Space Science Reviews (2023) 219:15
https://doi.org/10.1007/s11214-023-00961-3

SPECIAL COMMUNICATION

Internal and External Jovian Magnetic Fields: Community Code to Serve the Magnetospheres of the Outer Planets Community

R.J. Wilson[1] · M.F. Vogt[2] · G. Provan[3] · A. Kamran[3] · M.K. James[3] · M. Brennan[4] · S.W.H. Cowley[3]

Received: 17 November 2022 / Accepted: 24 January 2023 / Published online: 13 February 2023

Abstract
We report on a new international community coding project to provide shared scientific computer code that performs common calculations to aid in planning scientific observations, modeling, and data analysis. We have developed code which calculates Jupiter's internal and external magnetic fields. All magnetic field model code is provided in four programming languages (C++, IDL, MATLAB and Python). The code is freely available on GitHub. For Jupiter's internal magnetic field, we present a number of spherical harmonic internal magnetic field models. These include JRM33, the latest Jupiter internal magnetic field model (Connerney et al. in J. Geophys. Res., Planets 127(2):e07055, 2022), as well as older jovian models (e.g. JRM09 (Connerney et al. in Geophys. Res. Lett. 45(6):2590–2596, 2018), O6 (Connerney in Planetary Radio Emissions III, pp. 13–33, 1992), VIP4 (Connerney et al. in J. Geophys. Res. 103(A6):11,929–11,940, 1998) and VIPAL (Hess et al. in J. Geophys. Res. Space Phys. 116(A5):A05217, 2011)). The internal magnetic field code can be easily modified for other planets by simply inputting another spherical harmonic magnetic field model. We have also developed code to calculate the magnetic field perturbations due to the azimuthal and radial currents flowing externally around Jupiter in the jovian magnetodisc according to the model of Connerney et al. (J. Geophys. Res. 86(A10):8370–8384, 1981; J. Geophys. Res. Space Phys. 125(10):e28138, 2020). The internal and external magnetic field codes can be combined to model the magnetic field in Jupiter's magnetosphere. Finally, we provide field-line tracing software (C++ and a Python wrapper for C++) that utilizes the internal and external magnetic field models. The software can be used to trace along field lines from any position in the jovian magnetosphere to, for example, the ionosphere or an equator, and can also be utilized at different planets.

Keywords Jupiter · Magnetic field · Spherical harmonic model · Current sheet model · Community code

Note by the Editor: This is a Special Communication. In addition to invited review papers and topical collections, Space Science Reviews publishes unsolicited Special Communications. These are papers linked to an earlier topical volume/collection, report-type papers, or timely papers dealing with a strong space-science-technology combination (such papers summarize the science and technology of an instrument or mission in one paper).

Extended author information available on the last page of the article

1 Introduction

Spherical harmonic models of Jupiter's planetary magnetic field were first developed from Pioneer 11 data using the vector helium magnetometer observations (Smith et al. 1976) and the high field flux gate magnetometer observations (Acuña and Ness 1976). Data from Pioneer 10 and 11 also showed the distortion of the magnetic field at low latitudes and equatorial regions by the jovian magnetodisc. The existence of a jovian magnetodisc feature was first suggested by Gledhill (1967) (see also Caudal (1986)). Magnetic field perturbations associated with this external magnetodisc made it difficult for Ness et al. (1979) to develop an internal field model from Voyager 1 magnetometer observations. Connerney et al. (1981) developed a model of the magnetic field perturbations associated with the large-scale external magnetodisc current system. This has become known as the CAN current sheet model (Connerney, Acuña and Ness). Connerney et al. (1982) then developed a model of Jupiter's magnetic field from Voyager data, using an internal spherical harmonic expansion combined with an explicit model of the magnetic field created due to the external current systems. This two-step approach, considering the internal and external magnetic fields separately, is now seen as essential when modeling the magnetic field in Jupiter's magnetosphere.

The ability to accurately model both the internal and external magnetic fields of Jupiter is, of course, of great importance to colleagues studying the jovian magnetic environment. Over the years a number of internal magnetic field models of Jupiter's magnetic field have been developed. These include O6 (Connerney 1992), VIP4 and VIT4 (Connerney et al. 1998), VIPAL (Hess et al. 2011), ISaAC (Hess et al. 2017), JRM09 (Connerney et al. 2018), and JRM33 (Connerney et al. 2022). All these models present the internal magnetic field in terms of a spherical harmonic expansion, with the values of the coefficients and the order of the expansion varying between the models. Both Connerney (2007) and Russell and Dougherty (2010) provide good introductions to spherical harmonic models generally, and of models for different planets, while the methods of Imai (2016) provide the source material for this work. A number of different external field models have previously been developed (e.g. Khurana (1997); Alexeev and Belenkaya (2005); Khurana and Schwarzl (2005); Pensionerov et al. (2019)), including the CAN magnetodisc model that parameterizes the jovian current sheet by a current sheet field parameter, the radii of the inner and outer edges of the current sheet, and current sheet thickness. This work uses Con2020, an updated CAN model, described by Connerney et al. (2020). More recently, Wang et al. (2021) presented a global magnetic field model for Jupiter, and Wang et al. (2022) developed a new empirical model for the Jovian current sheet. Just as this paper was being finalized (too late to be considered for code here), Khurana et al. (2022) presented a model to determine the global structure of Jupiter's current sheet, developed from spacecraft data at Jupiter (Pioneers 10 and 11, Voyagers 1 and 2, Ulysses, Galileo, but not Juno) to determine the global structure of Jupiter's current sheet.

Many colleagues have written their own code to calculate the various internal field models, with the code shared informally between colleagues. However, confusion can easily occur when swapping from one field model to another, with different models expecting different inputs and outputs, using different units and defining key parameters slightly differently. For example, the value of one jovian radius (R_J) has changed over time, as has the definition of the System III co-ordinate system. Furthermore some models express the magnetic field in Gauss (G), and others in nano-Tesla nT (where 1 G $= 10^5$ nT). It is because of the importance of these magnetic field models to the Magnetospheres of the Outer Planets (MOP) community that we decided to develop a community coding repository, sharing code that is well-written, clearly-documented and extensively tested. All magnetic field

model code is presented in IDL, MATLAB, Python and C++, with some additional C++ code also included for field-line tracing and other functionalities. We have standardized the inputs and the outputs to the magnetic field model, as described in Sect. 2 below. Collecting these coefficients was not always trivial, as different papers defined the coefficients to different numbers of significant figures and sometimes typographical errors have been introduced over the years. However, this only exemplifies the importance of producing standardized models and codes for community use.

2 Standards

The codes presented here are standardized to take the same input forms, and give the same output forms. The coordinate system in use is the right handed System III (1965), which defined Jupiter's rotation rate as 870.536° per day. Colleagues should be aware however, especially when looking at older datasets, that there was a 1957 version of System III that defined Jupiter's rotation rate as 870.544° per day. This was superseded by the 1965 version, but some of the earlier Jupiter spherical harmonic models used this System III (1957) system. (For historical context, Table 3 in Russell and Dougherty (2010) provides details on Systems I, II, III (both 1957 and 1965) and IV.)

SPICE (Acton 1996) is a common tool for calculating planetary and spacecraft positions and performing rotations into various coordinate systems. It is commonly assumed that the SPICE frame 'IAU_JUPITER' is equivalent to a right handed System III. At the time of writing, the current SPICE planetary kernel is 'pck00010.tpc', which is dated 2011 October 21 and uses the 1965 System III definition. The same was true for the first available planetary kernel, 'pck00003.tpc' dated 1990 June 25 (versions 1, 2 and 4 have been lost to time (personal communication with SPICE team)). However, the IAU had a blip, where Jupiter's spin period was defined as a different rate, before reverting back. This affected three planetary kernels, pck00007.tpc (dated 2000 April 24), pck00008.tpc (2004 September 21) and pck00009.tpc (2010 March 03), which all used a Jupiter rotation rate of 870.5366420° per day. Therefore, using SPICE for the IAU_JUPITER frame (assuming it was System III (1965)) from early 2000 to late 2011 would have produced an offset.

SPICE planetary kernels define the equatorial radius of Jupiter, 1 $R_J = 71{,}492$ km, and all the codes we have developed and describe here uses this (1 bar radius) IAU standard. All the codes then take input position distances in units of this jovian radii (R_J). However, historically, that has not always been the case, and published spherical harmonic models have also used 71,372 km, 71,398 km, or 71,323 km as the definition of one jovian radius (further detail is provided in later sections). If the spherical harmonic model called uses a different R_J definition, then our code will expect the input in terms of 1 $R_J = 71{,}492$ km, and will re-scale the input internally in the code (by multiplying input distances by $71492/R_J^{Model}$ prior to the spherical harmonic calculations).

There are two main sets of code options: Cartesian or Spherical. For Cartesian inputs, position is provided in right-handed System III (1965) x, y and z (each in units of R_J). The Cartesian output is the magnetic field vector $\vec{B} = [B_x, B_y, B_z]$ with all three elements in units of nT. For Spherical inputs, position is provided in right-handed System III (1965) radial distance (r, units of R_J), Co-Latitude (θ, units of radians, 0-π), and East Longitude (ϕ, units of radians, 0-2π). The Spherical output is the magnetic field vector $\vec{B} = [B_r, B_\theta, B_\phi]$ with all three elements in units of nT.

All the codes will accept inputs as scalar values, or 1-dimensional (1D) arrays. For instance, if you want to know the internal field contribution of B along a known trajectory,

then passing a 1D array of all the positions is much faster computationally that calling the code one position at a time.

The codes in C++, MATLAB, IDL and Python have been tested against each other, and found to provide the same outputs given the same inputs to within rounding errors (of the order of 2×10^{-10} nT or less).

For all models in our codes, the degree and order of the spherical harmonic expansions are equal. Therefore, our codes use the words interchangeably, or only give the value for one of them, but the value would apply to both degree and order.

To summarize this section, inputs must be in right-handed System III (1965) co-ordinates, with distances in R_J where 1 R_J = 71,492 km, the position angles (Spherical only) in radians, and outputs are always magnetic field in nT.

3 Spherical Harmonic Internal Planetary B-Field

3.1 Spherical Harmonics Basic Equations

The magnetic field environment about a planetary body is comprised of contributions from an internal magnetic field and an external magnetic field model:

$$B = B_{Internal} + B_{External} \tag{1}$$

The external contribution varies with planet and methodology of modeling, such as a current sheet model used for Jupiter, discussed later in this paper. A planet's internal magnetic field is often modeled using spherical harmonics, where variations in the field are built up through a series of trigonometric Legendre polynomials. A set of coefficients to these polynomials generate a planet's particular internal magnetic field, fitted from past observations.

The internal magnetic field is calculated using a scalar potential, V, (e.g. Winch et al. 2005):

$$B_{Internal} = -\nabla V \tag{2}$$

$$= B_r\hat{r} + B_\theta\hat{\theta} + B_\phi\hat{\phi} \tag{3}$$

where

$$V = \sum_{n=1}^{n_{max}} \left(\frac{1}{r}\right)^{n+1} \sum_{m=0}^{n} P_n^m(\cos\theta)[g_n^m \cos(m\phi) + h_n^m \sin(m\phi)] \tag{4}$$

$P_n^m(\cos\theta)$ is a Schmidt-normalized Legendre function of degree n and order m, g_n^m and h_n^m are the Schmidt coefficients for the internal field.

In spherical polar coordinates, where $\nabla = \left(\frac{\partial}{\partial r}, \frac{1}{r}\frac{\partial}{\partial\theta}, \frac{1}{r\sin\theta}\frac{\partial}{\partial\phi}\right)$, the solution to equations (2)-(4) for the three components of the magnetic field are:

$$B_r = -\frac{\partial V}{\partial r} = \sum_{n=1}^{n_{max}} \frac{(n+1)}{r^{n+2}} \sum_{m=0}^{n} [g_n^m \cos(m\phi) + h_n^m \sin(m\phi)] P_n^m(\cos\theta) \tag{5}$$

$$B_\theta = -\frac{1}{r}\frac{\partial V}{\partial\theta} = -\sum_{n=1}^{n_{max}} \frac{1}{r^{n+2}} \sum_{m=0}^{n} [g_n^m \cos(m\phi) + h_n^m \sin(m\phi)] \frac{\mathrm{d}}{\mathrm{d}\theta} P_n^m(\cos\theta) \tag{6}$$

$$B_\phi = -\frac{1}{r\sin(\theta)}\frac{\partial V}{\partial \phi} = -\frac{1}{\sin(\theta)}\sum_{n=1}^{n_{max}}\frac{1}{r^{n+2}}\sum_{m=0}^{n} m[-g_n^m \sin(m\phi) + h_n^m \cos(m\phi)]P_n^m(\cos\theta) \tag{7}$$

Further details of the derivation and formulation of the spherical harmonic modeling of planetary magnetic fields can be found in numerous other works, i.e. Imai (2016) and Connerney (1993).

3.1.1 Pitfalls of Spherical Harmonic Models

The Spherical Harmonic coefficients are found by fitting data close to the planet, but apply at all distances (the contribution of the higher order terms fall off as distance increases). As one moves further from the planet, the internal planetary field is superposed with other sources of magnetic fields, such as those due to current flow in the plasma disk. Therefore, the internal field represented by the Spherical Harmonic model is, by itself, not a good representation of the observed magnetic field at large distances and one should superpose on external field models to get a better observational match.

The equations do have a mathematical singularity to avoid: Spherical Harmonic equation (7) has a divide by $\sin(\theta)$ (sine of colatitude), which goes to infinity if $\sin(\theta)$ goes to zero, i.e. at the north or south pole. This situation is rare, so far no spacecraft has ever come close to being that closely aligned to a pole, but it is plausible that one might encounter this situation when performing field line tracing. Our codes behave differently when $\sin(\theta) < 0.00001$, which translates to colatitudes less than 0.000573° or greater than 179.999427°, but the code will still return a value rather than an infinity or crashing the code. It is unclear what the best equations to use in such a situation are, and we have yet to find a published solution. Since it is such a rare occurrence, we have simply borrowed techniques found in previous codes without questioning the origin. However, this situation is the biggest item on where the Spherical Harmonic code could be improved. If concerned, it is up to the user to check whether their input positions are this closely aligned with the pole, and if so, to decide if the code outputs look sensible.

As previously stated, the codes presented here assume 1 R_J = 71,492 km, and if the model requested used a different value, the code will automatically adjust distances internally to match the value of R_J that model expected; that is all invisible to the code user. However, if the user is extracting input positions from older datasets in units of R_J, the user should check which R_J value they used, and if not 71,492 km, they should rescale those values to the 1 R_J = 71,492 km equivalent.

Similarly, some older models originally used a co-ordinate system different to System III (1965). If those models are in code form here, they have been rotated into System III (1965). If the input position is from an older dataset that was not in System III (1965), then one will need to convert to System III (1965) before use.

3.2 Jupiter Internal Magnetic Field Models

There have been many previous spherical harmonic models used to represent Jupiter's internal magnetic field over the years, starting from the Pioneer data. Many of these are simply too old (or to too few orders, based on a single fly-by) to still be in use, so our work focuses on the subset of models that are commonly used today.

Many, excluding the latest JRM models, are summarized in Connerney (2007), and their Table 3 provides the coefficients to many of these (although units are nT, not G as stated).

Table 1 Spherical harmonic information of the different time independent models

	Order	Fit Range[a]	Datasets[b]	R_J^{ref} in km[c]
O6	3	$1.6 < R_J < 10$	P11, V1	71372
VIP4[d]	4	$2.4 < R_J < 14$	P11, V1, U + Hubble + Ground Telescopes	71323
VIT4	4	$R_J < 7$	V1 + Hubble + Ground Telescopes	71323
VIPAL[e]	5	$4 < R_J$	P10, P11, V1 + Hubble	71492
ISaAC	10	$R_J < 10$	P10, P11, V1, U, G + Hubble	71492
JRM09	10	$R_J < 7$	J	71492
JRM33	13 or 18	$R_J < 2.5$	J	71492

[a]Extreme bounds if multiple spacecraft were used.

[b]P10 = Pioneer 10, P11 = Pioneer 11, V1 = Voyager 1, U = Ulysses, G = Galileo, J = Juno.

[c]If $R_J^{ref} \neq 71492$, our code multiplies input distances by $71492/R_J^{ref}$ to adjust for the different R_J each model expects.

[d]Connerney et al. (1998) state VIP4 is accurate for distances $< 30R_J$.

[e]Hess et al. (2011) state VIPAL is well fitted for distances $< 15R_J$, but other models should be used further out.

Each model's coefficients have been converted to System III (1965) co-ordinates, and are often listed to more significant figures than in the original papers while also changing units to nT, e.g. O4 was published in Table 1 of Connerney et al. (1982) in units of G, while VIP4 was originally published in Connerney et al. (1998) to 3 decimal place, in units of G, while Connerney (2007) effectively lists them to 5 decimal places in G (then converts to nT).

We note that Table 3 of Connerney (2007) lists the SHA model g_3^0 value incorrectly as 11300 (nT). The earlier Connerney et al. (1982) lists the same value as being negative, −0.113 G, quoting it from the even earlier Smith et al. (1976) (page 803) that had been calculated in System III (1957), but also lists $g_3^0 = -0.113$ G. Hence, we believe this to be a typographic error, and that for the SHA model $g_3^0 = -11300$ nT. None of these three sources mention alongside their tables which R_J to use for SHA, but do give R_J earlier in their texts: Smith et al. (1976) (from page 20) used 71,398 km, while Connerney et al. (1982) (page 3623) stated 71,323 km. Connerney (personal communication) suggests to always use the value of the original publication, thus we recommend using 71,398 km for SHA.

The following sections provide comments on the models that have been in use in recent years, for which we have developed code. The models are also summarized in Table 1. All seven of these time independent models were developed in the System III (1965) coordinate system, however they did not all use the same R_J value, and fitted to data from different radial ranges of the magnetosphere.

3.2.1 O6

The O6 model is of order 3, with coefficients provided in units of G to 5 decimal places as presented in Table 1 of Connerney (1992) (with the warning that 5 decimal places is to aid

conversion to nT, and not an indication of parameter accuracy). The O6 model assumed 1 $R_J = 71{,}372$ km, and for the fitting, used Pioneer 11 data out to 5 R_J, and Voyager 1 data out to 10 R_J. Connerney et al. (1998) also presents O6 coefficients in G to 3 decimal places (see their Table 1). Our codes for O6 are order 3, with Connerney (1992) Table 1 values converted from G to whole nT.

Yu et al. (2010) repeated the O6 model method (order of 3) but used 22 orbits of Galileo data as the input to look for secular variation. However, the planetary moment they calculated was, within error, the same as that for the O6 model that used Pioneer data, so no secular variation was identified. While they provided the g and h values as calculated from Galileo data, we have not provided code for that model here since it does not significantly improve on the O6 model concept.

3.2.2 VIP4 and VIT4

VIP4 and VIT4 are spherical harmonic models of order 4 that are constrained by Io's flux tube footprint. VIP4 fits used 100 locations of the Io flux tube footprint, and the magnetometer data from Pioneer 11 between 2.4 to 8.0 R_J, Voyager 1 data out to 10 R_J and Ulysses data out to 14 R_J. Whereas VIT4 used 500 locations of the Io flux tube footprint and just the theta component (hence the T in VIT4) of the Voyager 1 magnetometer data within 7 R_J. The models are originally presented in Connerney et al. (1998), with Table 1 giving values for the VIP4 coefficients in G to 3 decimal places (but not VIT4). However, in a later book, Connerney (2007), Table 3 lists the VIP4 and VIT4 coefficients in whole nT (e.g. equivalent to 5 decimal places in G), however incorrectly presents them in units of G rather than nT, while citing the original paper for VIT4.

VIP4 and VIT4 coefficients are thus given in nT to 0 decimal place, were calculated in System III (1965) and with 1 $R_J = 71{,}323$ km. The value of 71,323 km comes from Table 1 of Connerney et al. (1998), although earlier in their text when explaining spherical harmonic expansions they state "where a is the equatorial radius of Jupiter (71,398 km)". The later book does not resolve the issue, first claiming that Jupiter has a radius of 71,372 km and not explicitly providing the R_J value used for VIP4 (or VIT4) in the table caption (although it does for earlier Ulysses and Voyager models). However, when referring to Table 3 it states "The original publication should be consulted." Hence we will assume these models were created with $1 R_J = 71{,}323$ km.

Our codes for VIP4 and VIT4 are to order 4, with the coefficient values taken from Table 3 of Connerney (2007), with the units assumed to be nT.

Hess et al. (2011) also list VIP4 and VIT4 values in their Table 3, but to a lower precision, rounded 4 decimal places in units of G, but with two exceptions for VIT4 only: their value for VIT4 g_1^0 was rounded to 3 decimal places in G, and their value for VIT4 h_4^4 of 0.0126 G was likely mistyped, and is likely meant to have been 0.1264 G.

3.2.3 VIPAL

VIPAL (Hess et al. 2011) utilized Pioneer and Voyager data, together with the position of jovian moon footprints in the aurora, as viewed by Hubble, to better constrain their spherical harmonic model, with the aim to give a "better description of the satellite-related aurorae". They note that calculating dipole moments from the fit of the Io footprint positions is $\approx 5°$ off from that when calculated from the magnetic field alone, which appears to be directly related to using the IAU rotation rate of Jupiter at the time (presumably SPICE planetary kernels 7, 8 or 9), rather than the rotation rate of System III (1965), see their paper for further discussion.

VIPAL coefficients are presented in Table 3 (and also in their Supporting Information of Table 3 as a text file) in G to 1 to 4 decimal places depending on the coefficient, were calculated in System III (1965) and with $1 R_J = 71{,}492$ km, and only fitted to magnetometer data from distances above 4 R_J. The VIPAL authors state that the aim of their model was for investigating satellite related aurora, and suggest other field model codes are used for distances > 15 R_J.

Our codes for VIPAL are to order 5, with coefficient values defined in Hess et al. (2011) Table 3, converted from G to nT.

3.2.4 ISaAC

ISaAC (Hess et al. 2017) is an update to the VIPAL model, combining magnetometer data with auroral footprints of the moons, with the following stated aim: "The main purpose of the present model is to be an engineering model to perform studies requiring an accurate modeling of the magnetic field line topology and to process Juno data before the final Juno model is published." They add Ulysses and Galileo data to the previously used Pioneer and Voyager data.

ISaAC coefficients are shown in their Table 1 in G to 4 to 6 decimal places depending on the coefficient, were calculated in System III (1965) with 1 $R_J = 71{,}492$ km, and only fitted magnetometer data at distances below 10 R_J. The model is to order 10, but they state "high degree coefficients should not be considered as physically meaningful."

Our codes for ISaAC are order 10, with their Table 1 values converted from G to nT.

3.2.5 JRM09

JRM09 (Connerney et al. 2018) uses the first 9 orbits of Juno to fit a 20 order spherical harmonic model to the Juno magnetic field data observed within 7 R_J of Jupiter. The JRM09 coefficients were calculated in System III (1965), with 1 $R_J = 71{,}492$ km. The JRM09 coefficients are given in nT to 0 decimal place. The values of the coefficients up to order 10 are provided in Table S1 of Connerney et al. (2018) Supporting Material (higher orders of 11 to 20 are "poorly constrained, or unconstrained"). However, the coefficients for the full 20 orders fit are available on the Planetary Data System (PDS) by Connerney (2017) in the DATA/MODEL/JRM09 directory.

Our codes for JRM09 are to order 10, to match Table S1 of Connerney et al. (2018). It is not recommended to try to use JRM09 up to spherical harmonic order 20.

3.2.6 JRM33

JRM33 (Connerney et al. 2022) uses the first 33 orbits of Juno to fit an order 30 spherical harmonic model to 30 s averaged magnetic field data observed within 2.5 R_J of Jupiter. The use of the first 33 orbits completes Juno's primary mission map of sampling perijoves around Jupiter at 32 evenly spaced System III (1965) longitudes (Juno went into safe mode on approach to perijove 2, hence required 33 orbits to complete the map), each $\approx 11.25°$ apart.

JRM33 coefficients are given in nT to 1 decimal place and were calculated in System III (1965) with 1 $R_J = 71{,}492$ km. The values of the coefficients are provided in Table S1 of Connerney et al. (2022). The same file of values is also on the PDS at Connerney (2017) in the DATA/MODEL/JRM33 directory. Personal communication with the authors

suggests that most people should use JRM33 up to order and degree 13, since higher order coefficients are not "well resolved". In Connerney et al. (2022) they state "Therefore, in applying the model to calculate the field where it has not been measured, we use terms through degree 18 only, or through degree 13 only, if being more conservative. The satellite footprints calculated in Tables S2 and S3 use terms through degree 18".

Codes for JRM33 order 13 and order 18 are included in the community code, it is not recommended to try to use an order 30 set.

3.2.7 Time-Dependent Models Excluded from Our Coding Work

Here we briefly note the following time-dependent spherical harmonic models of Jupiter's internal field, which we have excluded from our coding work:

- **JCF and JSV**: Ridley and Holme (2016) also provided some order 7 spherical harmonic fits in their Table 3, that are time dependent. They used magnetometer data from both Pioneers, both Voyagers, Ulysses and Galileo.
- **The Sharan et al. (2022) Jupiter Field Model**: Sharan et al. (2022) use data from the first 4 years of Juno at Jupiter to simultaneously compute spherical harmonic models for a static field to 16 orders and a time varying spherical harmonic field model to 8 orders. They used data from the first 28 orbits, excluding data from orbit 2 (where there was no data due to Juno safing) and orbit 19 (due to "spurious oscillation" in the data).
- **Bloxham et al. (2022)** is a companion study to the JRM33 paper that further investigates the first 33 orbits of Juno data for secular variation, which they identify.

3.3 The JupiterMag and libjupitermag Packages for Spherical Harmonics

This section describes the installation and basic usage of the `libjupitermag` C++ library (James et al. 2022b) and its Python-based wrapper, `JupiterMag` (James et al. 2022a). This code provides access to the spherical harmonic models discussed in Sect. 3.1.1 (among other functionalities) and has been tested under Linux Mint 20.3, Windows 10 and MacOS 11 Big Sur. The `JupiterMag` module provides an interface which allows the models contained in `libjupitermag` to be called by Python.

3.3.1 Compiling and Installing libjupitermag

The compilation of `libjupitermag` requires the installation of `g++` (part of `gcc`), GNU `make` (for Linux and Mac OS) and `ld` (Linux) or `libtool` (Mac OS). In Windows, this would require the installation of TDM-GCC (https://jmeubank.github.io/tdm-gcc/). In MacOS, installing Xcode will provide the necessary tools for compiling the code. Finally, in Linux the tools may be installed from the Linux distribution's repositories, e.g. for Debian-based systems:

```
$ sudo apt install build-essential
```

or RPM-based systems:

```
$ sudo dnf install make gcc gcc-c++
```

Once the utilities required for building the library are installed, then the GitHub repository can be cloned:

```
$ git clone --recurse-submodules https://github.com/mattkjames7/libjupitermag.git
```

then compiled for Linux or Mac OS:

```
$ cd libjupitermag
$ make

#optionally fully install the library system-wide
$ sudo make install
```

equivalently in a Windows command line or PowerShell:

```
> cd libjupitermag
> compile.bat
```

If the compilation is successful, then a new library file will be created in the `lib/libjupitermag` subdirectory (`libjupitermag.so` or `libjupitermag.dll`) and a header file is placed within the `include` subdirectory (`jupitermag.h`) (for Windows paths, swap / with \).

If the library is installed system-wide on a Mac or Linux machine, then the header will be placed in `/usr/local/include` and the shared object file in `/usr/local/lib` directories by default. The header can simply be included into a C++ source file using `#include <jupitermag.h>` and the library can be linked by using the `-ljupitermag` flag. If it is to be used locally instead, then either a full or a relative path must be used to include the header (e.g. `#include "/abs/path/to/jupitermag.h"`) and the full or relative path to the library must be provided to the linker during compilation.

To report any bugs or for feature requests, please visit https://github.com/mattkjames7/libjupitermag/issues to submit a new issue.

3.3.2 Installing `JupiterMag`

There are multiple methods for installing the `JupiterMag` Python module - all of which require an installation of Python 3, and the following Python packages: NumPy, Matplotlib, DateTimeTools, RecarrayTools and PyFileIO.

The easiest and preferred method is to use Python's `pip` command as this will automatically install any dependencies, e.g.:

```
$ pip3 install JupiterMag --user
```

Another option would be to visit the releases page for the GitHub repository (https://github.com/mattkjames7/JupiterMag/releases), download the latest release and install using `pip` e.g. for version 1.0.8:

```
$ pip3 install --user JupiterMag-1.0.8-py3-none-any.whl
```

For the very latest code, the GitHub repository could be cloned directly,

```
$ git clone https://github.com/mattkjames7/JupiterMag.git
$ cd JupiterMag
```

then installed with `pip`

```
$ python3 setup.py bdist_wheel
$ pip3 install dist/JupiterMag-1.0.8-py3-none-any.whl --user
```

or without `pip`,

```
$ python3 setup.py install --user
```

Note that this installation method may include the latest bugs along with the latest features!

The `JupiterMag` Python module contains the `libjupitermag` library within it, precompiled for Linux and Windows. If it cannot be loaded when `JupiterMag` is imported for the first time in Python, then it will attempt to automatically recompile itself for the host system. In order to be able to recompile itself, it requires the tools listed in Sect. 3.3.1.

The `JupiterMag` package can easily be upgraded if `pip3` was used to install it, e.g.:

```
pip3 install JupiterMag --user --upgrade
```

which will replace the currently installed code with the latest release.

For issues or bugs relating to `JupiterMag`, please report them at https://github.com/mattkjames7/JupiterMag/issues.

3.3.3 Examples

In this section we provide a few simple examples of how to call the internal field model code contained within the `JupiterMag` Python module and the `libjupitermag` C++ library. Section 4.3 will discuss using them for external field models. All Python examples shown below should be run either in a Python script or a Python shell and C++ examples should be compiled and run in a terminal.

In Python, the internal field models are accessed via the `JupiterMag.Internal` module, which contains 6 functions including `Config()` and `Field()`. The `Config()` function can be used to set and return the current model configuration and should be called every time the model settings need to be altered. To return and display the current internal field model configuration:

```
import JupiterMag as jm

cfg = jm.Internal.Config()
```

where `cfg` is a dictionary. The `cfg` dictionary contains the following keys:

- `'Model'`: Lower case string containing the name of the model.
- `'CartesianIn'`: Boolean, if `True` then the input positions to the `Field()` function should be Cartesian, otherwise they should be spherical polar.
- `'CaresianOut'`: Boolean, if `True` then the output of `Field()` is Cartesian, or spherical polar if `False`.
- `'Degree'`: Integer - maximum degree (or order) of the model to use.

Each of the keys used in the `cfg` dictionary can also be used as a keyword argument in order to reconfigure the model, e.g.

```
# use the JRM33 model
jm.Internal.Config(Model='jrm33')

# set the input and output coordinate systems to spherical polar
jm.Internal.Config(CartesianIn=False,CartesianOut=False)

# set the maximum model degree to 5
jm.Internal.Config(Degree=5)
```

where the configuration dictionary returned by `cfg = jm.Internal.Config()` would be

```
{'Model':'jrm33', 'CartesianIn':False, 'CartesianOut':False, 'Degree':5}
```

The configuration may also be set using a dictionary, e.g.:

```
# create the dictionary
cfg = { 'Model' : 'o6',
        'CartesianIn' : True,
        'CartesianOut' : True,
        'Degree' : 3}


# alter config
jm.Internal.Config(**cfg)
```

Finally, the configuration can be returned to the default using:
`jm.Internal.Config('default')`
where the default configuration uses the degree 10 JRM09 model with Cartesian coordinates for both input and output. When configuring the model, invalid model strings will be ignored (the previous model will continue to be used) and the degree (or order, since degree = order in the codes) of the model must be an integer value between 1 and the maximum degree of the model.

Once the internal field model has been configured, the `Field()` function can be called to provide magnetic field vector(s). This function requires three input arguments, `b0`, `b1` and `b2` which are either the Cartesian x, y and z System III (1965) coordinates (if CartesianIn=True) or spherical polar coordinates r, θ and ϕ. `Field()` also accepts the optional `MaxDeg` keyword which temporarily overrides the degree of the model being used. Some examples are shown below,

```
import numpy as np
import JupiterMag as jm

# configure the model to accept Cartesian coordinates
jm.Internal.Config(CartesianIn=True,CartesianOut=True)

# Field() accepts scalars, lists, tuples and NumPy arrays
Bx,By,Bz = jm.Internal.Field(5.0,0.0,0.0)
Bx,By,Bz = jm.Internal.Field([5.0,6.0],[0.0,0.0],[0.0,0.0])
Bx,By,Bz = jm.Internal.Field((5.0,6.0),(0.0,0.0),(0.0,0.0))
Bx,By,Bz = jm.Internal.Field(np.array([5.0,6.0]), # continue on next line
                             np.array([0.0,0.0]), np.array([0.0,0.0]))

# configure polar coordinates
jm.Internal.Config(CartesianIn=False,CartesianOut=False)

# temporarily adjust the model degree
r = np.array([5.0,6.0])
theta = np.array([np.pi/2.0,np.pi/2.0])
phi = np.array([0.0,0.0])
Br,Bt,Bp = jm.Internal.Field(r,theta,phi,MaxDeg=5)
```

The `JupiterMag.Internal` submodule contains four other functions, each of which provides a test output. `JupiterMag.Internal.Test()` produces a figure similar to Fig. 4 of Connerney et al. (2018) showing the radial component of the JRM09 field model at $r_c = 0.85\ R_j$. `JupiterMag.Internal.TestOutput()` produces a terminal output of the model components at a few select positions in space. `JupiterMag.Internal.JRMFig5()` and `JupiterMag.Internal.JRMFig7()` produce plots based on Figs. 5 and 7 of Connerney et al. (2022). The output of `JupiterMag.Internal.JRMFig5()` is presented in Fig. 1, showing the radial component of the JRM33 magnetic field model at the presumed dynamo source region using the default degree 13 model (panel a) and the degree 18 model (panel b) as presented in Connerney et al. (2022).

Similarly, in C++ the spherical harmonic code can be accessed via an instance of the `InternalModel` class, e.g.

```
/* contents of example.cc */
#include <stdio.h>
#include <jupitermag.h>

int main() {

    /* create an instance of the InternalModel class */
    InternalModel model;
```

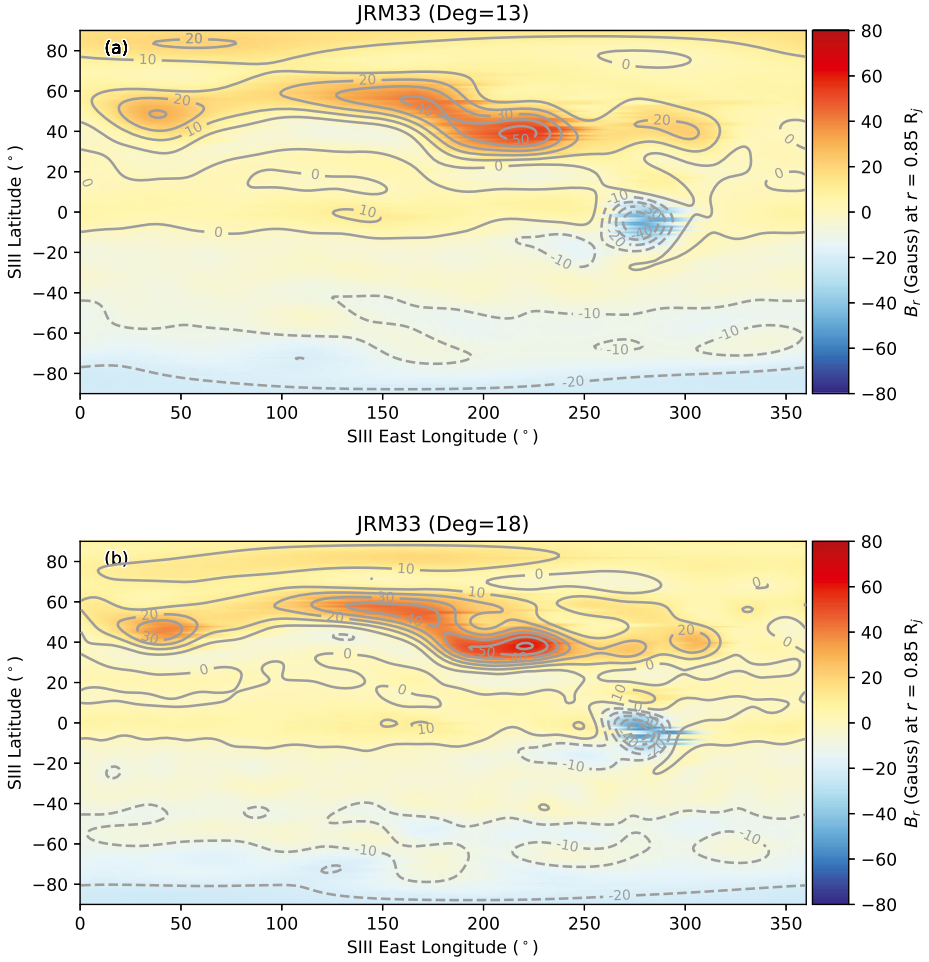

Fig. 1 A map of the radial component of the JRM33 degree/order 13 (a) and 18 (b) magnetic field model at $r_c = 0.85\ R_J$ in System III (1965) coordinates

```
/* tell the object to use a specific model */
model.SetModel("jrm09");

/* also tell it whether the input and output coordinates
 * are spherical polar or Cartesian */
model.SetCartIn(true);
model.SetCartOut(true);

/* either evaluate at a single position */
double x = 10.0;
double y =  5.0;
double z =  2.0;
double Bx,By,Bz;
model.Field(x,y,z,&Bx,&By,&Bz);
```

```
    printf("B=[%6.1f,%6.1f,%6.1f] at [%4.1f,%4.1f,%4.1f]\n",
            Bx,By,Bz,x,y,z);

    /* or at an array of positions */
    double  xa[5],  ya[5],  za[5];
    double Bxa[5], Bya[5], Bza[5];
    int i;
    for (i=0;i<5;i++) {
        xa[i] =  5.0*(i + 1);
        ya[i] =  0.0;
        za[i] = 10.0;
    }
    model.Field(5,xa,ya,za,Bxa,Bya,Bza);
    for(i=0;i<5;i++) {
        printf("B=[%6.1f,%6.1f,%6.1f] at [%4.1f,%4.1f,%4.1f]\n",
                  Bxa[i],Bya[i],Bza[i],    xa[i],ya[i],za[i]);
    }

    return 0;
}
```

which can be compiled and ran on Linux (as an example) using:

```
$ g++ example.cc -o example -ljupitermag
$ ./example
```

and would print out the following B-field vectors at the positions evaluated by the model:

```
B=[  82.1,   5.4,-265.9] at [10.0, 5.0, 2.0]
B=[ 372.8, -12.1, 344.0] at [ 5.0, 0.0,10.0]
B=[ 203.0,  -6.9,  33.7] at [10.0, 0.0,10.0]
B=[  83.1,  -3.5, -22.0] at [15.0, 0.0,10.0]
B=[  34.9,  -1.9, -22.1] at [20.0, 0.0,10.0]
B=[  15.9,  -1.1, -15.9] at [25.0, 0.0,10.0]
```

In the C++ example shown above, the `model` object is an instance of the `InternalModel` class with member functions `InternalModel::SetModel(const char *)` for selecting the model to use; `InternalModel::SetCartIn(bool)` and `InternalModel::SetCartOut(bool)` which determine whether the input and output coordinates are Cartesian or spherical polar System III (1965). The `InternalModel::Field()` member function outputs magnetic field vectors when provided with System III (1965) position vectors. It is an overloaded function which can use various input/output argument types (see `jupitermag.h` for more details) and can therefore provide individual components of a single vector, or 1D arrays of components for multiple B-field vectors.

3.4 The `PSH` Codes for Spherical Harmonics

The Planetary Spherical Harmonic, `PSH`, codes are available in IDL, MATLAB and Python, with separate codes per field model per Cartesian or Spherical coordinate system.

3.4.1 Notes

Equivalent codes are available in MATLAB, IDL and Python, with file extensions of .m, .pro and .py respectively. These are platform independent languages, and will run on a PC, Mac or Linux machine. The speed of each is dependent both on your machine and which version

of the software you have; use the option that is most convenient to yourself. For each model, a Cartesian (xyz) only and Spherical (rtp) only form is available per language, where both the inputs and outputs are in the same system.

File names specify the planet, the model, the order, Cartesian or Spherical and language: e.g. jovian_jrm33_order13_internal_xyz.m is the Jupiter based jrm33 model to order 13, in Cartesian form, for MATLAB.

In order to standardize these many files, a separate code was written that loops through and writes out the MATLAB, IDL and Python (3) codes for each model and xyz or rtp form. Essentially each language's code is a direct line for line translation of each other. Each code has comment headers that state the model, order, reference and specific g and h coefficients assumed (these are processed for later inclusion in the code in order to speed it up).

The Python models code's first line imports the NumPy library (Harris et al. 2020), but that is the only required Python library that must be locally installed (and often is pre-installed with Python).

Each code takes an input position vector as 3 separate variables, that are ideally doubles (but singles or integers are also accepted, or NumPy arrays for Python). These 3 variables may be scalar, or 1-dimensional arrays. In the case of MATLAB 1-dimensional arrays, it must be size n by 1, and not size 1 by n. While the code could simply check for that and transpose if necessary, that would slow it down, so it is faster to get the user to stick to the native 1-dimensional form in MATLAB.

Output will be a vector of size n by 3, where $n = 1$ if inputs were scalar, and is always of type double.

3.4.2 Install

Each code stands alone without any dependencies; save it to your local working directory and use. For Python, one must import each code prior to use, as is standard Python practice (see example in next section).

3.4.3 Examples

Figure 2 shows screens shots in the three languages for three quick tests (on the left) and their output (on the right). Aside from the difference in the native number of printed decimal places of each language, the values are identical. There is row- vs. column-major differences in languages (e.g. you'd have to transpose the IDL output to get it visually in the same form as the MATLAB or Python output), however, the order of dimensions in the output array is the same in all languages, n by 3.

3.5 Computational Speed Tests

Figure 3 shows the results from example speed tests, which should only be judged on trends. Actual speeds depend on the user's specific computer (e.g. CPU and memory available, is the user using a laptop in battery saving profile, etc.), and what other programs the user may be running. In addition the running speed will depend on which version of MATLAB, IDL, Python and our community codes the user is utilizing. Furthermore, the higher order codes will take longer to run, since the higher orders require more loops within the code. For this test we used 75,641 different positions (simulating a Juno orbit from apojove ($\approx$112 R_J) to apojove (for the Juno prime mission), which we used as inputs to the spherical harmonic codes, both in one go as a vector, or as separate runs in a for loop doing one position at a

MATLAB test code

```
function MATLAB_test

% Spherical coordinate example for scalar at 10 Rj, Colatitude on equator
% at East longitude of 38 degrees (converted to radians)
Brtp_scalar = jovian_jrm33_order13_internal_rtp(10, pi/2, 38*pi/180);
Brtp_scalar % print to screen

% Spherical coordinate example.
% 4 Quadrants all on the equator, with increasing r
r = [8;9;10;11];
t = pi/2 *[1;1;1;1];
p = [0; pi/2; pi; 1.5*pi];
Brtp = jovian_jrm33_order13_internal_rtp(r, t, p);
Brtp % print to screen

% Same 4 positions but now in Cartesian
x = [ 8; 0;-10;  0];
y = [ 0;  9;  0;-11];
z = [ 0;  0;  0;  0];
Bxyz = jovian_jrm33_order13_internal_xyz(x, y, z);
Bxyz % print to screen

whos Brtp_scalar Brtp Bxyz
```

MATLAB test Output

```
Brtp_scalar =

  -79.8398  399.4828  -53.4823

Brtp =

 -250.0396  779.3628  -48.0068
   38.3079  551.8268  -92.0848
  152.3795  421.1121   17.0471
  -42.3894  313.6507   55.7882

Bxyz =

 -250.0396  -48.0068 -779.3628
   92.0848   38.3079 -551.8268
 -152.3795  -17.0471 -421.1121
   55.7882   42.3894 -313.6507

  Name              Size             Bytes  Class     Attributes

  Brtp              4x3                 96  double
  Brtp_scalar       1x3                 24  double
  Bxyz              4x3                 96  double
```

IDL test code

```
PRO IDL_Test
  ; Spherical coordinate example for scalar at 10 Rj, Colatitude on equator
  ; at East longitude of 38 degrees (converted to radians)
  Brtp_scalar = jovian_jrm33_order13_internal_rtp(10d, !DPI/2d, 38d*!DPI/180d)
  PRINT,'Brtp_scalar = '
  PRINT, Brtp_scalar

  ; Spherical coordinate example.
  ; 4 Quadrants all on the equator, with increasing r
  r = [     8d,       9d,      10d,       11d]
  t = [!DPI/2d, !DPI/2d, !DPI/2d, !DPI/2d   ]
  p = [     0d, !DPI/2d, !DPI   , !DPI*1.5d]
  Brtp = jovian_jrm33_order13_internal_rtp(r, t, p)
  PRINT,'Brtp = '
  PRINT, Brtp

  ; Same 4 positions but now in Cartesian
  x = [  8d,  0d,-10d,  0d]
  y = [  0d,  9d,  0d,-11d]
  z = [  0d,  0d,  0d,  0d]
  Bxyz = jovian_jrm33_order13_internal_xyz(x, y, z)
  PRINT,'Bxyz = '
  PRINT, Bxyz

  HELP, Brtp_scalar, Brtp, Bxyz
END
```

IDL test Output

```
Brtp_scalar =
      -79.839826
       399.48279
      -53.482325
Brtp =
      -250.03964       38.307890       152.37954      -42.389403
       779.36280       551.82685       421.11209       313.65069
      -48.006775      -92.084823       17.047116       55.788200
Bxyz =
      -250.03964       92.084823      -152.37954       55.788200
      -48.006775       38.307890      -17.047116       42.389403
      -779.36280      -551.82685      -421.11209      -313.65069
BRTP_SCALAR     DOUBLE    = Array[1, 3]
BRTP            DOUBLE    = Array[4, 3]
BXYZ            DOUBLE    = Array[4, 3]
```

Python test code

```
import numpy as np
import jovian_jrm33_order13_internal_rtp as jrm33o13_rtp
import jovian_jrm33_order13_internal_xyz as jrm33o13_xyz

# Spherical coordinate example for scalar at 10 Rj, Colatitude on equator
# at East longitude of 38 degrees (converted to radians)
Brtp_scalar = jrm33o13_rtp.jovian_jrm33_order13_internal_rtp(10, np.pi/2, 38*np.pi/180);
print('Brtp_scalar = ')
print(Brtp_scalar)

# Spherical coordinate example.
# 4 Quadrants all on the equator, with increasing r
# Can be numpy arrays or not, if you use pi, should be
r =          [      8,       9,      10,        11]
t = np.array([np.pi/2, np.pi/2, np.pi/2, np.pi/2  ])
p = np.array([      0, np.pi/2, np.pi  , np.pi*1.5])
Brtp = jrm33o13_rtp.jovian_jrm33_order13_internal_rtp(r, t, p);
print('Brtp = ')
print(Brtp)

# Same 4 positions but now in Cartesian
# Can be numpy arrays, but since just numbers here, does not have to be
x = [  8,  0,-10,  0]
y = [  0,  9,  0,-11]
z = [  0,  0,  0,  0]
Bxyz = jrm33o13_xyz.jovian_jrm33_order13_internal_xyz(x, y, z);
print('Bxyz = ')
print(Bxyz)

print('Shape of Brtp_scalar: ', Brtp_scalar.shape)
print('Shape of Brtp       : ', Brtp.shape)
print('Shape of Bxyz       : ', Bxyz.shape)
```

Python test Output

```
Brtp_scalar =
[-79.8398262  399.48279169 -53.48232536]
Brtp =
[[-250.03964154  779.36280353  -48.0067748 ]
 [  38.30789003  551.82684557  -92.08482301]
 [ 152.37953988  421.11209401   17.04711562]
 [ -42.38940341  313.65069342   55.78819978]]
Bxyz =
[[-250.03964154  -48.0067748  -779.36280353]
 [  92.08482301   38.30789003 -551.82684557]
 [-152.37953988  -17.04711562 -421.11209401]
 [  55.78819978   42.38940341 -313.65069342]]
Shape of Brtp_scalar:  (3,)
Shape of Brtp       :  (4, 3)
Shape of Bxyz       :  (4, 3)
```

Fig. 2 Examples using the `PSH` codes and the output they give, from the three languages

time (i.e. scalar). The top panels show the speed tests when running the Spherical version of the spherical harmonic codes, and the bottom panels the Cartesian versions. The left panels show the speed when all positions were run once as a vector, and the right panels for each position run separately in a 'for' loop. (C++ does not do vectorized math and thus is coded to do one element at a time of an input 1D array, but for comparison with other languages we will term this as vectorized.)

It is immediately clear that running once with vector inputs is much faster than running separately for each position, and this use is encouraged where possible. However, for B-field line tracing purposes, where you do not know your next position until you have calculated

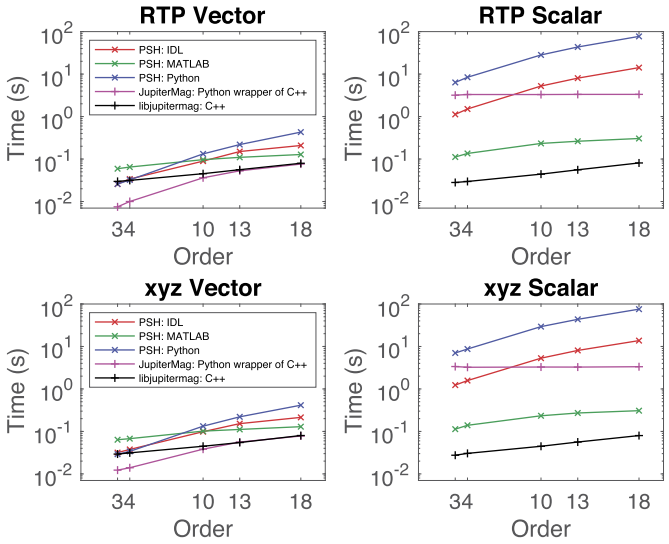

Fig. 3 Speed test of the different Spherical Harmonic codes over 75,641 positions. These were performed on a 2019 Mac laptop with MATLAB version 2022a, IDL version 8.8, and Python 3 version 3.8.3. The top half shows the spherical (RTP) co-ordinate code, and the bottom half the Cartesian (xyz) codes. The left side shows the runs with all 75,641 records run once as vectors, and the right side shows the scalar version where the code is run through all 75,641 positions one at a time, via a for-loop. The test was carried out on 5 models, of order 3, 4, 10, 13 and 18 (O6, VIP4, JRM09, JRM33 order 13 and order 18 respectively)

the field for the last, running in scalar mode is the only option. The spherical code should run a little faster than the Cartesian code (as there are fewer equations to run as spherical harmonics are natively in a spherical coordinate system), however, it is not a significant time saver.

4 External Magnetic Field Models

4.1 Con2020: The Connerney et al. (1981, 2020) Current Sheet Model for Jupiter's External Magnetic Field

The magnetic field observed by a spacecraft in Jupiter's magnetosphere is the sum of the internal planetary field and the external field, which includes contributions from several sources like the magnetodisc current sheet, a partial ring current, and magnetopause currents (e.g. Khurana et al. (2004)).The magnetodisc current flows in the azimuthal direction and perturbs the poloidal magnetic field (see Fig. 1 of Connerney et al. (1981)), producing the radially stretched field configuration that is observed in Jupiter's middle and outer magnetosphere. We have developed code to implement one commonly used model for Jupiter's magnetodisc magnetic field, the Connerney et al. (2020) magnetodisc model, also known as Con2020. It utilized Juno MAG data (Connerney et al. 2017) within 30 R_J of Jupiter from the first 24 Juno orbits, after removing a JRM09 internal field model, to fit the key parameters of the Con2020 model. Only data in MAG instrument range 0 was used, downsampled to a 10 minute cadence, which practically limited the observed dataset to be from distances of 6 to 30 R_J of inbound and outbound passes, within System III (1965) latitudes of -68 to $+40$ degrees. No data close to Jupiter or over the poles was included.

Table 2 Con2020 model code default parameters (true for all codes unless otherwise stated under variable name)

Variable Name	Description	Default Value (and Units)
mu_i_div2__current_parameter_nT	$\mu_0 I_0/2$, azimuthal current sheet field parameter	139.6 nT
i_rho__radial_current_MA	I_ρ, radial current term from Connerney et al. (2020) (set this to zero to turn radial currents off as in Connerney et al. (1981)	16.7 MA
r0__inner_rj	R_0, inner edge of Con2020 current disk	7.8 R_J
r1__outer_rj	R_1, outer edge of Con2020 current disk	51.4 R_J
d__cs_half_thickness_rj	D, current sheet half thickness	3.6 R_J
xt__cs_tilt_degs	ϑ_D, tilt angle of current sheet normal	9.3 degrees
xp__cs_rhs_azimuthal_angle_of_tilt_degs	φ_D, azimuthal angle of the tilt of the current sheet normal sheet tilt	155.8 degrees (right handed)
error_check	1 to check that inputs are valid (Default), or set to 0 to skip input checks (faster) not a Con2020 model parameter	1
CartesianIn *(Python only)*	Input is expected in Cartesian co-ords if True, or Spherical co-ords if False.	True
CartesianOut *(Python only)*	Output B is given in Cartesian co-ords if True, or Spherical co-ords if False.	True

The Con2020 current disc is described by seven key parameters that are defined by keywords in the model code. These are the inner edge of the current sheet R_0, outer edge R_1, half-thickness D, the tilt and azimuthal angles of the surface normal of current sheet, and azimuthal current parameter $\mu_0 I_0$ in units of nT. We note that the current density is given by $j = (I_0/\rho_0)\hat{\phi}$. Connerney et al. (2020) also includes a value for the radial current. Table 2 presents the variable names, description and default values for the seven Con2020 variables as used in our models. The default values are taken from Table 1 of Connerney et al. (2020) for all the constants, except for the radial current. We note that Table 2 of Connerney et al. (2020) gives individual values for PJs 1-24 (excluding PJ 2) (PJ is an abbreviation for perijove). We have used the approximate mean value from Table 2 column 5 to define the default radial current as 16.7 MA. (Due to a typo in Connerney et al. (2020), the fifth column of their Table 2 is erroneously labeled as $\mu_0 I_r/2\pi$.) In our codes each of the default keyword parameters can be changed to, for example, reproduce the Voyager-era CAN disc model of Connerney et al. (1981), or to fit individual Galileo or Juno orbits by changing the azimuthal and/or radial current parameters (see orbit-by-orbit fit values in Vogt et al. (2017); Ridley and Holme (2016); and Connerney et al. (2020)).

The Con2020 current disc and resulting radially stretched field lines are shown in Fig. 4, which presents Jupiter's magnetospheric field lines in a magnetic meridional plane where the field lines correspond to the JRM33 dipole plus the Con2020 magnetodisc model. The dipole is directed along the z axis and cylindrical radius ρ is the perpendicular distance from the magnetic axis. The Connerney et al. (2020) model cylindrical magnetodisc occupies the region within the dotted rectangle centered on the magnetic equator. The magnetodisc is defined by the default parameters presented in Table 2, namely inner radius $R_0 = 7.8\ R_J$, outer radius $R_1 = 51.4\ R_J$, half-thickness $D = 3.6\ R_J$, and current sheet field parameter $\mu_0 I_0/2 = 139.6$ nT. The Connerney et al. (2020) tilt angle of the surface normal of the

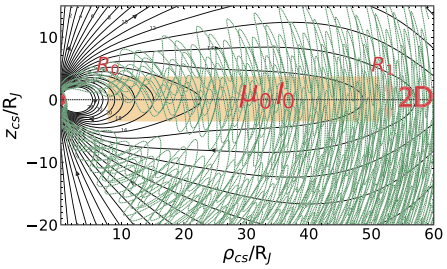

Fig. 4 Illustration of current sheet parameters in the ρ_{cs}-z_{cs} plane showing the inner edge of current sheet R_0, outer edge R_1, half-thickness D, and azimuthal current parameter $\mu_0 I_0$. Also shown (in green) are a selection of orbit trajectories from Juno's prime mission, plotted approximately every third orbit

current sheet to the Jovigraphic pole is $\vartheta_D = 9.3°$ and the azimuthal angle of the current sheet surface normal tilt $\varphi_D = 155.8°$. Overlaid in green in Fig. 4 are a selection of Juno's trajectories from the prime mission where we show approximately every third orbit. We note that for Fig. 4 we are assuming that both the tilt and azimuthal angle of current sheet surface normal are identical to the tilt of the JRM33 dipole from the Jovigraphic pole, so that the combined field is exactly axisymmetric.

4.2 Con2020 Equations

In this section we describe the equations used in our Con2020 model code. The CAN model is described in the current sheet frame of reference, denoted by the subscript $(_{cs})$. The current sheet frame is similar to the magnetic coordinate systems that are defined by the tilt of the dipole in various internal field models. In Con2020 the current sheet normal is tilted 9.3° degrees away from the System III (1965) z-axis toward 155.8° longitude (Right-Handed), whereas the JRM33 dipole is tiled by 10.25° from the System III (1965) rotation axis towards 163° longitude (Right-handed).

Connerney et al. (1981) presents equations for the $B_{\rho_{cs}}$ and $B_{z_{cs}}$ components of the model magnetic field in current sheet cylindrical coordinates (see their equations 14, 15, 17, 18). $B_{\rho_{cs}}$ and $B_{z_{cs}}$ are calculated by taking the integral of the product of two Bessel functions, using different equations for heights z_{cs} larger or smaller than the current sheet thickness D. Connerney et al. (1981) also present analytic equations (see their eqs. A1, A2, A7, A8, and A9) for $B_{\rho_{cs}}$ and $B_{z_{cs}}$ which approximate the full integral equations. However, the Connerney et al. (1981) analytic equations are not divergence-free, which led Edwards et al. (2001) to derive new analytic equations for the Connerney magnetodisc field (see their Eqs. 9a, 9b, 13a, and 13b). Using the Edwards et al. (2001) analytic equations instead of the full integral form introduces an error that is typically a few percent and is largest near the inner edge of the disk (see Fig. 5 here, and Fig. 2 of Edwards et al. (2001)). The azimuthal component of the magnetic field, $B_{\varphi_{cs}}$, which is included only in the 2020 model, can be derived from Ampère's law, as we describe below.

In the community code we present three different equation options: the full 'integral' equations of Connerney et al. (1981), the 'analytic' equations of Edwards et al. (2001), or a 'hybrid' option which uses the Edwards et al. (2001) analytic equations everywhere except at locations close to the disc, defined in cylindrical coordinates as $((\|z_{cs}\| < 1.5D)$ AND $(\|\rho_{cs} - R_0\| < 2R_J))$, where the code uses the Connerney et al. (1981) integrals. In addition, if the 'integral' method is chosen (or 'hybrid' but the position is in the integral method range), then for computational speed, our codes currently use the integral equations only for the inner edge of the current disk, and the analytical equations for the outer edge. This is a reasonable approximation since the outer edge is usually set to $> 50\ R_J$ and we recommend only using the Con2020 model at distances $< 30\ R_J$.

Our model Con2020 code takes input positions and outputs field components in the System III (1965) coordinate system (right-handed). It then rotates the input positions into a current sheet frame, calculates the Con2020 model field according to the equations described in the following sections, then rotates the field back to System III (1965). We do not provide the equations needed to perform the coordinate transformations here, but an example of the equations used to rotate from magnetic coordinates - or, in this case, current sheet coordinates - to System III (1965) coordinates is given in equations A10-A12 of Vogt et al. (2022).

4.2.1 Poloidal Field - Integral Solution

The integral equations for $B_{\rho_{cs}}$ and $B_{z_{cs}}$, the radial and axial components of the magnetic field in cylindrical coordinates, produced by a semi-infinite current disc starting at an inner edge a and extending to infinity, with a half-thickness D, are given by Connerney et al. (1981), equations 14, 15, 17, and 18 as follows.

For $\|z_{cs}\| \geq D$:

$$B_{\rho_{cs}}(\rho_{cs}, z_{cs}) = (\mathrm{sgn}(z_{cs}))\,\mu_0 I_0 \int_0^\infty J_1(\lambda\rho_{cs})\,J_0(\lambda a)\sinh(\lambda D)\,e^{-\lambda\|z_{cs}\|}\frac{d\lambda}{\lambda} \tag{8}$$

$$B_{z_{cs}}(\rho_{cs}, z_{cs}) = \mu_0 I_0 \int_0^\infty J_0(\lambda\rho_{cs})\,J_0(\lambda a)\sinh(\lambda D)\,e^{-\lambda\|z_{cs}\|}\frac{d\lambda}{\lambda} \tag{9}$$

For $\|z_{cs}\| < D$:

$$B_{\rho_{cs}}(\rho_{cs}, z_{cs}) = \mu_0 I_0 \int_0^\infty J_1(\lambda\rho_{cs})\,J_0(\lambda a)\sinh(\lambda z_{cs})e^{-\lambda D}\frac{d\lambda}{\lambda} \tag{10}$$

$$B_{z_{cs}}(\rho_{cs}, z_{cs}) = \mu_0 I_0 \int_0^\infty J_0(\lambda\rho_{cs})\,J_0(\lambda a)\left(1 - e^{-\lambda D}\cosh(\lambda z_{cs})\right)\frac{d\lambda}{\lambda} \tag{11}$$

where J_0 and J_1 are Bessel functions of zeroth and first order, respectively.

All versions of the Con2020 community code use the same approach to the numeric integration, performing the integral using the trapezoid rule and evaluating the Bessel functions using built-in functions or functions from standard libraries. We tested different values for the integral step size $d\lambda$ and the upper limit of integration λ_{max} and selected values that lead to an efficient but accurate numerical integration. For example, we performed the numerical integration at a wide range of input positions (ρ_{cs}, z_{cs}) and found that decreasing $d\lambda$ by a factor of 10 changed the output of our numerical integral by no more than one tenth of a percent. For calculating $B_{\rho_{cs}}$ we set $d\lambda = 0.0001$ and for calculating $B_{z_{cs}}$ we set $d\lambda = 0.00005$. For both $B_{\rho_{cs}}$ and $B_{z_{cs}}$ the quantity to be integrated depends on the input position, as illustrated in Fig. 5, so λ_{max} varies from 4 to 100 for $B_{\rho_{cs}}$ and from 20 to 100 for $B_{z_{cs}}$ depending on the input z_{cs}. Specifically, λ_{max} is largest for input z_{cs} close to the current sheet half-thickness D. For $B_{\rho_{cs}}$, λ_{max} is smallest when z_{cs} is small (top left of Fig. 5), and for $B_{z_{cs}}$, λ_{max} is smallest when z_{cs} is larger than D (bottom right of Fig. 5).

4.2.2 Poloidal Field - Analytic Approximation

The analytic approximation for the $B_{\rho_{cs}}$ and $B_{z_{cs}}$ components of the magnetic field produced by a semi-infinite current disc of inner edge a and half-thickness D are given by Edwards et al. (2001) as follows (see their equations 9a, 9b, 13a, and 13b):

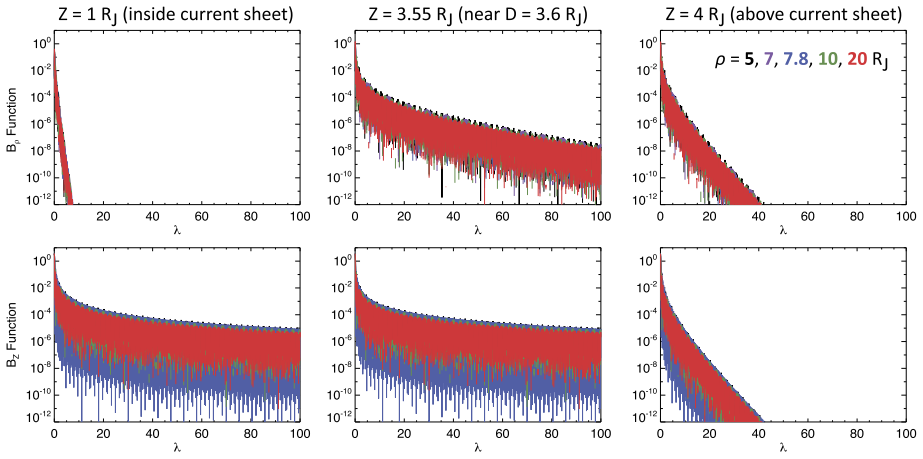

Fig. 5 Examples of the quantities to be numerically integrated by our Con2020 community code, plotted as a function of the variable of integration λ (see eqs. (8) through (11)). Top: $B_{\rho_{cs}}$ integral function as a function of λ for values of z_{cs} inside the current sheet (left), near the current sheet edge at $z_{cs} \sim D$ (middle), and above the current sheet (right). Bottom: $B_{z_{cs}}$ integral function as a function of λ for values of z_{cs} inside the current sheet (left), near the current sheet edge at $z_{cs} \sim D$ (middle), and above the current sheet (right). In all panels, colored lines indicate different values of ρ_{cs}

Small ρ approximation for $\rho_{cs} < R_0$ (CAN1), set $a = R_0$:

$$B_{\rho_{cs}\mathrm{CAN1}}(\rho_{cs}, z_{cs}) \approx \frac{\mu_0 I_0}{2}\left\{\frac{\rho_{cs}}{2}\left[\frac{1}{\sqrt{(z_{cs}-D)^2+a^2}} - \frac{1}{\sqrt{(z_{cs}+D)^2+a^2}}\right]\right.$$
$$\left. + \frac{\rho_{cs}^3}{16}\left[\frac{\left(a^2-2(z_{cs}-D)^2\right)}{\left(a^2+(z_{cs}-D)^2\right)^{\frac{5}{2}}} - \frac{\left(a^2-2(z_{cs}+D)^2\right)}{\left(a^2+(z_{cs}+D)^2\right)^{\frac{5}{2}}}\right]\right\} \quad (12)$$

$$B_{z_{cs}\mathrm{CAN1}}(\rho_{cs}, z_{cs}) \approx \frac{\mu_0 I_0}{2}\left\{\log\left[\frac{(z_{cs}+D)+\sqrt{(z_{cs}+D)^2+a^2}}{(z_{cs}-D)+\sqrt{(z_{cs}-D)^2+a^2}}\right]\right.$$
$$\left. + \frac{\rho_{cs}^2}{4}\left[\frac{(z_{cs}+D)}{((z_{cs}+D)^2+a^2)^{\frac{3}{2}}} - \frac{(z_{cs}-D)}{((z_{cs}-D)^2+a^2)^{\frac{3}{2}}}\right]\right\} \quad (13)$$

Large ρ approximation for $\rho_{cs} \geq R_0$ (CAN2), set $a = R_0$:

$$B_{\rho_{cs}\mathrm{CAN2}}(\rho_{cs}, z_{cs}) \approx \frac{\mu_0 I_0}{2}\left\{\frac{1}{\rho_{cs}}\left(\sqrt{(z_{cs}-D)^2+\rho_{cs}^2} - \sqrt{(z_{cs}+D)^2+\rho_{cs}^2}\right)\right.$$
$$\left. + \frac{\rho_{cs} a^2}{4}\left[\frac{1}{((z_{cs}+D)^2+\rho_{cs}^2)^{\frac{3}{2}}} - \frac{1}{((z_{cs}-D)^2+\rho_{cs}^2)^{\frac{3}{2}}}\right]\right\} \quad (14)$$
$$+ \frac{\mu_0 I_0}{\rho_{cs}} \times \begin{cases} D & , \text{for } z_{cs} \geq D \\ z_{cs} & , \text{for } \|z_{cs}\| < D \\ -D & , \text{for } z_{cs} \leq -D \end{cases}$$

$$B_{z_{cs}\mathrm{CAN2}}\left(\rho_{cs}, z_{cs}\right) \approx \frac{\mu_0 I_0}{2}\left\{\log\left[\frac{(z_{cs}+D)+\sqrt{(z_{cs}+D)^2+{\rho_{cs}}^2}}{(z_{cs}-D)+\sqrt{(z_{cs}-D)^2+{\rho_{cs}}^2}}\right] + \frac{a^2}{4}\left[\frac{(z_{cs}+D)}{((z_{cs}+D)^2+{\rho_{cs}}^2)^{\frac{3}{2}}} - \frac{(z_{cs}-D)}{((z_{cs}-D)^2+{\rho_{cs}}^2)^{\frac{3}{2}}}\right]\right\} \quad (15)$$

$B_{z_{cs}}$ is continuous between region 1 and region 2.

As a slight improvement on Edwards et al. (2001), the CAN poloidal component for both region 1 and region 2 within the semi-infinite current disc can then be calculated as follows, set $a = R_0$:

$$B_{\rho_{cs}\mathrm{CAN}}\left(\rho_{cs}, z_{cs}\right) = B_{\rho_{cs}\mathrm{CAN1}}\left(\frac{1-\tanh\left(\frac{\rho_{cs}-a}{\triangle\rho_{cs}}\right)}{2}\right) + B_{\rho_{cs}\mathrm{CAN2}}\left(\frac{1+\tanh\left(\frac{\rho_{cs}-a}{\triangle\rho_{cs}}\right)}{2}\right) \quad (16)$$

$$B_{z_{cs}\mathrm{CAN}}\left(\rho_{cs}, z_{cs}\right) = B_{z_{cs}\mathrm{CAN1}}\left(\frac{1-\tanh\left(\frac{\rho_{cs}-a}{\triangle\rho_{cs}}\right)}{2}\right) + B_{z_{cs}\mathrm{CAN2}}\left(\frac{1+\tanh\left(\frac{\rho_{cs}-a}{\triangle\rho_{cs}}\right)}{2}\right) \quad (17)$$

where $\triangle\rho_{cs} = 0.1\ R_J$ is the scale length used to smooth the transition between regions.

4.2.3 Accounting for the Finite Nature of the Current Sheet

The CAN current disc can be described as a finite current disc with inner edge R_0 and outer edge R_1, but the equations listed in Sects. 4.2.1 and 4.2.2 give the field produced by a semi-infinite current disc extending from $\rho = a$ to $\rho = \infty$. One less than ideal approach is to account for the finite nature of the current sheet by subtracting a constant value like $\frac{1}{10}\frac{\mu_0 I_0}{2}$ from $B_{z_{cs}}$ while leaving $B_{\rho_{cs}}$ unchanged (see Fig. 4 of Connerney et al. (1981) and appendix of Acuña et al. (1983)).

Another approach to terminating the current at the disc outer edge R_1, which we employ in the Con2020 community code, is to add a second disk of identical current parameter but opposite sign stretching from R_1 to ∞. This gives zero current in total beyond R_1. The total CAN field is then the sum of the field of the semi-infinite current sheet starting at $\rho = R_0$ and the field of the reverse current starting at $\rho = R_1$ as follows:

For $\rho < R_1$:

$$B_{\rho_{cs}\mathrm{CAN}} = B_{\rho_{cs}\mathrm{CAN}}\left(a = R_0\right) - B_{\rho_{cs}\mathrm{CAN1}}\left(a = R_1\right) \quad (18)$$

$$B_{z_{cs}\mathrm{CAN}} = B_{z_{cs}\mathrm{CAN}}\left(a = R_0\right) - B_{z_{cs}\mathrm{CAN1}}\left(a = R_1\right) \quad (19)$$

where $B_{\rho_{cs}\mathrm{CAN}}\left(R_0\right)$ and $B_{z_{cs}\mathrm{CAN}}\left(R_0\right)$ are determined using equations 16 and 17, calculated for $a = R_0$. $B_{\rho_{cs}\mathrm{CAN1}}\left(R_1\right)$ and $B_{z_{cs}\mathrm{CAN1}}\left(R_1\right)$ are the small ρ approximation for $a = R_1$.

For $\rho \geq R_1$, the full reversed equations must be used:

$$B_{\rho_{cs}\mathrm{CAN}} = B_{\rho_{cs}\mathrm{CAN}}\left(a = R_0\right) - B_{\rho_{cs}\mathrm{CAN}}\left(a = R_1\right) \quad (20)$$

$$B_{z_{cs}\mathrm{CAN}} = B_{z_{cs}\mathrm{CAN}}\left(a = R_0\right) - B_{z_{cs}\mathrm{CAN}}\left(a = R_1\right) \quad (21)$$

In our community code we always use the analytic equations of Edwards et al. (2001) to calculate the field due to the semi-infinite disk from R_1 to ∞, even when the user specifies using the integral or hybrid mode to calculate the field due to the inner disk. For the outer semi-infinite disc, the difference between the analytic and integral equations at radial distances $<\sim 30\ R_J$, where the CAN disc model is most valid, will be negligible.

4.2.4 Azimuthal Field

Finally, we derive the expression for the azimuthal field $B_{\phi_{cs}}$, which was omitted from the Voyager-era CAN disc model but introduced by Connerney et al. (2020) through the inclusion of a radial current term I_ρ in the Con2020 model. It is worth emphasizing that the radial current system producing $B_{\phi_{cs}}$ is entirely separate from the disc carrying the azimuthal current resulting in deflection in the $B_{\rho_{cs}}$ and $B_{z_{cs}}$ components. The radial current system consists of equal and opposite currents flowing into the two poles along the polar axis (like 'wires') closed by cylindrical radial currents flowing uniformly in an equatorial disc which here has been chosen to have a half-width D (but could have any value). The azimuthal field resulting from I_{rho} is then calculated from Ampère's law, which in SI units is:

$$\oint \vec{B} \cdot \vec{dl} = \mu_0 I_\rho \tag{22}$$

Therefore, starting with B_ϕ in Teslas (T = Webers/m^2), I_ρ in Amps (A), ρ_{cs} in meters (m), replacing μ_0 ($= 4\pi \times 10^{-7}$ H/m), where units are given in the square brackets, and convert to units of nano-Tesla (nT), Mega-Amps (MA) and jovian planetary radii ($R_J = 7.1492 \times 10^7$ m), it follows that for an outward current:

$$\begin{aligned}
B_\phi[\mathrm{T}] 2\pi \rho_{cs}[\mathrm{m}] &= 4\pi \times 10^{-7} I_\rho[\mathrm{A}] \\
\implies B_\phi[\mathrm{nT}] \times 10^{-9} &= \frac{4\pi}{2\pi} \times 10^{-7} I_\rho[\mathrm{A}]/\rho_{cs}[\mathrm{m}] \\
\implies B_\phi[\mathrm{nT}] &= 2 \times 10^2 \times I_\rho[\mathrm{MA}] \times 10^6 / \left(\rho_{cs}[R_J] \times 7.1492 \times 10^7\right) \qquad (23) \\
\implies B_\phi[\mathrm{nT}] &= \frac{20}{7.1492} \times I_\rho[\mathrm{MA}]/\rho_{cs}[R_J] \\
\implies B_\phi[\mathrm{nT}] &\approx 2.7975 \times I_\rho[\mathrm{MA}]/\rho_{cs}[R_J] \qquad (24)
\end{aligned}$$

Equation (24) ends with an approximation for Jupiter's azimuthal field, as it truncates the numbers to five significant figures; this is the equation used in this release of codes. Should one want to apply this equation to a different planet, use equation (23) and replace 7.1492×10^7 m with the appropriate planetary radius.

Equations (23) and (24) both have a divide by ρ_{cs} which would give infinity if $\rho_{cs} = 0$. If $\rho_{cs} = 0$ then we assume $B_\phi[\mathrm{nT}] = 0$ too. However, B_ϕ will still explode to very large values as ρ_{cs} approaches 0.

Considering how the azimuthal field changes inside the magnetodisc with half-thickness D, it is thus assumed (where z_{cs} and D are in the same units of R_J):

$$B_\phi[\mathrm{nT}] = \begin{cases} -2.79752 \left(I_\rho[\mathrm{MA}]/\rho_{cs}[R_J]\right) \frac{z_{cs}}{\|z_{cs}\|} & , \text{if } \|z_{cs}\| \geq D \text{ and } \rho_{cs} > 0 \\ -2.79752 \left(I_\rho[\mathrm{MA}]/\rho_{cs}[R_J]\right) \frac{z_{cs}}{D} & , \text{if } \|z_{cs}\| < D \text{ and } \rho_{cs} > 0 \\ 0 & , \text{if } \rho_{cs} = 0. \end{cases} \tag{25}$$

Overall, the CAN current disc model does a good job of reproducing the external magnetic field in the inner and middle magnetosphere (inside of $\sim 30\ R_J$) and is relatively straightforward to implement. However, it does not include any local time asymmetries, which can be significant beyond $\sim 20 - 30 R_J$ (e.g. Palmaerts et al. (2017), and references therein) or include features like hinging that can be significant in the middle-to-outer mag-

netosphere (Khurana 1992). The Connerney et al. (2020) parameters were fitted using data at distances of 6 to 30 R_J and the original Connerney et al. (1981) model also focused on data inside of 30 R_J. The model field also includes sharp gradients at the edge of the current disc and, in the analytic equations, certain model parameters can produce a discontinuity or other spurious results at large radial distances. Additionally, the finite nature of the current sheet leads to a discontinuity at the outer edge of the disk R_1 (see Sect. 4.2.3) if the analytical method was used; there is no discontinuity if the integral method was applied.

4.3 Con2020 Python and C++ Code: Con2020_Python, JupiterMag and libjupitermag

There are currently two Python code versions for the Con2020 model: `Con2020_Python` is a pure Python package, whereas `JupiterMag` is a Python wrapper for the `libjupitermag` C++ source code.

Instructions and examples on how to use the `Con2020_Python` model can be found in the GitHub repository (https://github.com/gabbyprovan/con2020). The easiest and quickest way to install this Python package is to use Python's PIP command:

```
$ pip3 install --user con2020
```

or if you have previously installed using this method

```
$ pip3 install --upgrade --user con2020
```

Alternatively, the `JupiterMag` package also includes a Con2020 magnetic field model. See the earlier sections on `libjupitermag` (Sect. 3.3.1) and `JupiterMag` (Sect. 3.3.2) for install instructions.

4.3.1 Examples

Here we show a brief example of how to use `Con2020_Python` module:

```
import con2020

# initialize a Con2020 model object with default
# parameters (inc. Cartesian in/out)
def_model = con2020.Model()

# initialize a Con2020 model object which uses spherical polar
# coordinates for input and output
sph_model = con2020.Model(CartesianIn=False,CartesianOut=False)

# initialize a Con202 model object with custom parameters (longhand)
custom_model = con2020.Model(mu_i_div2__current_parameter_nT=150.0,
                             r0__inner_rj=9.5,
                             d__cs_half_thickness_rj=3.1)

# Run a Con2020 model you've already initialized,
# assuming you already have set up x, y and z as scalars or NumPy arrays
# and x,y,z are System III (1965) right handed co-ordinates
# in units of planetary radii
Bxyz = def_model.Field(x,y,z)
```

The resultant output is a 2-dimensional array, Bxyz, of size n by 3, where n is the length of the input 1D arrays of x, y and z.

If you wish to use the `JupiterMag` package, one can configure and use the Con2020 magnetodisc magnetic fields in the following manner (as described in the GitHub (https://github.com/mattkjames7/JupiterMag)):

```
import JupiterMag as jm

# initialize Con2020 model which uses Cartesian coordinates for
# input and output, set other options here as you wish
jm.Con2020.Config(equation_type='analytic',CartesianIn=True,CartesianOut=True)

# Run Con2020 model
# assuming you already have set up x, y and z as scalars or NumPy arrays
# and x,y,z are System III (1965) right handed co-ordinates
# in units of planetary radii
Bx, By, Bz = jm.Con2020.Field(x,y,z)
```

The resultant output is three 1 dimensional arrays, Bx, By and Bz, each the same size as the input arrays. Note that the output is of a different format to the single 2 dimensional array of `Con2020_Python`, but the values within are equivalent.

Below is an example of using the C++ code directly to obtain field vectors from the con2020 model:

```
#include <stdio.h>
#include <jupitermag.h>

int main() {
    /* create instance of the Con2020 class */
    Con2020 model;

    /* variables to store position and magnetic field */
    double x = 20.0;
    double y = 10.0;
    double z =  5.0;
    double Bx, By, Bz;

    /* obtain the field vector at that position */
    model.Field(x,y,z,&Bx,&By,&Bz);

    printf("B=[%5.1f,%5.1f,%5.1f] at [%4.1f,%4.1f,%4.1f]\n",
            Bx,By,Bz,x,y,z);
}
```

which creates an instance of the `Con2020` class with default model parameters (see Table 2). This class object uses member functions similar to those used in the spherical harmonic example in Sect. 3.3.3 to configure the model (see `jupitermag.h` for more information) and to provide the B-field vectors.

Figure 6 is based on Fig. 2 of Connerney et al. (2020), illustrating a comparison of the residual magnetic field (after removal of the internal field using the JRM09 magnetic field model) and the modeled magnetodisc field for PJ16 ($r < 40 R_J$). From the top to the bottom we plot the azimuthal, theta and radial components of the field. The integral solution is shown in red and the analytical solution in light blue. The shaded region represents a strong field region (>1600 nT) dominated by the planetary field and excluded by Connerney et al. (2020) from consideration.

4.4 Con2020 Codes

The `Con2020` codes are available in both IDL and MATLAB versions, and their use is described in the following sub-sections.

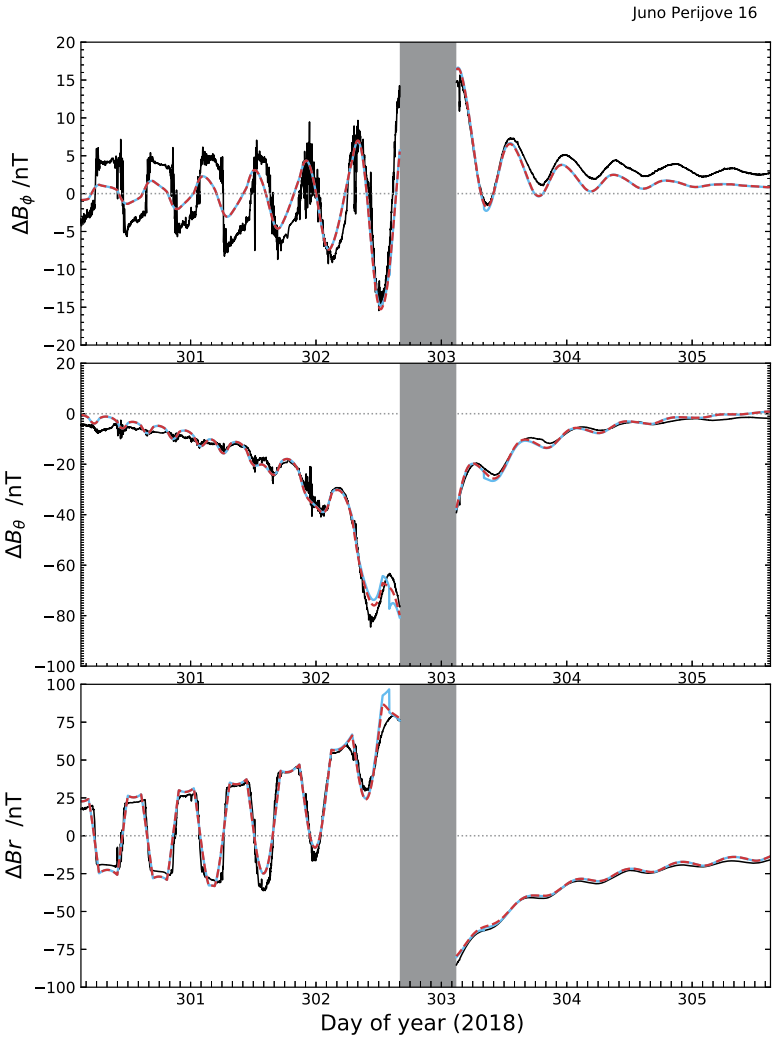

Fig. 6 Comparing observed data (black) with the Con2020 model (after removing the JRM09 internal field), showing both the integral (red) and the analytical (light blue) Con2020 model solutions

4.4.1 Notes

The base Con2020 model is in Cartesian form, therefore the Cartesian code should be ever so slightly faster to run than the Spherical code.

4.4.2 Install

For IDL or MATLAB, save both the Cartesian and Spherical codes to your local working directory and use as normal. The Cartesian version is standalone code with no dependencies. However, the Spherical code, at least in this initial release, calls the Cartesian code, so requires both codes to be on your computer.

MATLAB test code

```
function  Matlab_Con2020_test
%% Default Parameters
% Spherical coordinate example for Con2020 model, Integral method,
% for scalar at 10 Rj, Colatitude on equator at East longitude of
% 38 degrees (converted to radians)
Brtp_scalar = con2020_model_rtp('integral', 10, pi/2, 38*pi/180);
Brtp_scalar % print to screen

% Use a different equation type (put in call later)
eq_type = 'hybrid';

% Spherical coordinate example.
% 4 Quadrants all on the equator, with increasing r
r = [8;9;10;11];
t = pi/2 *[1;1;1;1];
p = [0; pi/2; pi; 1.5*pi];
Brtp = con2020_model_rtp(eq_type,r, t, p);
Brtp % print to screen

% Same 4 positions but now in Cartesian
x = [  8;  0;-10;  0];
y = [  0;  9;  0;-11];
z = [  0;  0;  0;  0];
Bxyz = con2020_model_xyz(eq_type,r, t, p);
Bxyz % print to screen

whos Brtp_scalar Brtp Bxyz

%% Now With User Edited Parameters

% Get structure of parameters, either of these two commands:
% params = con2020_model_rtp('default_values');   or
params = con2020_model_xyz('default_values');
params % print to screen

% Change a value, let's make i_rho__* = 100 (for fun)
params.i_rho__radial_current_MA = 100;
% Change others as you see fit

% Use new parameters
Brtp = con2020_model_rtp(eq_type,r, t, p, params);
Brtp % print to screen
% Or
Bxyz = con2020_model_xyz(eq_type,r, t, p, params);
Bxyz % print to screen
```

MATLAB test Output

```
Brtp_scalar =
  -17.3336  -93.0230   14.4660

Brtp =
  -32.0682 -129.0857   10.6036
   14.7991 -109.8230   15.5326
   35.6280  -86.4414   -7.7410
  -16.2343  -78.2079  -10.8104

Bxyz =
  -30.8765    7.8351  125.7618
  -11.5939    7.3293  106.0843
   13.8752    7.1152   86.5174
   37.3524    6.7266   75.0963

  Name             Size             Bytes  Class     Attributes

  Brtp             4x3                 96  double
  Brtp_scalar      1x3                 24  double
  Bxyz             4x3                 96  double

Returning structure of Default terms used in code.

params =
  struct with fields:
            mu_i_div2__current_parameter_nT: 139.6000
                                  r0__inner_rj: 7.8000
                                  r1__outer_rj: 51.4000
                        d__cs_half_thickness_rj: 3.6000
                                 xt__cs_tilt_degs: 9.3000
    xp__cs_rhs_azimuthal_angle_of_tilt_degs: 155.8000
                        i_rho__radial_current_MA: 16.7000
                                  error_check: 1

Brtp =
  -32.0682 -128.4396   20.2288
   14.7991 -110.4579   11.2822
   35.6280  -85.7953  -17.3662
  -16.2343  -78.8427   -6.5600
Bxyz =
  -32.5284   16.2479  124.9504
  -11.2366    5.0777  106.2888
   15.2499   -3.4681   87.4331
   39.2005  -10.5152   76.5297
```

IDL test code

```
PRO IDL_Con2020_Test
  ;; Default Parameters
  ; Spherical coordinate example for Con2020 model, Integral method,
  ; for scalar at 10 Rj, Colatitude on equator at East longitude of
  ; 38 degrees (converted to radians)
  Brtp_scalar = con2020_model_rtp('integral', 10d, !DPI/2d, 38d*!DPI/180d)
  PRINT,'Brtp_scalar = ' & PRINT, Brtp_scalar

  eq_type = 'hybrid' ; Use a different equation type (put in call later)

  ; Spherical coordinate example.
  ; 4 Quadrants all on the equator, with increasing r
  r = [     8d,     9d,    10d,    11d]
  t = [!DPI/2d, !DPI/2d, !DPI/2d, !DPI/2d ]
  p = [     0d, !DPI/2d, !DPI   , !DPI*1.5d]
  Brtp = con2020_model_rtp(eq_type,r, t, p);
  PRINT,'Brtp = ' & PRINT, Brtp

  ; Same 4 positions but now in Cartesian
  x = [  8d,  0d,-10d,  0d]
  y = [  0d,  9d,  0d,-11d]
  z = [  0d,  0d,  0d,  0d]
  Bxyz = con2020_model_xyz(eq_type,r, t, p);
  PRINT,'Bxyz = ' & PRINT, Bxyz

  HELP, Brtp_scalar, Brtp, Bxyz

  ;; Now With User Edited Parameters
  ; Get structure of parameters, either of these two commands:
  ; params = con2020_model_rtp('default_values');   or
  params = con2020_model_xyz('default_values')
  HELP, params ; print to screen

  ; Change a value, let's make i_rho__* = 100 (for fun)
  params.I_RHO__RADIAL_CURRENT_MA = 100d
  ; Change others as you see fit

  ; Use new parameters
  Brtp = con2020_model_rtp(eq_type,r, t, p, params)
  PRINT,'Brtp = ' & PRINT, Brtp
  ; Or
  Bxyz = con2020_model_xyz(eq_type,r, t, p, params)
  PRINT,'Bxyz = ' & PRINT, Bxyz
END
```

IDL test Output

```
Brtp_scalar =
      -17.333640
      -93.023046
       14.466042

Brtp =
      -32.068179       14.799057       35.628013      -16.234303
      -129.08574      -109.82303      -86.441446      -78.207874
       10.603557       15.532628      -7.7409540      -10.810417

Bxyz =
      -30.876548      -11.593885       13.875213       37.352446
       7.8351101       7.3293068       7.1151951       6.7266175
       125.76182       106.08433       86.517361       75.096299

BRTP_SCALAR     DOUBLE    = Array[1, 3]
BRTP            DOUBLE    = Array[4, 3]
BXYZ            DOUBLE    = Array[4, 3]

Returning structure of Default terms used in code.

** Structure <39d8268>, 8 tags, length=64, data length=57, refs=1:
   MU_I_DIV2__CURRENT_PARAMETER_NT
                   DOUBLE           139.60000
   R0__INNER_RJ    DOUBLE           7.8000000
   R1__OUTER_RJ    DOUBLE           51.400000
   D__CS_HALF_THICKNESS_RJ
                   DOUBLE           3.6000000
   XT__CS_TILT_DEGS
                   DOUBLE           9.3000000
   XP__CS_RHS_AZIMUTHAL_ANGLE_OF_TILT_DEGS
                   DOUBLE           155.80000
   I_RHO__RADIAL_CURRENT_MA
                   DOUBLE           16.700000
   ERROR_CHECK     BYTE         1

Brtp =
      -32.068179       14.799057       35.628013      -16.234303
      -128.43962      -110.45789      -85.795331      -78.842738
       20.228763       11.282223      -17.366160      -6.5600129
Bxyz =
      -32.528394      -11.236604       15.249938       39.200490
       16.247895       5.0777283      -3.4681055      -10.515195
       124.95037       106.28884       87.433127       76.529732
```

Fig. 7 Examples using the `Con2020` codes with the output they give, for IDL and MATLAB

4.4.3 Examples

Figure 7 shows screen shots from MATLAB and IDL for three quick tests then a change of parameters (on the left) and their output (on the right). Aside from the difference in the native number of printed decimal places of each language, the values are identical. As with the spherical harmonic codes, there is row- vs. column-major differences in languages (e.g. you'd have to transpose the IDL output to get it visually in the same form as the MATLAB), however, the order of dimensions in the output array is the same in all languages, n by 3.

4.5 Computational Speed Tests

We calculated the times to run the different Con2020 model codes along one Juno orbit (75,641 records, run through a for loop for the scalar runs). MATLAB is typically slowest and Python is typically fastest. The code runs faster with Cartesian inputs/outputs than with spherical inputs/outputs, and the run times are shown in Table 3.

The full integral version is significantly slower than using the analytic equations. In this test hybrid run the positions were only briefly in the region where the code uses the integral equations (~0.3%, see Table 3).

5 Examples of Using Both the Internal and Con2020 Models

To use both the internal field code plus the external Con2020 model, one simply sums the two fields together to get a total magnetic field ($B_{Total} = B_{internal} + B_{external}$). However, ensure the outputs of the internal and external field codes are both in Cartesian or both in Spherical formats.

For instance, in MATLAB or IDL, following on from examples given in Figs. 2 and 7, the magnetic field summation of both the internal and external fields would be something like this:

```
Brtp_internal = jovian_jrm33_order13_internal_rtp(r, t, p);
Brtp_external = con2020_model_rtp(eq_type, r, t, p);
B_Total = Brtp_internal + Brtp_external;
```

For IDL, the example is identical to the above, without the need to end with a semicolon, but it does not matter if you do leave in this IDL comment marker.

For the Python `JupiterMag` code the method is similar. Following on from the earlier Python examples, after first importing the packages and initializing the models, calculate the internal and external fields and sum them:

```
import JupiterMag as jm

# initialize the Internal field and Con2020 model first
# (as per earlier examples)

Bx_int,By_int,Bz_int = jm.Internal.Field(x,y,z)
Bx_ext,By_ext,Bz_ext = jm.Con2020.Field(x,y,z)

Bx_Total = Bx_int + Bx_ext
By_Total = By_int + By_ext
Bz_Total = Bz_int + Bz_ext
```

where `x`, `y` and `z` may be either scalar or arrays of Cartesian System III (1965) position(s) to evaluate the models at. Similarly, in C++:

```
#include <stdio.h>
#include <jupitermag.h>

int main() {

    /* create both model instances */
    InternalModel modelint;
    Con2020       modelext;
```

Table 3 Speed Test of Con2020 Spherical ('rtp') codes, as an indication only. Actual times will depend on your machine, operating system, and which versions of applications and software you have installed; these results were run on a 2019 Mac, with MATLAB 2022a, IDL 8.8.1 and Python 3.8.3

Equation type	Vectorized or Scalar[a] (Scalar = for loop over all)	Number of Integral runs out of 75,641	`Con2020` (MATLAB)	`Con2020` (IDL)	`Con2020_Python` (Python)	`JupiterMag` (Python wrapping `libjupitermag`)	`libjupitermag` (C++)
Analytic	Vectorized	0	0.064 s	0.023 s	0.017 s	0.010 s	0.009 s
Analytic	Scalar	0	0.869 s	1.636 s	8.351 s	2.423 s	0.008 s
Hybrid	Vectorized	221	10.24 s	11.44 s	11.39 s	10.38 s	10.65 s
Hybrid	Scalar	221	20.87 s	20.57 s	20.22 s	12.98 s	10.72 s
Integral[b]	Vectorized	75641	1799 s	1993 s	1916 s	1836 s	1847 s
Integral[b]	Scalar	75641	3965 s	3432 s	1959 s	1839 s	1844 s

[a]Scalar means a for-loop to run the code over each of the 75,641 records separately.

[b]These Integral times are from code versions given in Table 6, we expect the community will help make later versions significantly faster.

```
    /* variables to store position and fields */
    double x = 25.0;
    double y =  0.0;
    double z =  6.0;
    double Bx_int,   By_int,   Bz_int;
    double Bx_ext,   By_ext,   Bz_ext;
    double Bx_Total, By_Total, Bz_Total;

    /* evaluate each model at the given position */
    modelint.Field(x,y,z,&Bx_int,&By_int,&Bz_int);
    modelext.Field(x,y,z,&Bx_ext,&By_ext,&Bz_ext);

    /* calculate the total field */
    Bx_Total = Bx_int + Bx_ext;
    By_Total = By_int + By_ext;
    Bz_Total = Bz_int + Bz_ext;
    printf("B = [%5.1f,%5.1f,%5.1f]\n",Bx_Total,By_Total,Bz_Total);

}
```

which combines the `InternalModel` and `Con2020` classes used in the previous C++ examples.

6 Field Line Tracing

6.1 libjupitermag and JupiterMag Codes

The `libjupitermag` code (and the `JupiterMag` Python module by extension) is capable of field line tracing using any of the internal models, optionally combined with the Con2020 external field model. Tracing is done using the `JupiterMag.TraceField` object which is initialized by supplying the Cartesian System III (1965) starting coordinates for each trace, optionally setting the internal field model and external model(s) to use (using keyword arguments `IntModel` and `ExtModel`, respectively). At the time of writing, only the Con2020 model has been included with `JupiterMag`. Other keyword arguments control the maximum number of steps (`MaxLen`); the initial step size and limits (`InitStep,MinStep,MaxStep`); the direction in which to trace (`TraceDir`); and whether to output progress to the terminal (`Verbose`). The trace always continues until either the current position reaches Jupiter or the number of steps reaches `MaxLen`.

After the tracing is complete, the `TraceField` object stores variables such as coordinates along each field line and planetary footprint, see Table 4 for a list. The `TraceField` object also contains some member functions which can be used for plotting the traces:

- `PlotRhoZ()` plots the traces in the ρ-z plane (where $\rho = \sqrt{x^2 + y^2}$),
- `PlotXZ()` plots the projection of the traces in the x-z plane,
- `PlotXY()` plots the projection of the traces in the x-y plane,
- `PlotPigtail()` shows the footprints of the traces on Jupiter's surface for both hemispheres.

The following example demonstrates how to trace and plot field lines(s) using `JupiterMag` with JRM33 (order 13, the default),

```
import JupiterMag as jm
import numpy as np
import matplotlib.pyplot as plt
```

Table 4 Variables stored within a `TraceField` object

Name	Shape	Description
`x`	`(n,MaxLen)`	x-coordinate along each trace
`y`	`(n,MaxLen)`	y-coordinate along each trace
`z`	`(n,MaxLen)`	z-coordinate along each trace
`Bx`	`(n,MaxLen)`	x-component of the magnetic field along the trace
`By`	`(n,MaxLen)`	y-component of the magnetic field along the trace
`Bz`	`(n,MaxLen)`	z-component of the magnetic field along the trace
`s`	`(n,MaxLen)`	Distance along the trace in R_J
`R`	`(n,MaxLen)`	Radial coordinate of each position along the trace
`Rnorm`	`(n,MaxLen)`	Normalized radial distance along each trace ($R_{norm} = R/R_{max}$)
`nstep`	`(n,)`	Total number of steps in each trace
`LatN`	`(n,)`	Latitudes of the northern hemisphere footprints
`LonN`	`(n,)`	Longitudes of the northern hemisphere footprints
`LatS`	`(n,)`	Latitudes of the southern hemisphere footprints
`LonS`	`(n,)`	Longitudes of the southern hemisphere footprints
`Rmax`	`(n,)`	Farthest distance along each of the field traces
`LonEq`	`(n,)`	Longitudes of the furthest points along each trace
`FlLen`	`(n,)`	Length of the field line in R_J

```
# get some starting coords
n = 8
theta = (180.0 - np.linspace(22.5,35,n))*np.pi/180.0
r = np.ones(n)
x = r*np.sin(theta)
y = np.zeros(n)
z = r*np.cos(theta)

# configure the Con2020 field model
jm.Con2020.Config(equation_type='analytic')

# Create first TraceField object using JRM33, no external model
T0 = jm.TraceField(x,y,z,Verbose=True,IntModel='jrm33',ExtModel='none')

# create second TraceField object using JRM33 and Con2020
T1 = jm.TraceField(x,y,z,Verbose=True,IntModel='jrm33',ExtModel='Con2020')

# plot them
ax = T0.PlotRhoZ(label='JRM33',color='black')
ax = T1.PlotRhoZ(fig=ax,label='JRM33 + Con2020',color='red')
ax.set_xlim(-2.0,25.0)
ax.set_ylim(-10.0,10.0)
plt.show()
```

which would produce Fig. 8 showing a comparison between tracing using only JRM33 (black) and JRM33 combined with Con2020 (red). The same traces can also be performed directly in C++:

```
#include <jupitermag.h>
#define USE_MATH_DEFINES
#include <math.h>
#include <vector>
```

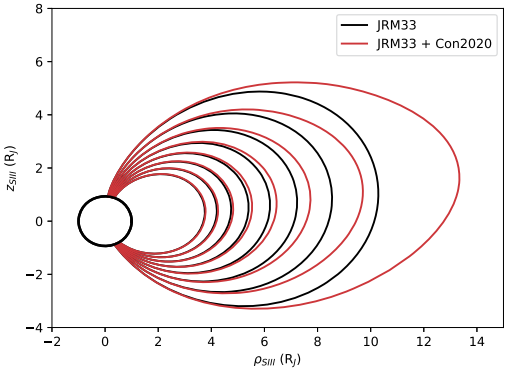

Fig. 8 Comparison of JRM33 order 13 field traces (black) with traces of JRM33 and Con2020 (red)

```
int main() {

    /* set up the starting coordinates */
    int    n = 8;
    double r = 1.0, x[n], y[n], z[n], theta;
    int    i;
    for (i=0;i<n;i++) {
        theta = (180.0 - i*(35.0-22.5)/(n-1))*M_PI/180.0;
        x[i] = r*sin(theta);
        y[i] = 0.0;
        z[i] = r*cos(theta);
    }

    /* obtain the functions to include in the trace,
     * adding their pointers to a vector */
    std::vector<FieldFuncPtr> Funcs;

    /* internal model */
    Funcs.push_back(jrm09Field);

    /* external model */
    Funcs.push_back(Con2020Field);

    /* initialise the trace object */
    printf("Create Trace object\n");
    Trace T(Funcs);

    /* add the starting positions for the traces */
    printf("Add starting position\n");
    T.InputPos(n,x,y,z);

    /* configure the trace parameters (defaults) */
    printf("Set the trace parameters \n");
    T.SetTraceCFG();
```

```
    /* set up the alpha calculation */
    printf("Initialize alpha\n");
    T.SetAlpha(0,NULL);


    /* Trace */
    printf("Trace\n");
    T.TraceField();

    /* trace distance, footprints, Rnorm */
    printf("Footprints etc...\n");
    T.CalculateTraceDist();
    T.CalculateTraceFP();
    T.CalculateTraceRnorm();

    printf("Tracing Complete\n");

}
```

where arrays of trace positions and field vectors are stored in member variables of the `Trace` object (`**x_, **y_, **z_` for position and `**bx_, **by_, **bz_` for magnetic field). For more information, see `jupitermag.h`. In both languages, any number of external field models may be combined as long as they have been compiled into `libjupitermag`. In Python, additional external field models can be included using the `ExtModel` keyword, by providing a list of model names e.g. `ExtModel=['model0', 'model1','model2',..]`. In C++ pointers to model wrapper functions are added to a `std::vector` (called `Funcs` in the example); the trace routine loops through each of the models provided and combines their individual contributions together for each step.

7 Where to Locate Input Positions to Use in These Codes

Generally, data files already exist that contain the position information of the spacecraft. SPICE (Acton 1996) can be used to retrieve these positions, and usually there are position files (or data files with position information included) on NASA's Planetary Data System (PDS, at https://pds.nasa.gov/). Often these are in the System III (1965) values you want, but you may have to tweak them slightly, for instance, converting km to planetary radii.

For example, the Juno MAG (Connerney et al. 2017) data planetocentric files (*_pc_*) on the PDS (Connerney 2017) give System III (1965) Cartesian position in km, so divide by 71492 to convert to jovian radii before using with the Cartesian version of these codes. Whereas the Juno JADE (McComas et al. 2017) Level 3 version 4 data on the PDS (Allegrini et al. 2019) give System III (1965) radial distance R, latitude and east-longitude in units of R_J, degrees and degrees respectively. Therefore use the spherical version of these codes, with:

$$r[R_J] = R[R_J] \tag{26}$$

$$\theta[\text{rads}] = (90 - \{\text{latitude}[^\circ]\})\frac{\pi}{180} \tag{27}$$

$$\phi[\text{rads}] = \{\text{East-Longitude}[^\circ]\}\frac{\pi}{180} \tag{28}$$

8 Future Work and Future Differences

These are intended to be living community codes, we hope the community will expand upon these by adding new internal field models, models for other planets/moons, other field-line tracing routines, and find ways to make the code more efficient.

At the time of publication, the different versions of the spherical harmonic and Con2020 codes (see Table 6) all provide the same output for the same inputs to rounding error levels, which in our testing showed that the different codes all gave the same outputs to within less than 2×10^{-10} nT. All the Con2020 codes use the inbuilt Bessel function commands of their respective language, but speed improvements for running the Integral method of Con2020 could be achieved by replacing the inbuilt Bessel functions with approximate forms of Bessel functions. However, results will not exactly match the output of these early codes. Likewise, if the code is updated for other planets and equation (24) is edited to use 20/7.1492 (for Jupiter) instead of the five significant digit value of 2.7975 currently in the codes (as was in the original donated code), the outputs will not exactly match those of these early codes, but the result will still be valid. i.e. we do not expect future/better versions of these codes to precisely match the outputs of these initial codes, but they should be similar to acceptable levels of rounding.

As this publication and release of initial code was being finalized, Khurana et al. (2022) released their current sheet model, and made their Fortran code available (see Khurana (2022)). We hope the community translates this (and other current sheet models, such as Wang et al. (2022)) into other computer languages to share, but they were simply too late for consideration for this study.

9 Conclusions and Summary

We report here on an international effort to provide high-quality and robust programming code for the Magnetospheres of the Outer Planets community. The code we have presented calculates internal and external magnetic fields of Jupiter, for use in both scientific analysis and mission planning.

The community codes have been placed on GitHub, and also archived and given DOIs via Zenodo (https://zenodo.org/), as given in Table 5. There are multiple DOIs per GitHub Repository: an 'All Versions' DOI that will always take the user to the latest version, and a DOI per each GitHub release. While it is possible the GitHub repositories may be renamed, move or even be deleted over the years, the code will always be available via Zenodo (which also links back to the GitHub repository used, if not deleted) and the DOIs. The current release (and DOIs) of the codes at the time of paper acceptance are shown in Table 6 (Wilson et al. 2022; James et al. 2022b,a; Vogt et al. 2023; Provan et al. 2023).

The spherical harmonic and Con2020 codes are provided in four programming languages (C++, IDL, MATLAB and Python) and is freely available on GitHub. The different codes (of Table 6) provide the same outputs for the same inputs (rounding errors aside, currently under 2×10^{-10} nT), we recommend that the user uses the programming language that they are most familiar with.

In the paper we have described how to install and use the models, and the common pitfalls associated with using them. To briefly summarize, inputs to the model should be in System III (1965) right-handed co-ordinates (with units of planetary radii and, if spherical co-ordinates, radians), as can be provided by the SPICE kernels. To calculate the internal magnetic field for Jupiter, at this time, we suggest using the JRM33 magnetic field model to

Table 5 Where to find these community codes

Spherical Harmonic Codes				
Code Language	Code Name	GitHub or Python Package Index (All Versions)	DOI (All Versions)	Code Type
IDL	`PSH`	https://github.com/rjwilson-LASP/PSH	10.5281/zenodo.6814109	Standalone Files
MATLAB	`PSH`	https://github.com/rjwilson-LASP/PSH	10.5281/zenodo.6814109	Standalone Files
Python	`PSH`	https://github.com/rjwilson-LASP/PSH	10.5281/zenodo.6814109	Standalone Files
C++	`libjupitermag`[a]	https://github.com/mattkjames7/libjupitermag	10.5281/zenodo.7306035	Package
Python wrapper of C++	`JupiterMag`[a]	https://github.com/mattkjames7/JupiterMag https://pypi.org/project/JupiterMag	10.5281/zenodo.6822191	Package

Con2020 Codes				
Code Language	Code Name	GitHub or Python Package Index (All Versions)	DOI (All Versions)	Code Type
IDL	`Con2020`	https://github.com/marissav06/con2020	10.5281/zenodo.6981615	Standalone Files
MATLAB	`Con2020`	https://github.com/marissav06/con2020	10.5281/zenodo.6981615	Standalone Files
Python	`Con2020_Python`	https://github.com/gabbyprovan/con2020 https://pypi.org/project/con2020	10.5281/zenodo.6959770	Package
C++	`libjupitermag`[a]	https://github.com/mattkjames7/libjupitermag	10.5281/zenodo.7306035	Package
Python wrapper of C++	`JupiterMag`[a]	https://github.com/mattkjames7/JupiterMag https://pypi.org/project/JupiterMag	10.5281/zenodo.6822191	Package

[a]The `libjupitermag` and `JupiterMag` packages will do both Spherical Harmonic and Con2020 calculations.

Table 6 Release version (and DOIs) at time of paper publication

Spherical Harmonic Codes			
Code Language	Code Name	GitHub Release	DOI (of GitHub release)
IDL[a]	`PSH`	v1.0.0	10.5281/zenodo.7327992
MATLAB[a]	`PSH`	v1.0.0	10.5281/zenodo.7327992
Python[a]	`PSH`	v1.0.0	10.5281/zenodo.7327992
C++[b]	`libjupitermag`	v1.0.2	10.5281/zenodo.7310141
Python wrapper of C++[c]	`JupiterMag`	v1.0.10	10.5281/zenodo.7374607

Con2020 Codes			
Code Language	Code Name	GitHub Release	DOI (of GitHub release)
IDL[d]	`Con2020`	v1.0.0	10.5281/zenodo.7586161
MATLAB[d]	`Con2020`	v1.0.0	10.5281/zenodo.7586161
Python[e]	`Con2020_Python`	v1.2.1	10.5281/zenodo.7589982
C++[b]	`libjupitermag`	v1.0.2	10.5281/zenodo.7310141
Python wrapper of C++[c]	`JupiterMag`	v1.0.10	10.5281/zenodo.7374607

[a]Wilson et al. (2022).

[b]James et al. (2022b).

[c]James et al. (2022a).

[d]Vogt et al. (2023).

[e]Provan et al. (2023).

order 13. For the external magnetodisc magnetic field we recommend running the 'hybrid' version of the Con2020 model, which uses the analytical equations everywhere except at locations close to the disc, defined in current sheet cylindrical coordinates in units of R_J by the pseudocode:
IF (($\|z_{cs}\| < 1.5D$) AND ($\|\rho_{cs} - R_0\| < 2$)) THEN 'Integral' ELSE 'Analytic'.

The Con2020 model uses Bessel functions for the integral method (and sometimes during the hybrid method) which are computationally slow, which can add hours on to the processing times, and therefore the integral method is only applied to the inner edge of the Con2020 current sheet (the outer edge uses the analytical method even when 'integral' or 'hybrid' methods are chosen). We hope a future technique will be found by the community that will find an efficient way of implementing Bessel functions so that future Con2020 code just uses the integral method all the time, for both the inner and outer edges of the Con2020 current disk, without the need for the hybrid or analytical options.

While these codes could provide output for any input position (although our code has an upper limit of 200 R_J, to error if someone tries to input position in km), they were generally only created using data to several R_J (see Table 1) so should not be trusted for large distances when other external field sources may become significant. We recommend the Con2020 model only be used out to about 30 R_J unless you can justify its use for larger distances. Thus we recommend that any field line tracing is only carried out within 30 R_J using these models. We note that Connerney et al. (2022) used JRM33 order 18 for their field line tracings rather than JRM33 order 13.

We suggest that users double check their inputs are really in a System III (1965) coordinate system (and with 1 $R_J = 71{,}492$ km), and not an earlier System III, or with positions from files generated during 2000 to 2011 when it is possible that SPICE was using different values for their IAU_Jupiter system (in which case, regenerate your positions with the latest SPICE using your record's time stamps).

Should you wish to see the g and h Schmidt coefficients used per internal field code, they are in the comments of the `PSH` codes, and within the source *.dat files within the `JupiterMag` package.

If using these codes, please cite this paper and please be specific on how you used the codes. If using the internal field codes for published work, please cite also the version number and DOI of the software (see Tables 5 and 6), and note which code language was used, which model and to how many orders, e.g. "The MATLAB community code (version 1.0.0, *{citation to DOI}*) was used for the JRM33 order 13 interior field model". Similarly, if using the Con2020 model, please quote the code language, whether ran in integral, hybrid or analytic mode, and if you used the default settings, or if not, what was changed, e.g. "The IDL community code (version 1.0.0, *{citation to DOI}*) was used for the Con2020 external field, ran in hybrid mode with the default parameters." (or "...with default parameters except i_rho_radial_current_MA was set to 0 MA.")

We note here that even though these codes are for the jovian system, the codes can easily be modified for other planets, or indeed moons and exo-planets, by changing the model parameters. Future version releases may include magnetic field models for other planets/moons, or newer magnetic field models that do not exist at this time.

Finally, initial feedback on this coding effort has been very positive. If colleagues continue to value this work, we recommend that the community begins a debate as to how such high-quality code can be developed, maintained, stored and can be made readily available to all.

Acknowledgements We are grateful to Masafumi Imai for sharing his own IDL JRM09 codes with the Juno community and to Krishan Khurana for sharing code years ago, which both formed the seed of the codes presented in this work. We thank Jack Connerney for his insight and useful discussions in clarifying equations and constants used throughout. We thank Fran Bagenal for encouraging and herding the original community efforts here. This was a community effort to write and test the codes, mostly done in spare time, hence largely unfunded. Writing this paper and setting up/documenting the GitHubs and Zenodo repositories was too big for spare time alone, hence partially funded. RJW was supported at the University of Colorado as a part of NASA's Juno mission supported by NASA through contact 699050X with the Southwest Research Institute. MFV was supported by NASA grant 80NSSC20K0559 through the New Frontiers Data Analysis Program. GP and MKJ were funded by STFC grant ST/W00089X/1. AK was supported by an STFC Studentship.

Authors' contributions All authors contributed equally to this work.

Availability of data and materials Not applicable.

Code Availability All codes are publicly available, GitHub URLs and Zenodo DOIs are listed in Table 5.

Declarations

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication All authors gave consent to publish.

Competing Interests The authors declare no competing interests.

References

Acton CH (1996) Ancillary data services of NASA's Navigation and Ancillary Information Facility. Planet Space Sci 44(1):65–70. https://doi.org/10.1016/0032-0633(95)00107-7

Acuña MH, Ness NF (1976) The main magnetic field of Jupiter. J Geophys Res 81(16):2917. https://doi.org/10.1029/JA081i016p02917

Acuña MH, Behannon KW, Connerney JEP (1983) Jupiter's magnetic field and magnetosphere. In: Dessler AJ (ed) Physics of the Jovian magnetosphere. Cambridge University Press, Cambridge, pp 1–50

Alexeev II, Belenkaya ES (2005) Modeling of the Jovian magnetosphere. Ann Geophys 23(3):809–826. https://doi.org/10.5194/angeo-23-809-2005

Allegrini F, Wilson RJ, Ebert RW et al (2019) Juno J/SW Jovian Auroral Distribution Calibrated V1.0, JNO-J/SW-JAD-3-CALIBRATED-V1.0. https://doi.org/10.17189/1519715

Bloxham J, Moore KM, Kulowski L et al (2022) Differential rotation in Jupiter's interior revealed by simultaneous inversion for the magnetic field and zonal flux velocity. J Geophys Res, Planets 127(5):e07138. https://doi.org/10.1029/2021JE007138

Caudal G (1986) A self-consistent model of Jupiter's magnetodisc including the effects of centrifugal force and pressure. J Geophys Res 91(A4):4201–4222. https://doi.org/10.1029/JA091iA04p04201

Connerney JEP (1992) Doing more with Jupiter's magnetic field. In: Planetary radio emissions III, pp 13–33

Connerney JEP (1993) Magnetic fields of the outer planets. J Geophys Res 98(E10):18,659–18,680. https://doi.org/10.1029/93JE00980

Connerney JEP (2007) Planetary magnetism. In: Schubert G (ed) Treatise on geophysics, vol 10. Elsevier, Amsterdam, pp 243–280. https://doi.org/10.1016/B978-044452748-6.00159-0

Connerney JEP (2017) Juno Mag Calibrated Data J V1.0, JNO-J-3-FGM-CAL-V1.0. https://doi.org/10.17189/1519711

Connerney JEP, Acuña MH, Ness NF (1981) Modeling the Jovian current sheet and inner magnetosphere. J Geophys Res 86(A10):8370–8384. https://doi.org/10.1029/JA086iA10p08370

Connerney JEP, Acuña MH, Ness NF (1982) Voyager 1 assessment of Jupiter's planetary magnetic field. J Geophys Res 87(A5):3623–3627. https://doi.org/10.1029/JA087iA05p03623

Connerney JEP, Acuña MH, Ness NF et al (1998) New models of Jupiter's magnetic field constrained by the Io flux tube footprint. J Geophys Res 103(A6):11,929–11,940. https://doi.org/10.1029/97JA03726

Connerney JEP, Benn M, Bjarno JB et al (2017) The Juno magnetic field investigation. Space Sci Rev 213(1–4):39–138. https://doi.org/10.1007/s11214-017-0334-z

Connerney JEP, Kotsiaros S, Oliversen RJ et al (2018) A new model of Jupiter's magnetic field from Juno's first nine orbits. Geophys Res Lett 45(6):2590–2596. https://doi.org/10.1002/2018GL077312

Connerney JEP, Timmins S, Herceg M et al (2020) A Jovian magnetodisc model for the Juno era. J Geophys Res Space Phys 125(10):e28138. https://doi.org/10.1029/2020JA028138

Connerney JEP, Timmins S, Oliversen RJ et al (2022) A new model of Jupiter's magnetic field at the completion of Juno's prime mission. J Geophys Res, Planets 127(2):e07055. https://doi.org/10.1029/2021JE007055

Edwards TM, Bunce EJ, Cowley SWH (2001) A note on the vector potential of Connerney et al.'s model of the equatorial current sheet in Jupiter's magnetosphere. Planet Space Sci 49(10–11):1115–1123. https://doi.org/10.1016/S0032-0633(00)00164-1

Gledhill JA (1967) Magnetosphere of Jupiter. Nature 214(5084):155–156. https://doi.org/10.1038/214155a0

Harris CR, Millman KJ, van der Walt SJ et al (2020) Array programming with NumPy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2

Hess SLG, Bonfond B, Zarka P et al (2011) Model of the Jovian magnetic field topology constrained by the Io auroral emissions. J Geophys Res Space Phys 116(A5):A05217. https://doi.org/10.1029/2010JA016262

Hess SLG, Bonfond B, Bagenal F et al (2017) A model of the Jovian internal field derived from in-situ and auroral constraints. In: Fischer G, Mann G, Panchenko M et al (eds) Planetary radio emissions VIII, pp 157–167. https://doi.org/10.1553/PRE8s157

Imai M (2016) Characteristics of Jovian Low-Frequency Radio Emissions during the Cassini and Voyager Flyby of Jupiter. PhD thesis. https://doi.org/10.14989/doctor.k19504

James MK, Wilson RJ, Vogt MF et al (2022a) Jupitermag. Zenodo. https://doi.org/10.5281/zenodo.7374607

James MK, Wilson RJ, Vogt MF et al (2022b) libjupitermag. Zenodo. https://doi.org/10.5281/zenodo.7310141

Khurana KK (1992) A generalized hinged-magnetodisc model of Jupiter's nightside current sheet. J Geophys Res 97(A5):6269–6276. https://doi.org/10.1029/92JA00169

Khurana KK (1997) Euler potential models of Jupiter's magnetospheric field. J Geophys Res 102(A6):11,295–11,306. https://doi.org/10.1029/97JA00563

Khurana KK (2022) Khurana Jupiter Current Sheet Structure Model 2022. https://doi.org/10.5281/zenodo.6555235

Khurana KK, Schwarzl HK (2005) Global structure of Jupiter's magnetospheric current sheet. J Geophys Res Space Phys 110(A7):A07227. https://doi.org/10.1029/2004JA010757
Khurana KK, Kivelson MG, Vasyliunas VM et al (2004) The configuration of Jupiter's magnetosphere. In: Bagenal F, Dowling TE, McKinnon WB (eds) Jupiter. The planet, satellites and magnetosphere. Cambridge planetary science, vol 1. Cambridge University Press, Cambridge, pp 593–616
Khurana KK, Leinweber HK, Hospodarsky GB et al (2022) Radial and local time variations in the thickness of Jupiter's magnetospheric current sheet. J Geophys Res Space Phys 127(10):e2022JA030664. https://doi.org/10.1029/2022JA030664
McComas DJ, Alexander N, Allegrini F et al (2017) The Jovian auroral distributions experiment (JADE) on the Juno mission to Jupiter. Space Sci Rev 213(1–4):547–643. https://doi.org/10.1007/s11214-013-9990-9
Ness NF, Acuna MH, Lepping RP et al (1979) Jupiter's magnetic tail. Nature 280(5725):799–802. https://doi.org/10.1038/280799a0
Palmaerts B, Vogt MF, Krupp N et al (2017) Dawn-dusk asymmetries in Jupiter's magnetosphere. In: Haaland S, Runov A, Forsyth C (eds) Dawn-dusk asymmetries in planetary plasma environments, pp 307–322. https://doi.org/10.1002/9781119216346.ch24
Pensionerov IA, Alexeev II, Belenkaya ES et al (2019) Model of Jupiter's current sheet with a piecewise current density. J Geophys Res Space Phys 124(3):1843–1854. https://doi.org/10.1029/2018JA026321
Provan G, Wilson RJ, Vogt MF et al (2023) con2020. Zenodo. https://doi.org/10.5281/zenodo.7589982RL
Ridley VA, Holme R (2016) Modeling the Jovian magnetic field and its secular variation using all available magnetic field observations. J Geophys Res, Planets 121(3):309–337. https://doi.org/10.1002/2015JE004951
Russell CT, Dougherty MK (2010) Magnetic fields of the outer planets. Space Sci Rev 152(1–4):251–269. https://doi.org/10.1007/s11214-009-9621-7
Sharan S, Langlais B, Amit H et al (2022) The internal structure and dynamics of Jupiter unveiled by a high-resolution magnetic field and secular variation model. Geophys Res Lett 49(15):e98839. https://doi.org/10.1029/2022GL098839
Smith EJ, Davis JL, Jones DE (1976) Jupiter's magnetic field and magnetosphere. In: Gehrels T, Matthews S (eds) IAU colloq. 30: Jupiter: studies of the interior, atmosphere, magnetosphere and satellites, pp 788–829
Vogt MF, Bunce EJ, Nichols JD et al (2017) Long-term variability of Jupiter's magnetodisk and implications for the aurora. J Geophys Res Space Phys 122(12):12,090–12,110. https://doi.org/10.1002/2017JA024066
Vogt MF, Bagenal F, Bolton SJ (2022) Magnetic field conditions upstream of Ganymede. J Geophys Res Space Phys 127(12):e2022JA030497. https://doi.org/10.1029/2022JA030497
Vogt MF, Wilson RJ, Provan G et al (2023) Con2020 - Current Sheet Model Code. Zenodo. https://doi.org/10.5281/zenodo.7586161
Wang Jz, Huo Zx, Zhang L (2021) A modular model of Jupiter's magnetospheric magnetic field based on Juno data. J Geophys Res Space Phys 126(5):e29085. https://doi.org/10.1029/2020JA029085
Wang Jz, Huo Zx, Zhang L (2022) An empirical model of the current sheet in Jupiter's magnetosphere. Planet Space Sci 211:105395. https://doi.org/10.1016/j.pss.2021.105395
Wilson RJ, Vogt MF, Provan G et al (2022) PSH: Planetary spherical harmonics community code. Zenodo. https://doi.org/10.5281/zenodo.7327992
Winch DE, Ivers DJ, Turner JPR et al (2005) Geomagnetism and Schmidt quasi-normalization. Geophys J Int 160(2):487–504. https://doi.org/10.1111/j.1365-246X.2004.02472.x
Yu ZJ, Leinweber HK, Russell CT (2010) Galileo constraints on the secular variation of the Jovian magnetic field. J Geophys Res, Planets 115(E3):E03002. https://doi.org/10.1029/2009JE003492

Authors and Affiliations

R.J. Wilson[1] · M.F. Vogt[2] · G. Provan[3] · A. Kamran[3] · M.K. James[3] · M. Brennan[4] · S.W.H. Cowley[3]

✉ R.J. Wilson
rob.wilson@lasp.colorado.edu

[1] Laboratory for Atmospheric and Space Physics, University Of Colorado Boulder, Boulder, CO, USA

[2] Center for Space Physics, Boston University, Boston, MA, USA

[3] School of Physics and Astronomy, University of Leicester, Leicester, UK

[4] NASA Jet Propulsion Laboratory, Pasadena, CA, USA