d_all.mat')
        grid_sph_d = mat['sph_d_all']

    if sph_d_loadpy == True:
        from SDSSIM_utility import variable_load
        grid_sph_d = variable_load('saved_variables/grid_sph_d.npy')
        grid_sph_d = np.multiply(grid_radius,grid_sph_d)

    if latlon_load == True:
        from SDSSIM_utility import variable_load
        grid_lat = variable_load('saved_variables/grid_lat.npy')
        grid_lon = variable_load('saved_variables/grid_lon.npy')
    else:
        EQSP = eqsp(float(N),2.0)
        grid_lat = EQSP['Latitude']
        grid_lon = EQSP['Longitude']
        s_cap = EQSP['s_cap']
        n_regions = EQSP['n_regions']
        #eqsp_cap_region = EQSP['eqsp_cap_region']

    return grid_lat, grid_lon, grid_sph_d, s_cap, n_regions

def SDSSIM_grid(N, grid_radius, *args, savegrid = False, custom_grid = False, ...
    calc_sph_d = False):
    """
    Grid function setting up the grid used in SDSSIM
        grid_lat: [N, np.array]
        grid_lon: [N, np.array]
        grid_sph_d: [N-by-N, np.matrix]
    """

    if savegrid == True:
        from SDSSIM_utility import variable_save
        import numpy as np
        grid_lat, grid_lon, grid_sph_d, s_cap, n_regions = eqsp_grid(N, ...
            grid_radius, sph_d_loadmat=True, sph_d_loadpy=False, latlon_load = False)
        variable_save('saved_variables/grid_lat', grid_lat)
        variable_save('saved_variables/grid_lon', grid_lon)
        variable_save('saved_variables/grid_sph_d', grid_sph_d)
        print('Spherical distances saved are for a unit sphere. Returned spherical ...
            distances are scaled to grid radius %0.1f km' % grid_radius)
        grid_sph_d = np.multiply(grid_radius,grid_sph_d)
        sdssim_grid = {"N":N, "grid latitude":grid_lat, "grid longitude":grid_lon, ...
            "grid spherical distances":grid_sph_d, "grid radius":grid_radius}
    else:
        if custom_grid == False:
            grid_lat, grid_lon, grid_sph_d, s_cap, n_regions = eqsp_grid(N, ...
                grid_radius, sph_d_loadmat=False, sph_d_loadpy=True, latlon_load = ...
                False)
        else:
            if calc_sph_d == False:
                grid_lat, grid_lon, grid_sph_d, s_cap, n_regions = eqsp_grid(N, ...
                    grid_radius, sph_d_loadmat=False, sph_d_loadpy=False, ...
                    latlon_load = False)
            else:
                import numpy as np
                from SDSSIM_utility import haversine
                grid_lat, grid_lon, grid_sph_d, s_cap, n_regions = eqsp_grid(N, ...
                    grid_radius, sph_d_loadmat=False, sph_d_loadpy=False, ...
                    latlon_load = False)
                lat_mesh, lon_mesh = np.meshgrid(grid_lat,grid_lon)
                grid_sph_d = haversine(grid_radius, lon_mesh, lat_mesh, ...
                    lon_mesh.T, lat_mesh.T)

        sdssim_grid = {"N":N, "grid latitude":grid_lat, "grid longitude":grid_lon, ...
            "grid spherical distances":grid_sph_d, "grid radius":grid_radius, ...
            "s_cap":s_cap, "n_regions":n_regions}
    return sdssim_grid
```

E.3 SDSSIM_data

```
"""
data for SDSSIM
"""

def rem_mean_lat(data, lat):
    import numpy as np
    lat_lim_save = np.empty([0,0],dtype=int)
    lat_mean_save = np.empty([0,],dtype=float)
    for n in range(-90,90):
        lat_lim = np.logical_and(n≤lat, lat≤(n+1))
        if any(lat_lim):
            lat_mean = np.mean(data[lat_lim])
            data[lat_lim] = data[lat_lim] - lat_mean
            lat_lim_save = np.append(lat_lim_save, np.argwhere(lat_lim))
            lat_mean_save = np.append(lat_mean_save, ...
                lat_mean*np.ones(len(np.argwhere(lat_lim))))

    lat_return_mean = np.array([lat_lim_save,lat_mean_save]).T
    return data, lat_return_mean


def synth_model(N, lat, lon, r = 6371.2, n_deg=60, type_model = 'Julien_core', ...
    custom_gauss = None, remmean = False, dipole = True):
    import numpy as np
    import GMT_tools as gt

    rad = np.pi/180
    a = 6371.2

    A_r, A_theta, A_phi = gt.design_SHA(r/a*np.ones([N-2]), ...
        (90.0-lat[1:-1])*rad,lon[1:-1]*rad, n_deg)

    G = np.vstack((A_r, A_theta, A_phi))

    if np.logical_or(type_model == 'Julien_core', type_model == ...
        'Julien_synth_dat_300km'):
        if dipole == True:
            Gauss_in = np.loadtxt('sh_models/Julien_Gauss_JFM_E-8_snap.dat')
        else:
            Gauss_in = np.loadtxt('sh_models/Julien_Gauss_JFM_E-8_snap_nodip.dat')
        print('Loading Julien model')
    elif type_model == 'Masterton_surface':
        Gauss_in = np.loadtxt('sh_models/Masterton_13470_total_it1_0.glm')
        print('Loading Masterton model')
    else:
        Gauss_in = np.loadtxt(custom_gauss, comments='%')
        print('Loading %s' %custom_gauss)

    i=0
    i_line=0

    g = np.zeros(len(A_r.T))

    for n in range(1,n_deg+1):
        for m in range(0,n+1):
            if m == 0:
                g[i]=Gauss_in[i_line,2]
                i += 1
                i_line += 1
            else:
                g[i]=Gauss_in[i_line,2]
                g[i+1]=Gauss_in[i_line,3]
                i+= 2
                i_line += 1


    data_dynamo = np.matrix(G)*np.matrix(g).T
    data = np.array(data_dynamo[:len(A_r)]).ravel()

```

```
    data_complete = np.zeros((N,))
    data_complete[1:-1] = data

    if remmean == True:
        data_complete, lat_return_mean = rem_mean_lat(data_complete, lat)
        print('Finished loading synthetic model')
        return data_complete, lat_return_mean


    print('Finished loading synthetic model')
    return data_complete

def SDSSIM_data(setup, grid, *args, **kwargs):
    import numpy as np
    lat_return_mean = None
    data_lat = None
    data_lon = None
    N = grid["N"]
    if np.logical_or(setup["type"] == 'core', setup["type"] == 'sat'):
        remmean = kwargs["remmean"]
        dipole = kwargs["dipole"]
        if remmean == False:
            data = synth_model(grid["N"], grid["grid latitude"], grid["grid ...
                longitude"], r = grid["grid radius"], n_deg=setup["n_sh"], ...
                type_model = setup["type_model"], remmean = remmean, dipole = dipole)
        else:
            data, lat_return_mean = synth_model(grid["N"], grid["grid latitude"], ...
                grid["grid longitude"], r = grid["grid radius"], ...
                n_deg=setup["n_sh"], type_model = setup["type_model"], remmean = ...
                remmean, dipole = dipole)
        data_lon = grid["grid longitude"]
        data_lat = grid["grid latitude"]
        data_radius = np.ones(grid["N"],)*setup['grid radius']

    elif setup["type"] == 'surface':

        data = synth_model(grid["N"], grid["grid latitude"], grid["grid ...
            longitude"], r = grid["grid radius"], n_deg=setup["n_sh"], type_model ...
            = setup["type_model"])
        data_lon = grid["grid longitude"]
        data_lat = grid["grid latitude"]
        data_radius = np.ones(grid["N"],)*setup['grid radius']
    else:
        print("No correct data type chosen")
        return

    sdssim_data = {"data":data, "latitude mean":lat_return_mean, "data ...
        radius":data_radius, "data latitude":data_lat, "data longitude":data_lon, ...
        "N":N}

    return sdssim_data

def handle_poles(grid_core, data_core, grid_sat, data_sat, setup_sat):
    import numpy as np

    data_core["data"][0] = ...
        np.mean(data_core["data"][1:int(grid_core["n_regions"][0,1])])
    data_core["data"][-1] = ...
        np.mean(data_core["data"][-int(grid_core["n_regions"][0,1]):][:-1])

    if np.logical_and(data_sat is not None, grid_sat is not None):
        idx_end_sat = grid_sat["N"]-1
        grid_sat["grid latitude"] = np.delete(grid_sat["grid ...
            latitude"],[0,idx_end_sat],0)
        grid_sat["grid longitude"] = np.delete(grid_sat["grid ...
            longitude"],[0,idx_end_sat],0)
        grid_sat["N"] = idx_end_sat-1

        data_sat["data latitude"] = np.delete(data_sat["data ...
            latitude"],[0,idx_end_sat],0)
        data_sat["data longitude"] = np.delete(data_sat["data ...
            longitude"],[0,idx_end_sat],0)
        data_sat["data radius"] = np.delete(data_sat["data radius"],[0,idx_end_sat],0)
```

```
        data_sat["data"] = np.delete(data_sat["data"],[0,idx_end_sat],0)
        data_sat["N"] = idx_end_sat-1
        setup_sat["N"] = idx_end_sat-1

        if grid_sat["grid spherical distances"] is not None:
            grid_sat["grid spherical distances"] = np.delete(grid_sat["grid ...
                spherical distances"],[0,idx_end_sat],0)
            grid_sat["grid spherical distances"] = np.delete(grid_sat["grid ...
                spherical distances"],[0,idx_end_sat],1)


        return data_core, grid_sat, data_sat, setup_sat
    else:
        return data_core
```

E.4 SDSSIM_semivar

```
"""
Semi-variogram functions
"""

def find_sort_d(grid_core, max_dist = 2000):
    import numpy as np
    range_d = grid_core["grid spherical distances"].ravel() < max_dist
    idx_range = np.array(np.where(range_d == True)).ravel()
    val_range = grid_core["grid spherical distances"].ravel()[idx_range]
    idx_sort_val_range = np.argsort(val_range)
    sort_d = idx_range[idx_sort_val_range]
    return sort_d

def semivariogram_model(h, a, C0, C1, C2 = None, C3 = None, mode = 'spherical'):
    import numpy as np
    if mode == 'spherical':
        '''
        Spherical model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*( 1.5*hla/a - 0.5*(hla/a)**3 )
        model[len(hla):] = C0 + C1
        model = model[hir]

    elif mode == 'dub_spherical':
        '''
        Spherical model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]

        model[0:len(hla)] = C0 + C1*( 1.5*hla/a - 0.5*(hla/a)**3 ) + C2*( ...
            1.5*hla/C3 - 0.5*(hla/C3)**3)
        model[len(hla):] = C0 + C1 + C2*( 1.5*hs[len(hla):]/C3 - ...
            0.5*(hs[len(hla):]/C3)**3)
        model[C3:] = C0 + C1 + C2
        model = model[hir]

```

```
    elif mode == 'gaussian':
        '''
        Gaussian model of the semivariogram
        '''
        model = C0 + C1*(1-np.exp(-(3*h)**2/a**2))
    elif mode == 'exponential':
        '''
        Exponential model of the semivariogram
        '''
        import numpy as np

        model = C0 + C1*(1-np.exp(-3*h/a))

    elif mode == 'power':
        '''
        Power model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*hla**a
        model[len(hla):] = C0 + C1*np.array(hs[len(hla):])**a
        model = model[hir]

    elif mode == 'hole':
        '''
        Hole model of the semivariogram
        '''
        model = C0 + C1*(1-np.cos(h/a*np.pi))

    elif mode == 'hole_damp':
        '''
        Hole model of the semivariogram
        '''
        model = C0 + C1*(1-np.exp(-3*h/C2)*np.cos(h/a*np.pi))

    elif mode == 'nested_hole_gau':
        '''
        Hole model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*(1-np.cos(hla/a*np.pi)) + ...
            C2*(1-np.exp(-(3*hla)**2/a**2))
        model[len(hla):] = C0 + C1*(1-np.cos(np.array(hs[len(hla):])/a*np.pi)) + ...
            C2*(1-np.exp(-(3*np.array(hs[len(hla):]))**2/a**2))
        model = model[hir]

    elif mode == 'nested_sph_gau':
        '''
        Nested spherical and gaussian model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*( 1.5*hla/a - 0.5*(hla/a)**3 ) + ...
            C2*(1-np.exp(-(3*hla)**2/a**2))
```

```
        model[len(hla):] = C0 + C1 + ...
            C2*(1-np.exp(-(3*np.array(hs[len(hla):]))**2/a**2))
        model = model[hir]

    elif mode == 'nested_sph_exp':
        '''
        Nested spherical and exponential model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*( 1.5*hla/a - 0.5*(hla/a)**3 ) + ...
            C2*(1-np.exp(-(3*hla)/a))
        model[len(hla):] = C0 + C1 + C2*(1-np.exp(-(3*np.array(hs[len(hla):]))/a))
        model = model[hir]

    elif mode == 'nested_exp_gau':
        '''
        Nested exponential and gaussian model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*(1-np.exp(-(3*hla)/a)) + ...
            C2*(1-np.exp(-(3*hla)**2/a**2))
        model[len(hla):] = C0 + C1*(1-np.exp(-(3*np.array(hs[len(hla):]))/a)) + ...
            C2*(1-np.exp(-(3*np.array(hs[len(hla):]))**2/a**2))
        model = model[hir]

    elif mode == 'nested_sph_exp_gau':
        '''
        Nested spherical and exponential model of the semivariogram
        '''

        hi = np.argsort(h)
        hir = np.argsort(hi)

        model = np.zeros(len(h))

        hs = h[hi]
        hla = hs[hs<a]
        model[0:len(hla)] = C0 + C1*( 1.5*hla/a - 0.5*(hla/a)**3 ) + ...
            C2*(1-np.exp(-(3*hla)/a)) + C3*(1-np.exp(-(3*hla)**2/a**2))
        model[len(hla):] = C0 + C1 + C2*(1-np.exp(-(3*np.array(hs[len(hla):]))/a)) ...
            + C3*(1-np.exp(-(3*np.array(hs[len(hla):]))**2/a**2))
        model = model[hir]

    else:
        print('Unknown model type')
        return

    return model

def varioLUT(distance, N, a, C0, C1, C2 = None, C3 = None, model = 'spherical'):
    import numpy as np
    from SDSSIM_utility import printProgressBar
    '''
    semi-variogram LUT generation
    '''
    vario_lut = np.zeros([N,N])

    for i in range(0,N):
```

```
        vario_lut[:,i] = ...
            semivariogram_model(distance[i,:],a,C0,C1,C2=C2,C3=C3,mode=model)
        printProgressBar (i, N, subject = 'Semi-variogram LUT progress')
    return vario_lut

def data_variogram(N, data, r, load = None, sort_d = None, sph_d_all = None):
    """
    Function for calculating or loading variogram from data
    """
    import numpy as np
    from SDSSIM_utility import variable_load, printProgressBar

    if load is None:
        cloud_all = np.zeros([N,N])

        print("")
        print('Generating variogram cloud')
        print("")

        for i in range(0,N):
            printProgressBar (i, N, subject = 'Variogram cloud progress')
            cloud = (data[i]-data)**2
            cloud_all[i,:] = cloud

        cloud_sorted = cloud_all.ravel()[sort_d]

        print("")
        print("")
        print('Variogram cloud generated and sorted')
        print("")

        if sort_d is None:
            sort_d = variable_load('saved_variables/sort_d_short.npy')

        if sph_d_all is None:
            sph_d_sorted = variable_load('saved_variables/sph_d_ravel_short_sort.npy')
            sph_d_sorted = np.multiply(r,sph_d_sorted)

        else:
            print('Sorting spherical distances')
            print("")
            sph_d_sorted = sph_d_all.ravel()[sort_d]
            print('Spherical distances sorted')
            print("")


    else:
        sph_d_sorted = variable_load('saved_variables/sph_d_ravel_short_sort.npy')
        if load is 'Julien':
            cloud_sorted = ...
                variable_load('saved_variables/cloud_ravel_short_sort_Julien.npy')
            sph_d_sorted = np.multiply(r,sph_d_sorted)
        elif load is 'Masterton':
            cloud_sorted = ...
                variable_load('saved_variables/cloud_ravel_short_sort_Masterton_nT.npy')
            sph_d_sorted = np.multiply(r,sph_d_sorted)
        else:
            print('Wrong load name, use Julien or Masterton')
            return

    return sph_d_sorted, cloud_sorted

def data_semivariogram(max_cloud, n_lags, sph_d_ravel_short, cloud_ravel_short, N):
    """
    Function for calculating semivariogram from data by taking the mean of
    equidistant lags
    """

    import numpy as np
    from SDSSIM_utility import printProgressBar

    print('Generating semi-variogram cloud')
    print("")
```

```python

    lag = int(max_cloud/n_lags)

    pics = np.zeros(n_lags-1)
    lags = np.zeros(n_lags-1)

    for n in range(1,n_lags):
        printProgressBar (n, n_lags, subject = 'Semi-variogram cloud progress')

        pic = 0.5*np.mean(cloud_ravel_short[(n-1)*lag:lag*n:1])

        pics[n-1] = pic

        lag_u = np.max(sph_d_ravel_short[(n-1)*lag:lag*n:1])

        lag_l = np.min(sph_d_ravel_short[(n-1)*lag:lag*n:1])

        lag_c = (lag_u-lag_l)/2+lag_l

        lags[n-1] = lag_c

    print("")
    print("")
    print('Semi-variogram cloud generated')
    print("")
    return pics, lags

def sv_sim_cloud(lag_coarse, cloud_coarse, rz, zs, N, sort_d, data_type = 0):
    """
    Function for calculating semivariogram from simulations by taking the mean of
    equidistant lags
    """
    from SDSSIM_utility import variable_load
    import numpy as np

    cloud_zs = np.zeros([lag_coarse,rz])

    if data_type == 'Julien':
        sort_d = variable_load('saved_variables/sort_d_short.npy')

    for j in range(0,rz):
        cloud_all = np.zeros([N,N])
        if rz>10:
            #print('Computing sv for realization: ',j)
            from SDSSIM_utility import printProgressBar
            printProgressBar (j, rz, subject = 'Computing semi-variograms for ...
                realizations')
        for i in range(0,N):
            cloud = (zs[i,j]-zs[:,j])**2
            cloud_all[i,:] = cloud

        #% """"""""""""""""""Point cloud""""""""""""""""""

        cloud_ravel = cloud_all.ravel()
        cloud_ravel = cloud_ravel[sort_d]

        pics_c = np.zeros(lag_coarse)

        for n in range(0,lag_coarse):
            pic = 0.5*np.mean(cloud_ravel[n*cloud_coarse:cloud_coarse*(n+1)])
            pics_c[n] = pic

        cloud_zs[:,j] = pics_c
    return cloud_zs

def SDSSIM_semivar(data, grid, setup, *args, model_lags = 'all', model = ...
    'nested_sph_exp_gau', sort_d = None, sph_d_all = None, lag_length = 5, nolut = ...
    False, bounds = True, zero_nugget = False, set_model = False, load = None):
    from math import inf
    import numpy as np
    from scipy.optimize import curve_fit

```

```
    sph_d_sorted, cloud_sorted = data_variogram(data['N'], data['data'], ...
        setup['grid radius'], load = load, sort_d = sort_d, sph_d_all = sph_d_all)

    max_cloud = len(sort_d)
    d_max = np.max(sph_d_sorted)

    n_lags = int(d_max/lag_length) # lags from approx typical distance between ...
        core grid points

    print("____semi-variogram setup___")
    print("")
    print("Number of data used: %d" %max_cloud)
    print("Max data distance: %.3f km" %d_max)
    print("Lag length chosen: %.1f km" %lag_length)
    print("Number of lags: %d" %n_lags)
    print("Number of modelling lags:",model_lags)
    print("")

    pics, lags = data_semivariogram(max_cloud, n_lags, sph_d_sorted, cloud_sorted, ...
        data["N"])

    print('Generating semi-variogram model')
    print("")

    if model_lags == 'all':
        lags_model = lags
        pics_model = pics
    else:
        lags_model = lags[:model_lags]
        pics_model = pics[:model_lags]

    model_name = {'spherical':'spherical', 'dub_spherical':'double spherical', ...
        'gaussian':'gaussian', 'exponential':'exponential', 'power':'power', ...
        'hole':'hole', 'hole_damp':'dampened hole', ...
        'nested_hole_gau':'hole+Gaussian', 'nested_sph_gau':'spherical+Gaussian', ...
        'nested_sph_exp':'spherical+exponential', ...
        'nested_exp_gau':'exponential+Gaussian', ...
        'nested_sph_exp_gau':'spherical+exponential+Gaussian'}

    """ZERO NUGGET OR NOT"""
    if set_model == False:
        if model == 'spherical':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1):
                    return C0 + C1*(1.5*lags_model/a-0.5*(lags_model/a)**3)
            else:
                def semivar_return(lags_model, a, C1):
                    return C1*(1.5*lags_model/a-0.5*(lags_model/a)**3)
        elif model == 'dub_spherical':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1, C2, C3):
                    return C0 + C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                        C2*(1.5*lags_model/C3-0.5*(lags_model/C3)**3)
            else:
                def semivar_return(lags_model, a, C1, C2, C3):
                    return C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                        C2*(1.5*lags_model/C3-0.5*(lags_model/C3)**3)
        elif model == 'gaussian':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1):
                    return C0 + C1*(1-np.exp(-(3*lags_model)**2/a**2))
            else:
                def semivar_return(lags_model, a, C1):
                    return C1*(1-np.exp(-(3*lags_model)**2/a**2))
        elif model == 'exponential':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1):
                    return C0 + C1*(1-np.exp(-3*lags_model/a))
            else:
                def semivar_return(lags_model, a, C1):
                    return C1*(1-np.exp(-3*lags_model/a))
        elif model == 'power':
            if zero_nugget == False:
```

```
                def semivar_return(lags_model, a, C0, C1):
                    return C0 + C1*lags_model**a
            else:
                def semivar_return(lags_model, a, C1):
                    return C1*lags_model**a
        elif model == 'hole':
            def semivar_return(lags_model, a, C0, C1):
                return C0 + C1*(1-np.cos(lags_model/a*np.pi))
        elif model == 'hole_damp':
            def semivar_return(lags_model, a, C0, C1, C2):
                return C0 + C1*(1-np.exp(-3*lags_model/C2)*np.cos(lags_model/a*np.pi))
        elif model == 'nested_hole_gau':
            def semivar_return(lags_model, a, C0, C1, C2):
                return C0 + C1*(1-np.cos(lags_model/a*np.pi)) + ...
                    C2*(1-np.exp(-(3*lags_model)**2/a**2))
        elif model == 'nested_sph_gau':
            def semivar_return(lags_model, a, C0, C1, C2):
                return C0 + C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                    C2*(1-np.exp(-(3*lags_model)**2/a**2))
        elif model == 'nested_sph_exp':
            def semivar_return(lags_model, a, C0, C1, C2):
                return C0 + C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                    C2*(1-np.exp(-(3*lags_model)/a))
        elif model == 'nested_exp_gau':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1, C2):
                    return C0 + C1*(1-np.exp(-(3*lags_model)/a)) + ...
                        C2*(1-np.exp(-(3*lags_model)**2/a**2))
            else:
                def semivar_return(lags_model, a, C1, C2):
                    return C1*(1-np.exp(-(3*lags_model)/a)) + ...
                        C2*(1-np.exp(-(3*lags_model)**2/a**2))
        elif model == 'nested_sph_exp_gau':
            if zero_nugget == False:
                def semivar_return(lags_model, a, C0, C1, C2, C3):
                    return C0 + C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                        C2*(1-np.exp(-(3*lags_model)/a)) + ...
                        C3*(1-np.exp(-(3*lags_model)**2/a**2))
            else:
                def semivar_return(lags_model, a, C1, C2, C3):  # FOR ZERO NUGGET
                    return C1*(1.5*lags_model/a-0.5*(lags_model/a)**3) + ...
                        C2*(1-np.exp(-(3*lags_model)/a)) + ...
                        C3*(1-np.exp(-(3*lags_model)**2/a**2)) # FOR ZERO NUGGET

        else:
            print('wrong model type chosen')

        if bounds == True:
            """Bounds and start values for curve fit"""
            if model == 'power':

                if zero_nugget == False:
                    p0 = [2.0,np.min(pics_model),np.max(pics_model)]
                    bounds = (0, [2.0, inf, inf])
                else:
                    p0 = [2.0,np.max(pics_model)]
                    bounds = (0, [2.0, inf])
            elif np.logical_or(model=='nested_sph_gau',model=='nested_sph_exp'):
                p0 = ...
                    [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_model
                bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                    np.max(pics_model)])
            elif model=='nested_exp_gau':
                if zero_nugget == False:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                        np.max(pics_model)])
                else:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.max(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], np.max(pics_model), ...
                        np.max(pics_model)])
```

```
            elif model=='nested_hole_gau':
                p0 = ...
                    [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_model
                bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                    np.max(pics_model)])
            elif model=='hole_damp':
                p0 = ...
                    [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_model
                bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                    10*np.max(lags_model)])
            elif model == 'nested_sph_exp_gau':
                if zero_nugget == False:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                        np.max(pics_model),np.max(pics_model)])
                else:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], np.max(pics_model), ...
                        np.max(pics_model),np.max(pics_model)])
            elif model == 'dub_spherical':
                if zero_nugget == False:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], inf, np.max(pics_model), ...
                        np.max(pics_model),lags_model[-1]])
                else:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], np.max(pics_model), ...
                        np.max(pics_model),lags_model[-1]])
            else:
                if zero_nugget == False:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.min(pics_model),np.max(pics_m
                    bounds = (0, [lags_model[-1], inf, np.max(pics_model)])
                else:
                    p0 = ...
                        [np.mean(lags_model[-int(len(lags_model)/4.0)]),np.max(pics_model)]
                    bounds = (0, [lags_model[-1], np.max(pics_model)])


            popt, pcov = curve_fit(semivar_return, lags_model, pics_model, ...
                bounds=bounds, p0 = p0)
        else:
            popt, pcov = curve_fit(semivar_return, lags_model, pics_model, ...
                method='lm')

        """Calculate or define nugget"""
        if zero_nugget == False:
            C0 = popt[1]
            C1 = popt[2]
            C2 = None
            C3 = None
            if model=='nested_sph_gau':
                C2 = popt[3]
            elif model=='nested_sph_exp':
                C2 = popt[3]
            elif model=='nested_exp_gau':
                C2 = popt[3]
            elif model=='nested_hole_gau':
                C2 = popt[3]
            elif model=='hole_damp':
                C2 = popt[3]
            elif model == 'nested_sph_exp_gau':
                C2 = popt[3]
                C3 = popt[4]
            elif model == 'dub_spherical':
                C2 = popt[3]
                C3 = popt[4]
        else:
            C0 = 0.0 # FOR ZERO NUGGET
```

```
            C1 = popt[1] # FOR ZERO NUGGET
            C2 = None
            C3 = None
            if np.logical_or(model=='nested_sph_gau',model=='nested_sph_exp'):
                C2 = popt[2]
            elif model=='nested_exp_gau':
                C2 = popt[2]
            elif model=='nested_hole_gau':
                C2 = popt[2]
            elif model=='hole_damp':
                C2 = popt[2]
            elif model == 'nested_sph_exp_gau':
                C2 = popt[2] # FOR ZERO NUGGET
                C3 = popt[3] # FOR ZERO NUGGET
            elif model == 'dub_spherical':
                C2 = popt[2] # FOR ZERO NUGGET
                C3 = popt[3] # FOR ZERO NUGGET
        """Calculate or define correlation length"""
        a = popt[0]
    else:
        a = set_model["a"]
        C0 = set_model["C0"]
        C1 = set_model["C1"]
        C2 = set_model["C2"]
        C3 = set_model["C3"]

    """Spherical model prediction"""
    lags_sv_curve = np.arange(0,int(np.round(lags[-1])))

    if model=='nested_sph_gau':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, mode = ...
            model)
    elif model=='nested_sph_exp':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, mode = ...
            model)
    elif model=='nested_exp_gau':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, mode = ...
            model)
    elif model=='nested_hole_gau':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, mode = ...
            model)
    elif model=='hole_damp':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, mode = ...
            model)
    elif model == 'nested_sph_exp_gau':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, C3 = C3, ...
            mode = model)
    elif model == 'dub_spherical':
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, C2 = C2, C3 = C3, ...
            mode = model)
    else:
        sv_curve = semivariogram_model(lags_sv_curve, a, C0, C1, mode = model)

    print('Semi-variogram model determined, starting LUT computation')
    print("")
    if nolut == False:
        if sph_d_all is None:
            """Semi-variogram LUT"""
            if np.logical_or(model=='nested_sph_gau',model=='nested_sph_exp'):
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, model = model)
            elif model=='nested_exp_gau':
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, model = model)
            elif model=='nested_hole_gau':
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, model = model)
            elif model=='hole_damp':
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, model = model)
            elif model == 'nested_sph_exp_gau':
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, C3 = C3, model = model)
```

```
            elif model == 'dub_spherical':
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, C2 = C2, C3 = C3, model = model)
            else:
                sv_lut = varioLUT(grid['grid spherical distances'], setup['N'], a, ...
                    C0, C1, model = model)
        else:
            if np.logical_or(model=='nested_sph_gau',model=='nested_sph_exp'):
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, model ...
                    = model)
            elif model=='nested_exp_gau':
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, model ...
                    = model)
            elif model=='nested_hole_gau':
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, model ...
                    = model)
            elif model=='hole_damp':
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, model ...
                    = model)
            elif model == 'nested_sph_exp_gau':
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, C3 = ...
                    C3, model = model)
            elif model == 'dub_spherical':
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, C2 = C2, C3 = ...
                    C3, model = model)
            else:
                sv_lut = varioLUT(sph_d_all, data['N'], a, C0, C1, model = model)

    if nolut == False:
        sdssim_semivar = {"semi-variogram LUT":sv_lut, "total data lags":lags, ...
            "total data sv":pics, "model data lags":lags_model, "model data ...
            sv":pics_model, "model names":model_name, "sv model y":sv_curve, "sv ...
            model x":lags_sv_curve, "sv model":model, "a":a, "C0":C0, "C1":C1, ...
            "C2":C2, "C3":C3, "n_lags":n_lags, "max_cloud":max_cloud, ...
            "sph_d_sorted":sph_d_sorted, "sort_d":sort_d}
    else:
        sdssim_semivar = {"total data lags":lags, "total data sv":pics, "model ...
            data lags":lags_model, "model data sv":pics_model, "model ...
            names":model_name, "sv model y":sv_curve, "sv model x":lags_sv_curve, ...
            "sv model":model, "a":a, "C0":C0, "C1":C1, "C2":C2, "C3":C3, ...
            "n_lags":n_lags, "max_cloud":max_cloud}

    return sdssim_semivar
```

E.5 SDSSIM_condtab

```
"""
Conditional distribution table
"""

def SDSSIM_condtab(data, setup, *args, normsize = 20, table = 'rough', load = False):
    """
    Setup for DSSIM
    """
    import numpy as np
    from scipy.stats import norm
    #import scipy.stats as st
    from SDSSIM_utility import printProgressBar

    """Target statistics"""
    target_var_dat = np.var(data["data"])
    ##target_var = args[0]
    target_var = np.var(data["data"])
    target_mean_dat = np.mean(data["data"])
    target_mean = 0.0

    """Linearly spaced value array with start/end very close to zero/one"""
    if setup["condtab_compiler"] == 'Fortran':
```

```
        start = 1e-26
    else:
        start = 1e-16 #Python min
        #start = 0.01 #Python min


    linspace = np.linspace(start,1-start,normsize)

    """target histogram cdf/ccdf"""
    data_sorted = np.sort(data["data"])

    if table == 'fine':
        rangn = np.linspace(-3.5,3.5,505)
        rangv = np.linspace(start,1.0,505)
    else:
        rangn = np.linspace(-3.5,3.5,101)
        rangv = np.linspace(start,2.0,101)
        #rangv = np.geomspace(1e-16,2.0,101)

    """Normscored local conditional distributions"""
    CQF_dist = np.zeros((len(rangn),len(rangv),len(linspace)))
    CQF_mean = np.zeros((len(rangn),len(rangv)))
    CQF_var = np.zeros((len(rangn),len(rangv)))


    if setup["condtab_compiler"] == 'Python':
        from sklearn.preprocessing import QuantileTransformer
        quantiles = 1000000

        """QuantileTransformer setup"""
        qt = QuantileTransformer(n_quantiles=quantiles, random_state=None, ...
            output_distribution='normal',subsample=1e8)
        qt.fit(data_sorted.reshape(-1,1))

        vrg = qt.transform(data_sorted.reshape(-1,1))

        print("")
        for i in range(0,len(rangn)):
            for j in range(0,len(rangv)):
                CQF_dist[i,j,:] = ...
                    np.sort(qt.inverse_transform((norm.ppf(linspace,loc=rangn[i],scale=np.sqrt(rangv[j]))
                CQF_mean[i,j] = np.mean(CQF_dist[i,j,:],axis=0,dtype=np.float64)
                CQF_var[i,j] = np.var(CQF_dist[i,j,:],axis=0,ddof=1,dtype=np.float64)


                #CQF_var[i,j] = np.var(CQF_dist[i,j,:],axis=0,dtype=np.float64)
            printProgressBar (i, len(rangn), subject = 'Python conditional ...
                probability table progress')

    elif setup["condtab_compiler"] == 'Fortran':

        if load == True:
            from SDSSIM_utility import variable_load
            CQF_dist = variable_load('CQF_dist_Julien.npy')
            CQF_mean = variable_load('CQF_mean_Julien.npy')
            CQF_var = variable_load('CQF_var_Julien.npy')
        else:
            import nscoresig
            import backtrsig

            tmin = target_mean - 10.0*np.sqrt(target_var)
            tmax = target_mean + 10.0*np.sqrt(target_var)
            iwt = 0
            wt = np.ones(len(data_sorted))
            tmp = np.zeros(len(data_sorted))
            lout = 0
            ierror = np.zeros(len(data_sorted))
            disc = 0
            vrg, ierror = nscoresig.nscore(data_sorted,tmin,tmax,iwt,wt,tmp,lout,disc)

            zmin = target_mean - 10.0*np.sqrt(target_var)
            zmax = target_mean + 10.0*np.sqrt(target_var)
            ltail = 0
```

```
            utail = 0
            utpar = 1.0
            ltpar = 1.0
            discrete = disc

            print("")
            for i in range(0,len(rangn)):
                for j in range(0,len(rangv)):
                    rn_dist = norm.ppf(linspace,loc=rangn[i],scale=np.sqrt(rangv[j]))
                    CQF_dist[i,j,:] = ...
                        np.array([backtrsig.backtr(vrgs,data_sorted,vrg,zmin,zmax,ltail,ltpar,utail,utpa
                        for vrgs in rn_dist])
                    CQF_mean[i,j] = np.mean(CQF_dist[i,j,:],axis=0,dtype=np.float64)
                    CQF_var[i,j] = ...
                        np.var(CQF_dist[i,j,:],axis=0,ddof=1,dtype=np.float64)
                    if CQF_var[i,j] < 0.0:
                        CQF_var[i,j] = 0.0

                printProgressBar(i, len(rangn), subject = 'Fortran conditional ...
                    probability table progress')
    else:
        print('Error: compiler must be Python or Fortran')

    sdssim_condtab = {"target variance":target_var, "target ...
        variance_dat":target_var_dat, "target mean":target_mean, "target ...
        mean_dat":target_mean_dat, "QF norm range":rangn, "QF var range":rangv, ...
        "CQF dist":CQF_dist, "CQF mean":CQF_mean, "CQF var":CQF_var, "target ...
        normscore":vrg, "compiler":setup["condtab_compiler"], "normsize":normsize, ...
        "start":start}
    return sdssim_condtab
```

E.6 SDSSIM_greens

```
"""
Greens functions and data implementation
"""

def Gr(r_s, r_d, lat_s, lat_d, lon_s, lon_d, angular_distance = False, angdist = 0):
    import numpy as np

    theta_s, theta_d, lon_s, lon_d, angdist = map(np.radians, [90.0-lat_s, ...
        90.0-lat_d, lon_s, lon_d, angdist])

    h = r_s/r_d

    if angular_distance == True:
        mu = np.cos(angdist)
    else:
        mu = ...
            np.cos(theta_d)*np.cos(theta_s)+np.sin(theta_d)*np.sin(theta_s)*np.cos(lon_d-lon_s)

    R = np.sqrt(r_d**2+r_s**2-2*r_d*r_s*mu)
    f = R/r_d

    G_r = 1/(4*np.pi)*h**2*(1-h**2)/f**3
    G_r = np.matrix(G_r)
    return G_r

def Gr_vec(r_s, r_d, lat_s, lat_d, lon_s, lon_d, angdist_out = False):
    import numpy as np

    theta_s, theta_d, lon_s, lon_d = map(np.radians, [np.matrix(90.0-lat_s), ...
        np.matrix(90.0-lat_d), np.matrix(lon_s), np.matrix(lon_d)])

    r_s = np.matrix(r_s)
    r_d = np.matrix(r_d)

    mu = np.cos(theta_d.T)*np.cos(theta_s)+np.multiply(np.sin(theta_d.T)
```

```
    *np.sin(theta_s),np.cos(lon_d.T-lon_s))

    h = r_s.T/r_d

    def rs(r_s,r_d, mu):
        r_d_sq = np.power(r_d,2)
        r_s_sq = np.power(r_s,2)
        rr_ds = r_d.T*r_s
        rr_ds_mu = 2*np.multiply(rr_ds,mu)
        rr_ds_sq_sum = r_d_sq.T+r_s_sq
        R = np.sqrt(rr_ds_sq_sum-rr_ds_mu)
        f = R.T/r_d
        return f

    f = rs(r_s,r_d, mu)

    h_sq = np.power(h,2)

    f_cb = np.power(f,3)

    G_r = (1/(4*np.pi)*np.multiply(h_sq,(1-h_sq))/f_cb).T
    if angdist_out == True:
        return G_r, mu
    else:
        return G_r

def Gr_vec2(r_s, r_d, lat_s, lat_d, lon_s, lon_d):
    import numpy as np

    lat_mesh_s, lat_mesh_d = np.meshgrid(lat_s, lat_d)
    lon_mesh_s, lon_mesh_d = np.meshgrid(lon_s, lon_d)

    theta_s, theta_d, lon_s, lon_d = map(np.radians, [np.matrix(90.0-lat_mesh_s), ...
        np.matrix(90.0-lat_mesh_d), np.matrix(lon_mesh_s), np.matrix(lon_mesh_d)])


    r_s = np.matrix(r_s)
    r_d = np.matrix(r_d)

    mu = ...
        np.multiply(np.cos(theta_d),np.cos(theta_s))+np.multiply(np.multiply(np.sin(theta_d),np.sin(thet

    h = r_s.T/r_d

    r_d_sq = np.power(r_d,2)
    r_s_sq = np.power(r_s,2)
    rr_ds = r_d.T*r_s
    rr_ds_mu = 2*np.multiply(rr_ds,mu)
    rr_ds_sq_sum = r_d_sq.T+r_s_sq

    R = np.sqrt(rr_ds_sq_sum-rr_ds_mu)

    f = R.T/r_d

    h_sq = np.power(h,2)

    f_cb = np.power(f,3)

    G_r = (1/(4*np.pi)*np.multiply(h_sq,(1-h_sq))/f_cb).T
    return G_r

def SDSSIM_greens(data, data_prior, setup_data,  setup_prior, condtab, semivar, ...
    semivar_prior, grid, errorvar = 1.0, G_d_only = False, all_weights = True, DN ...
    = None, SN = 30, kriging_method = "ordinary", global_coverage = False, ...
    coverage_type = "uniform"):
    import numpy as np

    """ DETERMINE DIFFERENTIAL APPROXIMATIONS """
    Glut_gk = None
    K_data_weights = None
    K_data_weights_OK = None
    lagrange = None
    G_k = None
```

```
    GG_K = None
    lsq_rank = None
    idx_data_support = np.empty([0,],dtype=int)

    def greens_differentials(grid):
        s_cap = grid["s_cap"].T
        s_cap_diff = np.diff(s_cap,axis=0)
        s_cap_diff = np.vstack((s_cap[0],s_cap_diff))

        n_regions = grid["n_regions"].T

        d_theta_core = np.empty([0,1],dtype=float)
        d_phi_core = np.empty([0,1],dtype=float)

        for i in range(0,len(n_regions)):

            d_theta_core = ...
                np.vstack((d_theta_core,(s_cap_diff[i]*np.ones(int(n_regions[i]))).T))

            d_phi_core = ...
                np.vstack((d_phi_core,(2*np.pi/n_regions[i]*np.ones(int(n_regions[i]))).T))

        theta_core = np.matrix(90.0-grid["grid latitude"])*np.pi/180.0

        return np.multiply(np.multiply(d_theta_core,d_phi_core),np.sin(theta_core.T))

    dd_theta_phi_core = greens_differentials(grid)

    """ KRIGING SETUP """

    G_d = np.multiply(dd_theta_phi_core.T,Gr_vec(data_prior["data ...
        radius"]*np.ones(data_prior["N"]), data["data radius"], data_prior["data ...
        latitude"], data["data latitude"], data_prior["data longitude"], ...
        data["data longitude"]))

    if G_d_only == False:
        G_k = G_d*(condtab["target variance"]-semivar_prior["semi-variogram LUT"])

        GG_K = G_k*G_d.T  + np.diag(errorvar*np.ones(setup_data["N"],)) # BEST

        if all_weights == True:
            if kriging_method == "simple":
                lsq_sol = np.linalg.lstsq(GG_K, G_k, rcond=None)
                K_data_weights = lsq_sol[0]
                lsq_rank =  lsq_sol[2]

            elif kriging_method == "ordinary":
                G_k_OK = np.vstack((G_k,1.0*np.ones(data_prior["N"])))

                lagrange_vert_dat = 1.0*np.ones((data["N"],1))
                lagrange_horz_dat = np.vstack((lagrange_vert_dat,0.0)).T

                GG_K_OK = np.append(GG_K, lagrange_vert_dat, axis=1)
                GG_K_OK = np.append(GG_K_OK, lagrange_horz_dat, axis=0)

                lsq_sol = np.linalg.lstsq(GG_K_OK, G_k_OK, rcond=None)
                K_data_weights_OK = lsq_sol[0]
                lsq_rank =  lsq_sol[2]

                lagrange = K_data_weights_OK[-1,:].T
                K_data_weights_OK = K_data_weights_OK[:-1,:]

            elif kriging_method == "ordinary_scaled":
                scale_const = data["N"]/(data["N"]+SN)
                G_k_OK = np.vstack((G_k,scale_const*np.ones(data_prior["N"])))

                lagrange_vert_dat = 1.0*np.ones((data["N"],1))
                lagrange_horz_dat = np.vstack((lagrange_vert_dat,0.0)).T

                GG_K_OK = np.append(GG_K, lagrange_vert_dat, axis=1)
                GG_K_OK = np.append(GG_K_OK, lagrange_horz_dat, axis=0)

                lsq_sol = np.linalg.lstsq(GG_K_OK, G_k_OK, rcond=None)
```

```
                K_data_weights_OK = lsq_sol[0]
                lsq_rank =  lsq_sol[2]

                lagrange = K_data_weights_OK[-1,:].T
                K_data_weights_OK = K_data_weights_OK[:-1,:]

            elif kriging_method == "ordinary_half":
                scale_const = 0.5
                G_k_OK = np.vstack((G_k,scale_const*np.ones(data_prior["N"])))

                lagrange_vert_dat = 1.0*np.ones((data["N"],1))
                lagrange_horz_dat = np.vstack((lagrange_vert_dat,0.0)).T

                GG_K_OK = np.append(GG_K, lagrange_vert_dat, axis=1)
                GG_K_OK = np.append(GG_K_OK, lagrange_horz_dat, axis=0)

                lsq_sol = np.linalg.lstsq(GG_K_OK, G_k_OK, rcond=None)
                K_data_weights_OK = lsq_sol[0]
                lsq_rank =  lsq_sol[2]

                lagrange = K_data_weights_OK[-1,:].T
                K_data_weights_OK = K_data_weights_OK[:-1,:]

        if DN is not None:

            def take_along_axis(arr, ind, axis):
                """
                ... here means a "pack" of dimensions, possibly empty

                arr: array_like of shape (A..., M, B...)
                    source array
                ind: array_like of shape (A..., K..., B...)
                    indices to take along each 1d slice of `arr`
                axis: int
                    index of the axis with dimension M

                out: array_like of shape (A..., K..., B...)
                    out[a..., k..., b...] = arr[a..., inds[a..., k..., b...], b...]
                """
                if axis < 0:
                   if axis ≥ -arr.ndim:
                       axis += arr.ndim
                   else:
                       raise IndexError('axis out of range')
                ind_shape = (1,) * ind.ndim
                ins_ndim = ind.ndim - (arr.ndim - 1)    #inserted dimensions

                dest_dims = list(range(axis)) + [None] + list(range(axis+ins_ndim, ...
                    ind.ndim))

                # could also call np.ix_ here with some dummy arguments, then ...
                    throw those results away
                inds = []
                for dim, n in zip(dest_dims, arr.shape):
                    if dim is None:
                        inds.append(ind)
                    else:
                        ind_shape_dim = ind_shape[:dim] + (-1,) + ind_shape[dim+1:]
                        inds.append(np.arange(n).reshape(ind_shape_dim))

                return arr[tuple(inds)]

            if global_coverage == True:
                if coverage_type == "uniform":
                    idx_data_support = np.random.randint(0, data["N"], size = ...
                        (DN,data_prior["N"]))

                elif coverage_type == "geometric":
                    sort_support = np.flipud(np.argsort(G_d,axis=0))
                    N_close = int(DN/2)
                    idx_close_support = sort_support[:N_close,:]
                    geom_prob = np.geomspace(1,0.1,data["N"]-N_close)
                    geom_prob = geom_prob/np.sum(geom_prob)
```

```
                    idx_far_support = ...
                        np.matrix([np.random.choice(np.ravel(sort_support[N_close:,x]),size=(int(DN/2),)
                        for x in range(0,data_prior["N"])]).T
                    idx_data_support = np.vstack((idx_close_support,idx_far_support))
                else:
                    sort_support = np.flipud(np.argsort(G_d,axis=0))
                    N_close = int(DN/2)
                    idx_close_support = sort_support[:N_close,:]
                    idx_far_support = ...
                        sort_support[N_close:,:][np.random.randint(0, ...
                        int(sort_support.shape[0]-N_close), size = int(DN/2)), :]
                    idx_data_support = np.vstack((idx_close_support,idx_far_support))

            else:
                idx_data_support = np.flipud(np.argsort(G_d,axis=0))[:DN,:]

            Glut_gk = take_along_axis(G_k, idx_data_support, axis=0).T

            if all_weights is not True:
                G_k = None

    greens = {"G_k":G_k, "G_d": G_d, "GG_K":GG_K, "K_data_weights":K_data_weights, ...
        "K_data_weights_OK":K_data_weights_OK, "weights_rank_OK":lsq_rank, ...
        "lagrange":lagrange, "dd_theta_phi_core":dd_theta_phi_core, ...
        "errorvar":errorvar, "G_k_DN":Glut_gk, "G_idx_DN":idx_data_support.T, ...
        "kriging_method":kriging_method}

    return greens
```

E.7 SDSSIM_sdssim

```
"""
SDSSIM
"""

def cond_lookup(data_min, data_max, target_var, CQF_mean, CQF_var, kriging_mean, ...
    kriging_var, shape):
    import numpy as np

    dm = data_max - data_min
    dv = target_var

    #dist = ((CQF_mean-kriging_mean)/dm)**2.0 + ((CQF_var-kriging_var)/dv)**2.0
    #dist = (CQF_mean-kriging_mean) + (CQF_var-kriging_var)


    dist = np.power((CQF_mean-kriging_mean)/dm,2) + ...
        np.power((CQF_var-kriging_var)/dv,2)
    #dist = ((CQF_mean-kriging_mean)/dm)**2.0 + abs(CQF_var-kriging_var)/np.sqrt(dv)

    #dist = ((CQF_mean-kriging_mean)/dm)**2.0 + abs(CQF_var-kriging_var)/dv
    #dist = ((abs(CQF_mean-kriging_mean))/dm) + (abs(CQF_var-kriging_var)/dv)
    inv = np.unravel_index(np.argmin(dist),shape)

    im_sel = inv[0]
    iv_sel = inv[-1]

    return im_sel, iv_sel

def oz_correction(idx_n, idx_v, Zf, Zk, kriging_var, CQF_mean, CQF_var, ...
    CQF_var_max, on_off = 'on'):
    import numpy as np

    if on_off == 'on':
        Zf_mean = CQF_mean[idx_n,idx_v]
        Zf_std = np.sqrt(CQF_var[idx_n,idx_v],dtype=np.float64)

        if 0.0 ≥ Zf_std:
```

```
            Zf_std = np.sqrt(CQF_var_max)

        Z = (Zf - Zf_mean)*np.sqrt(kriging_var)/Zf_std+Zk
    else:
        Z = Zf
    return Z

def sdssim(N, SN, DN, vario_lut, idx_rnd, condtab, prior, grid, data_support, ...
    greens, shape, run, ozcorr = 'off', sort_method = "cut-off", neighborhood = ...
    "data_limited", kriging_method = "simple", threshold_factor = 1.0, mean_CQF = ...
    False, mean_burn_in = False, N_burn_in = 10):
    import numpy as np
    #import cupy as cp
    import scipy as sp

    from SDSSIM_utility import printProgressBar
    # create array for the output and initializations
    M = np.zeros((N))
    walked_in_reach = 0
    var_lz = 0
    idx_v = np.empty([0,],dtype=int)
    idx_n = np.empty([0,],dtype=int)
    data_min = np.min(prior["data"])
    data_max = np.max(prior["data"])

    locations_walked = np.empty([0,],dtype=int)

    CQF_var_max = np.max(condtab["CQF var"][condtab["CQF var"]>0.0])
    CQF_dist_len = len(condtab["CQF dist"][0,0,:])

    vario_max = vario_lut[0,-1]

    len_walked = 0
    N_no_sim = 0
    save_weights = list()
    save_weights_rel_dat = list()
    save_lagrange = list()
    save_kriging_mv = list()
    save_idx_nv = list()
    save_invshape = list()
    save_lstsq = list()
    save_Zi = list()

    # Start random walk
    for step in idx_rnd:
        idx = step

        K_ss = np.empty([0,],dtype=float)
        K_dd = np.empty([0,],dtype=float)
        K_ds = np.empty([0,],dtype=float)
        K_sys = np.empty([0,],dtype=float)

        k_ss = np.empty([0,],dtype=float)
        k_dd = np.empty([0,],dtype=float)
        k_sys = np.empty([0,],dtype=float)

        Zk = np.empty([0,],dtype=float)
        kriging_var = np.empty([0,],dtype=float)
        idx_n = np.empty([0,],dtype=int)
        idx_v = np.empty([0,],dtype=int)
        Zf = np.empty([0,],dtype=float)

        idx_data_support_SN = np.empty([0,],dtype=int)

        vario_near = np.empty([0,],dtype=float)
        idx_vario_sort = np.empty([0,],dtype=int)
        idx_SN = np.empty([0,],dtype=int)
        vario_SN = np.empty([0,],dtype=float)
        kriging_weights = np.empty([0,],dtype=float)
        Zi = np.empty([0,],dtype=float)

        lagrange = 0.0
        lagrange_vert_sim = np.empty([0,],dtype=float)
```

```
        lagrange_horz_sim = np.empty([0,],dtype=float)

        lstsq_sol = np.empty([0,],dtype=float)

        walked_in_reach += 1


        if sort_method == "cut-off":
            vario_near = vario_lut[idx,locations_walked]
            idx_vario_sort = vario_near.argsort()
            idx_SN = locations_walked[idx_vario_sort][:SN]
            vario_SN = vario_near[idx_vario_sort][:SN]

        elif sort_method == "threshold":

            vario_near = vario_lut[idx,locations_walked]
            idx_vario_sort = vario_near.argsort()
            idx_SN = locations_walked[idx_vario_sort][:SN]
            vario_SN = vario_near[idx_vario_sort][:SN]

            idx_vario_thresh = np.array(np.where(vario_SN < ...
                threshold_factor*vario_max)).ravel()
            idx_SN = idx_SN[idx_vario_thresh]
            vario_SN = vario_SN[idx_vario_thresh]

        else:
            print("no correct sort method chosen")
            return

        """""""""NEIGHBORHOOD SETUP""""""""
        if neighborhood == "stochastic":
            """STOCHASTIC SIMULATION (NO DATA)"""
           # Generate samples when no other value is in reach
            if np.size(idx_SN)==0:
                Zk = prior["data"][np.random.randint(0,N)] # Random draw

                kriging_var = condtab["target variance"] # Set kriging variance
                #kriging_var = vario_max

            # Generate sample when only one value is in reach
            elif np.size(idx_SN)==1:
                # Pull the other simulated value
                Zi = M[idx_SN]

                # Find kriging weight
                kriging_weights = (condtab["target variance"] - ...
                    vario_SN)/condtab["target variance"]

                # Compute kriging mean
                Zk = np.float(np.array(kriging_weights*(Zi - condtab["target ...
                    mean"]) + condtab["target mean"]))

                # Compute kriging variance
                kriging_var = np.float(np.array(vario_SN)) # target_var - ...
                    (target_var - vario_near)

            # Simple kriging when more than one value are in reach
            else:

                # Find nearest location simulation values
                Zi = np.matrix(M[idx_SN]).T

                # Set up k
                k_sys = condtab["target variance"] - np.matrix(vario_SN).T

                # Lookup all closest location semi-variances to each other ...
                    (efficiently)
                K_sys = condtab["target variance"] - (vario_lut.ravel()[(idx_SN + ...
                    (idx_SN * ...
                    vario_lut.shape[1]).reshape((-1,1))).ravel()]).reshape(idx_SN.size, ...
                    idx_SN.size)

        elif neighborhood == "no_cross":
```

```
            """SN SIM + ALL DATA"""

            if len(idx_SN) < 1:
                if kriging_method == "simple":
                    kriging_weights = greens["K_data_weights"][:,idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                elif kriging_method == "ordinary":
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    lagrange = greens["lagrange"][idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                else:
                    print("no correct kriging method + neighborhood combination ...
                        chosen")
                    return
            else:
                # Set up all data and find nearest simulation values
                Zi = np.matrix(data_support["data"]).T
                Zi = np.vstack((Zi,np.matrix(M[idx_SN]).T))
                # Set up k
                k_ss = condtab["target variance"] - np.matrix(vario_SN).T

                # Lookup all closest location semi-variances to each other ...
                    (efficiently)
                K_ss = condtab["target variance"] - (vario_lut.ravel()[(idx_SN + ...
                    (idx_SN * ...
                    vario_lut.shape[1]).reshape((-1,1))).ravel()]).reshape(idx_SN.size, ...
                    idx_SN.size)

                if kriging_method == "simple":

                    kriging_weights_SSK = np.linalg.lstsq(K_ss, k_ss, rcond=None)[0]
                    k_dd = greens["G_k"][:, idx]
                    k_sys = np.vstack((k_dd,k_ss))
                    kriging_weights = greens["K_data_weights"][:,idx]
                    kriging_weights = np.vstack((kriging_weights,kriging_weights_SSK))

                    kriging_var = condtab["target variance"] - ...
                        np.float(kriging_weights.T*k_sys)
                    kriging_var = np.float(kriging_var)
                    Zk = np.float(np.array(kriging_weights.T*(Zi - condtab["target ...
                        mean"]) + condtab["target mean"]))

                elif kriging_method == "ordinary":

                    lagrange_vert_sim = np.ones((len(K_ss),1))
                    lagrange_horz_sim = np.vstack((lagrange_vert_sim,0.0)).T
                    K_ss = np.append(K_ss, lagrange_vert_sim, axis=1)
                    K_ss = np.append(K_ss, lagrange_horz_sim, axis=0)


                    if greens["kriging_method"] == "ordinary":
                        k_ss = np.vstack((k_ss,1.0))
                    elif greens["kriging_method"] == "ordinary_scaled":
                        scale_const = SN/(data_support["N"]+SN)
                        k_ss = np.vstack((k_ss,scale_const))
                    elif greens["kriging_method"] == "ordinary_half":
                        k_ss = np.vstack((k_ss,0.5))

                    kriging_weights_SOK = np.linalg.lstsq(K_ss, k_ss, rcond=None)[0]

                    #lagrange = greens["lagrange"][idx] + kriging_weights_SOK[-1]
                    lagrange = greens["lagrange"][idx]

                    kriging_weights_SOK = kriging_weights_SOK[:-1]
                    k_ss = k_ss[:-1]

                    k_dd = greens["G_k"][:, idx]
                    k_sys = np.vstack((k_dd,k_ss))
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    kriging_weights = np.vstack((kriging_weights,kriging_weights_SOK))

```

```
                    kriging_var = condtab["target variance"] - ...
                        np.float(kriging_weights.T*k_sys) - lagrange
                    kriging_var = np.float(kriging_var)
                    Zk = np.float(np.array(kriging_weights.T*Zi))

                else:
                    print("only simple kriging available for no_cross neighborhood")
                    return

        elif neighborhood == "mean_sim":
            """SN SIM + ALL DATA"""

            if len(idx_SN) < 1:
                if kriging_method == "simple":
                    kriging_weights = greens["K_data_weights"][:,idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                elif kriging_method == "ordinary":
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    lagrange = greens["lagrange"][idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                else:
                    print("no correct kriging method + neighborhood combination ...
                        chosen")
                    return
            else:
                # Set up all data and find nearest simulation values
                Zi = np.matrix(data_support["data"]).T
                Zi_S = np.matrix(M[idx_SN]).T
                # Set up k
                k_ss = condtab["target variance"] - np.matrix(vario_SN).T

                # Lookup all closest location semi-variances to each other ...
                    (efficiently)
                K_ss = condtab["target variance"] - (vario_lut.ravel()[(idx_SN + ...
                    (idx_SN * ...
                    vario_lut.shape[1]).reshape((-1,1))).ravel()]).reshape(idx_SN.size, ...
                    idx_SN.size)

                if kriging_method == "simple":

                    kriging_weights_SSK = np.linalg.lstsq(K_ss, k_ss, rcond=None)[0]
                    k_dd = greens["G_k"][:, idx]
                    k_sys = np.vstack((k_dd,k_ss))
                    kriging_weights = greens["K_data_weights"][:,idx]
                    kriging_weights = np.vstack((kriging_weights,kriging_weights_SSK))

                    kriging_var = condtab["target variance"] - ...
                        np.float(kriging_weights.T*k_sys)
                    kriging_var = np.float(kriging_var)
                    Zk = np.float(np.array(kriging_weights.T*(Zi - condtab["target ...
                        mean"]) + condtab["target mean"]))

                elif kriging_method == "ordinary":

                    lagrange_vert_sim = np.ones((len(K_ss),1))
                    lagrange_horz_sim = np.vstack((lagrange_vert_sim,0.0)).T
                    K_ss = np.append(K_ss, lagrange_vert_sim, axis=1)
                    K_ss = np.append(K_ss, lagrange_horz_sim, axis=0)


                    if greens["kriging_method"] == "ordinary":
                        k_ss = np.vstack((k_ss,1.0))
                    elif greens["kriging_method"] == "ordinary_scaled":
                        scale_const = SN/(data_support["N"]+SN)
                        k_ss = np.vstack((k_ss,scale_const))
                    elif greens["kriging_method"] == "ordinary_half":
                        k_ss = np.vstack((k_ss,0.5))

                    kriging_weights_SOK = np.linalg.lstsq(K_ss, k_ss, rcond=None)[0]

                    lagrange_SOK = kriging_weights_SOK[-1]
```

```
                    lagrange = greens["lagrange"][idx]

                    kriging_weights_SOK = kriging_weights_SOK[:-1]
                    k_ss = k_ss[:-1]

                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    k_dd = greens["G_k"][:, idx]

                    kriging_var = np.float(condtab["target variance"] - ...
                        np.float(kriging_weights.T*k_dd) - lagrange)
                    kriging_var_SOK = np.float(condtab["target variance"] - ...
                        np.float(kriging_weights_SOK.T*k_ss) - lagrange_SOK)

                    kriging_var = (kriging_var + kriging_var_SOK)/2.0

                    Zk = np.float(np.array(kriging_weights.T*Zi))
                    Zk_SOK = np.float(np.array(kriging_weights_SOK.T*Zi_S))
                    Zk = (Zk + Zk_SOK)/2.0
                else:
                    print("only simple kriging available for no_cross neighborhood")
                    return

        elif neighborhood == "data_limited":
            """SN SIM + DATA IN SIM SPACE + DATA/SIM"""

            if np.logical_and.reduce((len(idx_SN) < 1, ...
                np.logical_or(greens["K_data_weights_OK"] is not None, ...
                greens["K_data_weights"] is not None))):
                if kriging_method == "simple":
                    kriging_weights = greens["K_data_weights"][:,idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                elif kriging_method == "ordinary":
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    lagrange = greens["lagrange"][idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                else:
                    print("no correct kriging method + neighborhood combination ...
                        chosen")
                    return
                N_no_sim += 1

            elif np.logical_and.reduce((mean_burn_in is True, ...
                np.logical_or(greens["K_data_weights_OK"] is not None, ...
                greens["K_data_weights"] is not None), len_walked ≤ N_burn_in)):
                if kriging_method == "simple":
                    kriging_weights = greens["K_data_weights"][:,idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                elif kriging_method == "ordinary":
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    lagrange = greens["lagrange"][idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                else:
                    print("no correct kriging method + neighborhood combination ...
                        chosen")
                    return
                N_no_sim += 1

            else:

                # Find nearest data
                idx_data_support_SN = greens["G_idx_DN"][idx,:]

                # Set up k
                k_ss = condtab["target variance"] - np.matrix(vario_SN).T

                k_dd = np.matrix(greens["G_k_DN"][idx,:]).T


```

```
                # Lookup all closest location semi-variances to each other ...
                    (efficiently)
                K_ss = condtab["target variance"] - (vario_lut.ravel()[(idx_SN + ...
                    (idx_SN * ...
                    vario_lut.shape[1]).reshape((-1,1))).ravel()]).reshape(idx_SN.size, ...
                    idx_SN.size)
                #K_ss = K_ss + np.diag(1**2*np.ones(K_ss.shape[0],))

                # Efficient lookup of Greens
                K_dd = (np.ravel(greens["GG_K"])[(idx_data_support_SN + ...
                    (idx_data_support_SN * ...
                    greens["GG_K"].shape[1]).reshape((-1,1))).ravel()]).reshape(idx_data_support_SN.size
                    idx_data_support_SN.size)

                K_ds = greens["G_k_DN"][idx_SN,:]

                k_sys = np.vstack((k_dd,k_ss))

                K_sys = np.zeros((len(K_dd)+len(K_ss),len(K_dd)+len(K_ss)))

                K_sys[:len(K_dd),:len(K_dd)] = K_dd
                if len(idx_SN) ≥ 1:
                    K_sys[-len(K_ss):,-len(K_ss):] = K_ss
                    K_sys[:len(K_dd),-len(K_ss):] = K_ds.T
                    K_sys[-len(K_ss):,:len(K_dd)] = K_ds

                #print('\n Current kriging size:', np.shape(K_sys), end = '\n')

                Zi = np.matrix(data_support["data"][idx_data_support_SN]).T
                Zi = np.vstack((Zi,np.matrix(M[idx_SN]).T))

        elif neighborhood == "data_all":
            """SN SIM + DATA IN SIM SPACE + DATA/SIM"""

            if len(idx_SN) < 1:
                if kriging_method == "simple":
                    kriging_weights = greens["K_data_weights"][:,idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                elif kriging_method == "ordinary":
                    kriging_weights = greens["K_data_weights_OK"][:,idx]
                    lagrange = greens["lagrange"][idx]
                    k_sys = greens["G_k"][:,idx]
                    Zi = np.matrix(data_support["data"]).T
                else:
                    print("no correct kriging method + neighborhood combination ...
                        chosen")
                    return
            else:

                k_dd = greens["G_k"][:,idx]
                K_dd = greens["GG_K"]
                K_ds = greens["G_k"][:,idx_SN]
                Zi = np.matrix(data_support["data"]).T

                # Set up k
                k_ss = condtab["target variance"] - np.matrix(vario_SN).T
                k_sys = np.vstack((k_dd,k_ss))

                # Lookup all closest location semi-variances to each other ...
                    (efficiently)
                K_ss = condtab["target variance"] - (vario_lut.ravel()[(idx_SN + ...
                    (idx_SN * ...
                    vario_lut.shape[1]).reshape((-1,1))).ravel()]).reshape(idx_SN.size, ...
                    idx_SN.size)

                K_sys = np.zeros((len(K_dd)+len(K_ss),len(K_dd)+len(K_ss)))
                K_sys[:len(K_dd),:len(K_dd)] = K_dd
                if len(idx_SN) ≥ 1:
                    K_sys[-len(K_ss):,-len(K_ss):] = K_ss
                    K_sys[:len(K_dd),-len(K_ss):] = K_ds
                    K_sys[-len(K_ss):,:len(K_dd)] = K_ds.T
```

```
                #print('\n Current kriging size:', np.shape(K_sys), end = '\n')


                Zi = np.vstack((Zi,np.matrix(M[idx_SN]).T))

        else:
            print("no correct neigborhood chosen")
            return

        # if the kriging variables haven't already been found (eg. stochastic with ...
            zero or one idx_near)
        # find them with chosen method
        if np.size(Zk) == 0:
            if kriging_method == "simple":
                """SIMPLE KRIGING (SK)"""
                if np.size(kriging_weights) == 0:
                    lstsq_sol = np.linalg.lstsq(K_sys, k_sys, rcond=None)
                    kriging_weights = lstsq_sol[0]
                    save_lstsq.append([lstsq_sol[2]])

                    #print(np.all(np.linalg.eigvals(K_sys) ≥ 0))

                    #L = np.linalg.cholesky(K_sys)
                    #y = sp.linalg.solve_triangular(L,k_sys, check_finite=False)
                    #kriging_weights = sp.linalg.solve_triangular(L.T,y, ...
                        check_finite=False)

                kriging_var = condtab["target variance"] - ...
                    np.float(kriging_weights.T*k_sys)
                kriging_var = np.float(kriging_var)
                Zk = np.float(np.array(kriging_weights.T*(Zi - condtab["target ...
                    mean"]) + condtab["target mean"]))

                #print(np.all(np.linalg.eigvals(K_sys) ≥ 0))

            elif kriging_method == "ordinary":
                """ORDINARY KRIGING (OK)"""
                # ORDINARY KRIGING CONDITIONS FOR K MATRIX
                if np.size(kriging_weights) == 0:
                    #print(np.all(np.linalg.eigvals(K_sys) > 0))
                    lagrange_vert_sim = np.ones((len(K_sys),1))
                    lagrange_horz_sim = np.vstack((lagrange_vert_sim,0.0)).T
                    K_sys = np.append(K_sys, lagrange_vert_sim, axis=1)
                    K_sys = np.append(K_sys, lagrange_horz_sim, axis=0)
                    #K_sys = K_sys + np.diag(np.ones(K_sys.shape,))
                    #K_sys[-1,-1] = 0.0

                    k_sys = np.vstack((k_sys,1.0))

                    #print(np.all(np.linalg.eigvals(K_sys) > 0))

                    # linear system solvers
                    cupy_test = False
                    solve_test = False
                    use_cholesky = False

                    if cupy_test == False:
                        if solve_test == False:

                            lstsq_sol = np.linalg.lstsq(K_sys, k_sys, rcond=None)
                            kriging_weights = lstsq_sol[0]
                            save_lstsq.append([lstsq_sol[2]])
                        else:
                            if np.logical_and(use_cholesky == True,len_walked>100):

                                L = np.linalg.cholesky(K_sys)
                                y = sp.linalg.solve_triangular(L,k_sys, ...
                                    check_finite=False)
                                kriging_weights = ...
                                    sp.linalg.solve_triangular(L.T,y, ...
                                    check_finite=False)
                            else:
```

```
                                kriging_weights = np.linalg.solve(K_sys, k_sys)
                        #else:
                            # CUPY
                            #K_sys_cp = cp.asarray(K_sys)
                            #k_sys_cp = cp.asarray(k_sys)
                            #kriging_weights = cp.linalg.solve(K_sys_cp, k_sys_cp)
                            #kriging_weights = cp.asnumpy(kriging_weights)

                        lagrange = kriging_weights[-1]
                        kriging_weights = kriging_weights[:-1]
                        k_sys = k_sys[:-1]

                    kriging_var = condtab["target variance"] - ...
                        np.float(kriging_weights.T*k_sys) - lagrange
                    kriging_var = np.float(kriging_var)
                    Zk = np.float(np.array(kriging_weights.T*Zi))

                else:
                    print("no correct kriging method + neighborhood combination chosen")
                    return



            # Ensure positive kriging_var, add error for run print
            if kriging_var < 0.0:
                print(kriging_var)
                kriging_var = condtab["target variance"]
                var_lz += 1

            # Smallest differences from kriging to transformed Gaussian distribution ...
                ranges
            ## TMH/OZ STYLE ##
            idx_n, idx_v = cond_lookup(data_min, data_max, condtab["target variance"], ...
                condtab["CQF mean"], condtab["CQF var"], Zk, kriging_var, shape)


            # Get the closest local distribution and draw
            if mean_CQF is True:
                Zf = condtab["CQF mean"][idx_n,idx_v]
            else:
                if mean_burn_in is True:
                    if len_walked < N_burn_in:
                        Zf = condtab["CQF mean"][idx_n,idx_v]
                    else:
                        Zf = condtab["CQF ...
                            dist"][idx_n,idx_v,np.random.randint(0,CQF_dist_len,size=1)]
                else:
                    Zf = condtab["CQF ...
                        dist"][idx_n,idx_v,np.random.randint(0,CQF_dist_len,size=1)]

            # Pull simulated value and distribution mean/variance
            M[idx] = oz_correction(idx_n, idx_v, Zf, Zk, kriging_var, condtab["CQF ...
                mean"], condtab["CQF var"], CQF_var_max, on_off = ozcorr)


            # Sample neighborhoods
            if len_walked == int(10):
                idx_show_data_support_tenth = idx_data_support_SN
                idx_show_sim_support_tenth = idx_SN
                idx_step_tenth = idx
                save_weights.append([kriging_weights])
                save_weights_rel_dat.append([np.multiply(kriging_weights,Zi)])
                save_Zi.append([np.ravel(Zi)])
            if len_walked == int(N/4):
                idx_show_data_support_quarter = idx_data_support_SN
                idx_show_sim_support_quarter = idx_SN
                idx_step_quarter = idx
                save_weights.append([kriging_weights])
                save_weights_rel_dat.append([np.multiply(kriging_weights,Zi)])
                save_Zi.append([np.ravel(Zi)])
            elif len_walked == int(N/2):
                idx_show_data_support_half = idx_data_support_SN
                idx_show_sim_support_half = idx_SN
```

```
            idx_step_half = idx
            save_weights.append([kriging_weights])
            save_weights_rel_dat.append([np.multiply(kriging_weights,Zi)])
            save_Zi.append([np.ravel(Zi)])
        elif len_walked == int(3*N/4):
            idx_show_data_support_tq = idx_data_support_SN
            idx_show_sim_support_tq = idx_SN
            idx_step_tq = idx
            save_weights.append([kriging_weights])
            save_weights_rel_dat.append([np.multiply(kriging_weights,Zi)])
            save_Zi.append([np.ravel(Zi)])
        elif len_walked == N-1:
            idx_show_data_support_final = idx_data_support_SN
            idx_show_sim_support_final = idx_SN
            idx_step_final = idx
            save_weights.append([kriging_weights])
            save_weights_rel_dat.append([np.multiply(kriging_weights,Zi)])
            save_Zi.append([np.ravel(Zi)])
            show_neighborhoods = {"tenth":(idx_show_data_support_tenth, ...
                idx_show_sim_support_tenth, idx_step_tenth),
                                  "quarter":(idx_show_data_support_quarter, ...
                                      idx_show_sim_support_quarter, idx_step_quarter),
                                  "half":(idx_show_data_support_half, ...
                                      idx_show_sim_support_half, idx_step_half),
                                  "three quarter":(idx_show_data_support_tq, ...
                                      idx_show_sim_support_tq, idx_step_tq),
                                  "final":(idx_show_data_support_final, ...
                                      idx_show_sim_support_final, idx_step_final)}

        # Count locations walked for search neighborhood
        locations_walked = np.append(locations_walked, idx)
        len_walked += 1

        # Save running variables
        save_invshape.append([np.shape(K_sys)])
        save_lagrange.append(float(np.array(lagrange).ravel()))
        save_kriging_mv.append([Zk, kriging_var])
        save_idx_nv.append([idx_n, idx_v])
        printProgressBar (len(locations_walked), N, subject = ' realization nr. ...
            %d' % run)


    print('Kriging variance less than zero:', var_lz)
    print("No. of no sim data used:", N_no_sim)
    return M, np.array(save_lagrange), np.array(save_kriging_mv), save_weights, ...
        save_invshape, save_weights_rel_dat, save_Zi, show_neighborhoods, ...
        np.array(save_idx_nv), save_lstsq, N_no_sim

def SDSSIM_sdssim(prior, semivar, condtab, setup, grid, data, greens, N_sim = 1, ...
    SN = 25, DN = 100, sort_method = "cut-off", neighborhood = "data_limited", ...
    kriging_method = "ordinary", oz = 'off', errorstd = 1, threshold_factor = 1.0, ...
    mean_CQF = False, mean_return = 'off', mean_burn_in = False, N_burn_in = 10):
    import numpy as np
    import time
    import random
    import gc

    """
    Neighborhood methods:
        - stochastic
        - data_limited
        - data_all
    Possible kriging_method(s):
        - simple
        - ordinary

    All combinations can be computed with either "threshold" or "cut-off" as sort ...
        method.
    """

    """efficiency ravels"""
    shape = condtab["CQF mean"].shape

```

```
    """Number of simulations"""
    zs = np.zeros((setup["N"],N_sim))
    zs_mean = np.zeros((setup["N"],N_sim))
    time_average = np.zeros((N_sim))

    """save variables"""
    idx_nv = list()
    lagrange = list()
    kriging_mv = list()
    rand_paths = list()
    invshapes = list()
    kriging_weights = list()
    kriging_weights_rel_dat = list()
    Zis = list()
    lstsq_param = list()

    """Start conditions"""
    print("")
    print("___________Starting SDSSIM with the following settings___________")
    print("")
    print("SN: %d" %SN)
    print("DN:", DN)
    print("N_sim: %d" %N_sim)
    print("neighborhood: %s" %neighborhood)
    print("kriging_method: %s" %kriging_method)
    print("sort_method: %s" %sort_method)
    if sort_method == "threshold":
        print("threshold_factor: %g" %threshold_factor)
    print("oz: %s" %oz)
    print("")

    """ Run sequential simulations"""
    for run in range(0,N_sim):

        gc.collect() # Garbage memory collect
        M = np.empty([setup["N"],],dtype=float)
        # Start timing
        t0 = time.time()
        random.seed(a=None)
        np.random.seed()

        # Initialize sequential simulation with random start
        idx_rnd = np.arange( setup["N"] )

        # Randomize index array to create random path
        random.shuffle(idx_rnd)

        """Run spherical direct sequential simulation"""
        M, save_lagrange, save_kriging_mv, save_weights, save_invshape, ...
            save_weights_rel_dat, save_Zi, show_neighborhoods, save_idx_nv, ...
            lstsq_sol, N_no_sim = sdssim(setup["N"], SN, DN, ...
            semivar["semi-variogram LUT"], idx_rnd, condtab, prior, grid, data, ...
            greens, shape, run+1, ozcorr = oz, sort_method = sort_method, ...
            neighborhood = neighborhood, kriging_method = kriging_method, ...
            threshold_factor = threshold_factor, mean_CQF = mean_CQF, mean_burn_in ...
            = mean_burn_in, N_burn_in = N_burn_in)

        # End timing
        t1 = time.time()

        # Keep all realizations
        zs[:,run] = M

        if np.logical_and(mean_return == 'on', setup["type"] == 'core'):
            zs_mean[:,run] = zs[:,run]
            zs_mean[prior["latitude mean"][:,0].astype(int),run] += ...
                prior["latitude mean"][:,1]

        # Plot statistics of run
        time_average[run] = (t1-t0)
        if time_average[run] < 60:
            print('Run time: %.3f' %(time_average[run]), 'seconds', '')
        elif time_average[run] < 3600:
```

```
            print('Run time: %.3f' %(time_average[run]*60**(-1)), 'minutes', '')
        else:
            print('Run time: %.3f' %(time_average[run]*60**(-2)), 'hours', '')
        if np.sum(time_average[:(run+1)])*60**(-1) > 60:
            print('Total elapsed time: %.3f' ...
                %(np.sum(time_average[:(run+1)])*60**(-2)), 'hours', '')
        else:
            print('Total elapsed time: %.3f' ...
                %(np.sum(time_average[:(run+1)])*60**(-1)), 'minutes', '')

        print('Variance: %.3f' %np.var(M))
        print('Mean: %.3f' %np.mean(M))
        print('Max: %.3f' %np.max(M))
        print('Min: %.3f' %np.min(M))
        print('Run nr.:', run+1)
        print('')

        idx_nv.append(save_idx_nv)
        lagrange.append(save_lagrange)
        kriging_mv.append(save_kriging_mv)
        rand_paths.append(idx_rnd)
        invshapes.append(save_invshape)
        kriging_weights.append(save_weights)
        kriging_weights_rel_dat.append(save_weights_rel_dat)
        Zis.append(save_Zi)
        lstsq_param.append(lstsq_sol)

    print('Mean avg. all: %.3f' %np.mean(zs.ravel()))
    print('Std.dev. all: %.3f' %np.std(zs.ravel()))
    print('Variance avg. all: %.3f' %np.var(zs.ravel()))
    print('Run time avg. all: %.3f' %np.mean(time_average), 'seconds', '')
    print('Total elapsed time: %.3f' %(np.sum(time_average[:(run+1)])*60**(-1)), ...
        'minutes', '')

    #% FORWARD PROBLEM REALIZATIONS
    B_pred = greens["G_d"]*zs
    rz_mean = np.matrix(np.mean(zs,axis=1)).T
    B_pred_mean = greens["G_d"]*rz_mean

    # MISFIT
    misfit_forward = ...
        np.sum(np.power((np.matrix(data["data"]).T-B_pred)/errorstd,2),0)/data["N"]
    misfit_prior = ...
        np.sum(np.power((np.matrix(prior["data"]).T-zs)/errorstd,2),0)/prior["N"]

    residual_forward = np.matrix(data["data"]).T-B_pred
    residual_forward_mean = np.matrix(data["data"]).T-B_pred_mean

    residual_prior = np.matrix(prior["data"]).T-zs
    residual_prior_mean = np.matrix(prior["data"]).T-rz_mean


    # SEMI-VARIOGRAM
    print("")
    print('Computing semi-variogram for each realization...')
    from SDSSIM_semivar import sv_sim_cloud
    import numpy as np
    l_var = 1
    lag_coarse = int(semivar["n_lags"]/l_var)
    cloud_coarse = int(semivar["max_cloud"]/lag_coarse)

    sph_d_sorted = semivar["sph_d_sorted"]

    sv_zs = sv_sim_cloud(lag_coarse, cloud_coarse, N_sim, zs, prior["N"], sort_d = ...
        semivar["sort_d"], data_type = setup["type"])
    sv_zs_mean = np.mean(sv_zs, axis=1)
    lags_posterior = ...
        np.array([np.mean(sph_d_sorted[n*cloud_coarse:cloud_coarse*(n+1)]) for n ...
        in range(0,lag_coarse)])

    #SAMPLE NEIGHBORHOODS


```

```
    realizations = {"realizations":zs, "realizations with mean":zs_mean, ...
        "mean":np.mean(zs.ravel()),
                    "variance":np.var(zs.ravel()), "Oz correction":oz, "return ...
                        Julien mean":mean_return,
                    "realization amount":N_sim, "SN":SN, "DN":DN, "run ...
                        time":time_average, "run time mean":np.mean(time_average), ...
                        "run time total":np.sum(time_average),
                    "idx_nv":idx_nv, "lagrange":lagrange, "kriging_mv":kriging_mv, ...
                        "random path":rand_paths, "threshold_factor":threshold_factor,
                    "sort_method":sort_method, "neighborhood":neighborhood, ...
                        "kriging_method":kriging_method, "data prediction":B_pred, ...
                        "realizations mean":rz_mean,
                    "data prediction mean":B_pred_mean, "realizations sv":sv_zs, ...
                        "realizations mean sv":sv_zs_mean,
                    "realizations sv lags":lags_posterior, ...
                        "misfit_forward":misfit_forward,
                    "misfit_prior":misfit_prior, "error variance":greens["errorvar"],
                    "kriging_weights":save_weights, ...
                        "residual_forward":residual_forward,
                    "residual_forward_mean":residual_forward_mean,
                    "residual_prior":residual_prior, ...
                        "residual_prior_mean":residual_prior_mean, "inv ...
                        shape":save_invshape, ...
                        "kriging_weights_rel":save_weights_rel_dat, ...
                        "lstsq_param":lstsq_sol, "Zi":save_Zi, ...
                        "neighborhoods":show_neighborhoods}
    print('Finished!')
    return realizations
```

DTU Space
National Space Institute
Technical University of Denmark

Elektrovej, building 328
DK - 2800 Kgs. Lyngby
Tlf. (+45) 4525 9500
Fax (+45) 4525 9575

www.space.dtu.dk