The Astrophysical Journal, 911:45 (25pp), 2021 April 10 https://doi.org/10.3847/1538-4357/abe62f

PINT: A Modern Software Package for Pulsar Timing

Jing Luo[1,2,3], Scott Ransom[4], Paul Demorest[5], Paul S. Ray[6], Anne Archibald[7], Matthew Kerr[6], Ross J. Jennings[8], Matteo Bachetti[9], Rutger van Haasteren[10,18], Chloe A. Champagne[11,19], Jonathan Colen[12], Camryn Phillips[13], Josef Zimmerman[13], Kevin Stovall[5], Michael T. Lam[14,15], and Fredrick A. Jenet[16,17]
[1] University of Texas at San Antonio, San Antonio, TX 78249, USA; luojing1211@gmail.com
[2] Center for Advanced Radio Astronomy, University of Texas Rio Grande Valley, Brownsville, TX 78520, USA
[3] Canadian Institute for Theoretical Astrophysics, University of Toronto, Toronto, ON M5S3H8, Canada
[4] National Radio Astronomy Observatory, 520 Edgemont Road, Charlottesville, VA 22903, USA
[5] National Radio Astronomy Observatory, P.O. Box O, Socorro, NM 87801, USA
[6] U.S. Naval Research Laboratory, Washington, DC 20375-5352, USA
[7] Newcastle University, NE1 7RU, UK
[8] Department of Astronomy, Cornell University, Ithaca, NY 14853, USA
[9] INAF-Osservatorio Astronomico di Cagliari, via della Scienza 5, I-09047 Selargius, Italy
[10] Jet Propulsion Laboratory, California Institute of Technology, Pasadena CA 91109, USA
[11] Vanderbilt University, Nashville, TN 37235, USA
[12] Naval Research Enterprise Internship Program (NREIP), resident at U.S. Naval Research Laboratory, Washington, DC 20375-5352, USA
[13] University of Virginia, Charlottesville, VA 22903, USA
[14] School of Physics and Astronomy, Rochester Institute of Technology, Rochester, NY 14623, USA
[15] Laboratory for Multiwavelength Astrophysics, Rochester Institute of Technology, Rochester, NY 14623, USA
[16] Center for Advanced Radio Astronomy, University of Texas Rio Grande Valley, Brownsville, TX, USA
[17] University of Texas at San Antonio, San Antonio, TX, USA
Received 2020 March 30; revised 2021 February 7; accepted 2021 February 9; published 2021 April 14

Abstract

Over the past few decades, the measurement precision of some pulsar timing experiments has advanced from $\sim$10 μs to $\sim$10 ns, revealing many subtle phenomena. Such high precision demands both careful data handling and sophisticated timing models to avoid systematic error. To achieve these goals, we present PINT (**P**INT **I**s **N**ot **T**EMPO3), a high-precision `Python` pulsar timing data analysis package, which is hosted on GitHub and available on the `Python` Package Index (PyPI) as `pint-pulsar`. PINT is well tested, validated, object oriented, and modular, enabling interactive data analysis and providing an extensible and flexible development platform for timing applications. It utilizes well-debugged public `Python` packages (e.g., the NUMPY and ASTROPY libraries) and modern software development schemes (e.g., version control and efficient development with `git` and GitHub) and a continually expanding test suite for improved reliability, accuracy, and reproducibility. PINT is developed and implemented without referring to, copying, or transcribing the code from other traditional pulsar timing software packages (e.g., TEMPO/TEMPO2) and therefore provides a robust tool for cross-checking timing analyses and simulating pulse arrival times. In this paper, we describe the design, use, and validation of PINT, and we compare timing results between it and TEMPO and TEMPO2.

Unified Astronomy Thesaurus concepts: Millisecond pulsars (1062); Radio pulsars (1353); Pulsar timing method (1305); Pulsars (1306); Astronomy software (1855)

1. Introduction

Since their discovery in 1967 (Hewish et al. 1968), the study of pulsars has yielded major advances in a wide range of physics and astrophysical problems. Pulsars are natural laboratories for studying extreme magnetic fields (Gavriil et al. 2008; Makishima 2016), equations-of-state of dense matter (Demorest et al. 2010; Antoniadis et al. 2013; Cromartie et al. 2020), and theories of gravity (Damour & Taylor 1991; Kramer et al. 2006; Archibald et al. 2018). The most powerful aspect of pulsars is the regularity of their pulses, enabling their use as clocks spread throughout our galaxy. Pulsar timing is the technique by which observed pulse arrival times are compared to predicted arrival times based on a physical model of the pulsar signal and its propagation to the observatory. This technique can be used to study both the pulsar itself as well as the effects of binary companions (where applicable), the interstellar medium (Jones et al. 2017; Donner et al. 2019), and Galactic dynamics (Kiel & Hurley 2009; Verbunt et al. 2017).

Millisecond pulsars (MSPs; Backer et al. 1982) have undergone a period of accretion from a companion star, the end result of which is often a very stable, fast-spinning pulsar (spin period $\lesssim$10 ms). Via the long-term observations of high-quality MSPs, whose pulse arrival times can be measured to better than 1 μs, the pulsar timing technique can achieve the precision required for detecting ultra-low-frequency ($\sim 10^{-9}$ Hz) gravitational waves (Foster & Backer 1990; Taylor et al. 2016), whose realistic astrophysical amplitudes in pulsar timing residuals will be of the order of 10 ns. The North American Nanohertz Observatory for Gravitational Waves (NANOGrav; McLaughlin 2013; Ransom et al. 2019) is an ongoing effort to detect nanohertz frequency gravitational waves by monitoring a set of well-timed MSPs using the 305 m William E. Gordon Telescope (Arecibo) of Arecibo Observatory[20] and the 100 m Robert C. Byrd Green Bank Telescope (GBT) of the Green

[18] Currently employed at Microsoft Corporation.
[19] NREIP Intern at U.S. Naval Research Laboratory.
[20] http://outreach.naic.edu/ao/scientist-user-portal/

Bank Observatory.[21] The international effort of pulsar timing for gravitational waves is under the International Pulsar Timing Array (IPTA; Manchester & IPTA 2013) consortium, comprising NANOGrav, the European Pulsar Timing Array (EPTA; Kramer & Champion 2013), the Parkes Pulsar Timing Array (PPTA; Manchester et al. 2013), and recent efforts started in South Africa (MeerTime; Bailes et al. 2020), India (InPTA; Joshi et al. 2018), and China (CPTA; Lee 2016).

Pulsar timing for gravitational waves requires a good understanding of many astrophysical processes that impact the pulse times of arrival (TOAs), including the pulsar system dynamics (e.g., pulsar spin, pulsar system motion, and proper motion, etc.), solar system dynamics (e.g., motions of Earth and planets), and the effects of the interstellar medium (e.g., dispersion and scintillation). Timing is done for each pulsar by creating a mathematical model for these effects, and then refining this model via fitting to the observed TOAs. For decades, the vast majority of radio pulsar timing has been accomplished using one of two major software packages: TEMPO[22] and TEMPO2[23] (Hobbs et al. 2006).

A robust future detection of gravitational waves using pulsar timing will require results to be verified with independent software packages. However, the underlying TEMPO2 code largely consists of TEMPO Fortran-based algorithms, updated to use C. Due to the similarities in these two codes, it is necessary to develop an independent pulsar timing package for cross-checking. The growth of computational power has allowed for high-level scripting languages, such as `Python`, to become more popular in astronomical applications. `Python` has many advantages including brevity, modularity, excellent documentation, robust testing, ease of code reuse, and a large community developing powerful open-source libraries for a wide range of applications. These features considerably improve the speed of development and the code's extensibility, allowing us to fix many of the limitations of traditional timing software. For instance, in order to add an external high-precision orbit integrator for the pulsar triple system (Ransom et al. 2014) or use a spline-based model to handle timing noise (Dib et al. 2009), it was necessary to circumvent large parts of TEMPO/TEMPO2 or abandon them entirely, while PINT is designed to permit the use of only the relevant parts or easy addition of user-written components. In addition, modern version control and distributed development environments like `git` and GitHub[24] have facilitated community contributions that have greatly increased the pace of development and sped the adoption of these packages by the astronomical community. Motivated by the reasons mentioned above, a new pulsar timing software project, PINT, was launched in 2013 by the NANOGrav collaboration.

The PINT project[25] has developed a TEMPO/TEMPO2-independent `Python` toolkit—the PINT software package—for high-precision pulsar timing analysis to precisions of $\sim$1 ns,[26] and including known physical effects with timing amplitudes of $\sim$1 ns or greater. The PINT software package follows modern software development schemes and practices: object-oriented design, modularized classes and components, a documented programming interface, and an automated test suite that is run after every change. A major feature of the PINT package is the use of well-debugged libraries such as NumPy[27] (Harris et al. 2020), SciPy[28] (Virtanen et al. 2020), and Astropy[29] (Astropy Collaboration et al. 2013; The Astropy Collaboration et al. 2018). Because of their large active user and developer communities, such packages are improved frequently and tested thoroughly. Using such packages improves development and maintenance efficiency. Conversely, a key goal of PINT is that it be usable as a library itself, so key functions from PINT can be used in other pulsar-timing-related applications (for example, correcting light travel time delays in high-energy photon arrival times).

In this paper, we present an overview of the PINT pulsar timing analysis package—the full software documentation is available online.[30] In Section 2, we give a brief background of pulsar timing methodology. We then describe the PINT software package, including its setup, code architecture, and key modules, in Section 3. In Section 4, we present one example of a PINT analysis and compare it with TEMPO/TEMPO2. The tests and maintenance procedures are discussed in Section 5. We also introduce common use cases and their command-line scripts in Section 6.

2. Overview of Pulsar Timing

Pulsar timing refers to the process of unambiguously, and to high precision, accounting for pulse TOAs at a telescope using a relatively simple timing model. Here we give a brief overview of pulsar timing including (i) obtaining TOAs, (ii) modeling the pulse emission and propagation time, (iii) comparing the model to observed data, and (iv) improving the model.

2.1. Measuring TOAs

The key measurement for pulsar timing, a TOA, notionally measures the time when a fiducial point of a pulsar pulse profile reaches an observer. Normally, these measurements are actually made on the coherent average of many pulses, the folded pulse profile, both to increase the signal-to-noise ratio and to mitigate the effects of pulse-to-pulse variations (Lorimer & Kramer 2004; Cordes & Downs 1985). This coherent averaging process, also called "folding," sums the radio power as a function of the pulse phase (see Section 2.2.1 for the definition of "pulse phase"); this is computed from the existing pulsar timing model. In the case of high-energy observations, such as from X-ray or γ-ray observatories, TOAs are not necessarily the focus; individual photon arrival times have their pulse phases computed and can be binned into a pulse profile (Ray et al. 2011) or treated individually (Pletsch & Clark 2015).

Given an observation of a pulsar, one generally compares the folded pulse profile to a known template describing the pulsar's (usually stable) pulse profile. A template-matching algorithm (e.g., Taylor 1992) permits a very accurate computation of a shift, expressed in units of rotational phase from -0.5 to 0.5, of

[21] https://greenbankobservatory.org/telescopes/gbt/
[22] http://tempo.sourceforge.net
[23] https://bitbucket.org/psrsoft/tempo2
[24] https://git-scm.com/, https://github.com/
[25] Available at https://github.com/nanograv/PINT and https://pypi.org/project/pint-pulsar/.
[26] For most machines on which PINT will be run, that $\sim$1 ns level of precision is set by the hardware-supported 80 bit floating-point numbers used for many of the time-based calculations.
[27] http://www.numpy.org/
[28] https://www.scipy.org/
[29] http://www.astropy.org/
[30] https://nanograv-pint.readthedocs.io/en/latest/

the observed pulse compared to the template. Phase zero denotes perfect alignment with the template. This computed phase shift is then used to construct a TOA. This begins with the phase-zero moment (according to the ephemeris used for folding) nearest the middle of the observation span and adjusts that time by the measured phase shift multiplied by the pulse period. The TOA is thus the idealized arrival time of the phase-zero part of the template near the middle of the observation span. The TOA value itself is generally represented as a Modified Julian Day (MJD) in the Coordinated Universal Time (UTC) time system,[31] as recorded by an observatory clock. The TOAs require certain additional data, including the observatory where the TOA was recorded, an estimate for the error in the determination of the TOA, and the radio frequency at which it was recorded. Further information can also be recorded, such as the pulsar name, the signal-to-noise ratio of the measurement, the instrument with which it was recorded, etc.

PINT does not provide functionality for measuring TOAs; that is left for codes specific to particular types of data. Nevertheless, PINT can be used to compute the pulse phases for data folding or other calculations (e.g., photon phases). For instance, PINT has a module to generate and interpolate the coefficients of polynomial approximations of the pulse phase (i.e., polycos).

2.2. Modeling TOAs

In order to understand the physics behind the TOAs, we compare them to a timing model, which is a mathematical description of (i) the rotation of the pulsar and (ii) the propagation of its pulses to the observer. The pulsar rotation is mathematically represented using the rotational phase. The propagation process is modeled in terms of time delays related to the light travel time from the pulsar to the observer. In the following subsections, we describe these two parts in more detail.

2.2.1. Rotational Phase

The rotational phase, often referred to as simply phase, describes a pulsar's rotational status in a reference frame that is comoving with the pulsar. One complete rotation is represented by an increase in phase of 1. As the pulsar rotates, the phase naturally increases and is often written as $N(t)$, the cumulative phase number. In cases where the absolute pulse number is not needed or not available, the integer portion may be ignored, and a wrapping fractional phase ranging from 0 to 1 (or −0.5 to 0.5) is used. There is some arbitrariness in the definition of phase zero; it is usually defined as the zero in the phase of an idealized pulse-profile template; this is frequently chosen to be either the highest point or center of mass of the profile for pulsars whose profile consists of only a single component.

Because pulsars do not rotate at constant pulse frequencies, a Taylor expansion typically describes the rotational phase as

$$N(t) = N_0 + \nu_0(t - t_0) + \frac{1}{2}\dot{\nu}_0(t - t_0)^2 + \frac{1}{6}\ddot{\nu}_0(t - t_0)^3 + \ldots, \quad (1)$$

where N_0 is the phase/pulse number at a reference epoch t_0, ν_0 is the pulse frequency (i.e., the first time derivative of the phase) at t_0, and $\dot{\nu}_0$ and $\ddot{\nu}_0$ are the first and second derivatives of the pulse frequency (e.g., Lorimer & Kramer 2004). More complicated rotational models are possible, for instance, those with glitches (a sudden change in pulse frequency; Manchester & Taylor 1974) and glitch relaxation.

If we choose one pulse's arrival time as our reference time t_0, our model parameters are known exactly, and without noise, then the phase at other pulse arrival times $N(t_{\rm TOA})$ will be an integer value.

Practically, in order to evaluate Equation (1), we must transform our observed TOAs into the pulsar comoving frame. In the next Section 2.2.2, these transformations, including timescale conversions and propagation time modeling, are discussed.

2.2.2. Pulse Delays

The delay portion of the timing model characterizes the total pulse propagation time, determined by a variety of physical processes between the pulsar and the observer. Given the TOA at the observatory, we can compute the pulse emission time via the total delay,

$$t_{\rm e} = t_{\rm obs} - \Delta, \quad (2)$$

where $t_{\rm e}$ is the pulse emission time, $t_{\rm obs}$ is the pulse observation time, and Δ represents the total delay, from a wide variety of causes. The total delay

$$\Delta = \Delta_{\rm A} + \Delta_{\rm R\odot} + \Delta_{\rm E\odot} + \Delta_{\rm S\odot} + \Delta_{\rm SB} + \Delta_{\rm fd} + \Delta_{\rm binary} + \ldots, \quad (3)$$

where we have listed the most common delays in the timing process (e.g., Lorimer & Kramer 2004). The first term $\Delta_{\rm A}$ represents the delay caused by the "hydrostatic" atmospheric effects of topocentric observations, modeled as the product of the delay at zenith (Davis et al. 1985) and an azimuthally symmetric function that maps the delay onto any other position in the sky (Niell 1996). The next three terms, $\Delta_{\rm R\odot}$, $\Delta_{\rm E\odot}$, and $\Delta_{\rm S\odot}$, are the solar system geometric or Rømer delay, Einstein delay (comprised of gravitational redshift and time dilation; Taylor & Weisberg 1989), and solar system Shapiro delay (due to the gravitational perturbation of the light path; Shapiro 1964). Although the Shapiro delay term formally includes contributions from all solar system bodies, we normally only include those from the Sun and major planets (i.e., time delays bigger than 1 ns; Hobbs et al. 2006). The $\Delta_{\rm SB}$ term gives the light travel time from the pulsar system to the solar system. Its initial value, which is a very large quantity, can be absorbed in the phase calculation because a phase is computed relative to a reference epoch (see below). The time-dependent part of this delay due to relative motion is separated into delays that vary due to the transverse and radial motion of the pulsar. The former is modeled as the proper motions via the solar system Rømer delay; however, the radial component effect is generally hard to distinguish from the pulse period derivative. The $\Delta_{\rm fd}$ term includes a variety of radio-frequency-dependent time delays, such as the dispersion delay caused by the ionized interstellar and interplanetary media. The last term, $\Delta_{\rm binary}$, includes the pulsar system's Rømer, Einstein,[32] and Shapiro

[31] This has known problems; see Section 3.1.

[32] This "Einstein delay" is not actually a delay; instead, it is the cumulative effect of gravitational and special-relativistic time dilation on the pulsar. In normal pulsar work, the units of time for the pulsar are rescaled so that the mean time dilation is zero and the "Einstein delay" oscillates around zero.

delays. The pulsar Rømer delay is controlled by the position of the pulsar at the moment of pulse emission, rather than the moment of pulse arrival at the solar system Barycenter. Thus, $\Delta_{\rm binary}$ needs to be evaluated at a time that needs $\Delta_{\rm binary}$ itself as input; older timing models incorporate an approximate solution to this inversion problem in their formulas (Damour & Deruelle 1986), while more modern ones solve it directly by root finding (Ransom et al. 2014). These delay terms' typical range of values are summarized in Hobbs et al.'s (2006) Table 2.

Given the transformation from pulse observed time $t_{\rm obs}$ to pulse emission time (ignoring a constant pulsar system Einstein delay, see footnote 29)

$$N(t_{\rm obs}) = N_0 + \nu_0(t_{\rm obs} - \Delta - t_0) + \frac{1}{2}\dot{\nu}_0(t_{\rm obs} - \Delta - t_0)^2 + \frac{1}{6}\ddot{\nu}_0(t_{\rm obs} - \Delta - t_0)^3 + \ldots. \quad (4)$$

The computed phases are described relative to a reference phase N_0 at the reference time t_0. In practice, N_0 is defined by specifying a moment at which the phase is zero ($N = 0$). This moment is specified in the reference frame by a reference MJD, observatory site, and radio frequency (often denoted by the parameters `TZRMJD`, `TZRSITE`, and `TZRFRQ`), as was done in TEMPO/TEMPO2. `TZRMJD` is treated as a hypothetical arrival time measurement, in the timescale of the observatory clock. To transform that time to other timescales, standard clock corrections need to be applied as per any other TOA (see Section 3.1). The resulting phases are used in the process of refining the timing model. Currently, if `TZRMJD` is not specified, the phase of the first TOA in the TOAs table is defined to be zero.

2.3. Comparing Model with the Data

In order to improve the accuracy of a timing model, the timing residuals, defined as the differences between the observed TOAs and the TOAs predicted by the given timing model, are introduced:

$$R_{\rm time} \equiv t_{\rm obs} - t_{\rm model}. \quad (5)$$

Because of the periodic nature of the pulsar's signal, the residuals thus obtained are known only modulo one rotation of the pulsar—that is, a priori, we do not know the integer number of rotations between two pulse arrival time measurements. In an established pulsar timing program, as described in Section 2.2.1, our estimated model is generally accurate enough that the predicted TOAs will differ from the observed TOAs by less than one pulse period. That is, a sufficiently good model allows us to infer the exact number of rotations between any two observations. When the model is insufficient, perhaps because we are observing a new pulsar, or there has been a long gap in observations, or a glitch has occurred, the uncertain number of rotations between observations can make the task of finding or improving a timing solution a highly discontinuous and difficult optimization problem. Traditionally, this has been addressed by hand, with users introducing turn-number guidance into the TOA data files, iteratively working with larger and larger subsets of observations until a satisfactory "phase-connected" solution has been found. Automated tools for phase connection have been implemented (Freire & Ridolfi 2018). Alternatively, if precise rotation numbers have been inferred for the TOAs, these can be coded into the input files, reducing or removing the discontinuous nature of the fitting problem.

Multiplying the time residual by the pulse frequency, we can write the residuals in terms of phase number:

$$R_{\rm phase} = N(t_{\rm obs}) - N_i(t_{\rm obs}), \quad (6)$$

where N_i is the inferred integer phase number at $t_{\rm obs}$. In terms of the phase residual, the time residual can be also written as

$$R_{\rm time} = R_{\rm phase}/\nu(t_{\rm obs}), \quad (7)$$

where $\nu(t_{\rm obs})$ is the apparent pulsar pulse frequency at $t_{\rm obs}$ in the frame of the observer. Traditionally, ν_0, the pulsar pulse frequency at reference time t_0 has been used for this scaling, and for many pulsars, the error is negligible, but PINT implements this more correct time residual calculation. From the residuals, the current timing model can be updated by using a variety of fitting methods. Because of the issue of phase connection, pulsar timing is generally carried out in an iterative way: an approximate model is successively updated as new data becomes available or as more complex models are applied. In each iteration, the previous postfit timing model is treated as the input model and gets updated by tuning the parameter values or using new models (Lorimer & Kramer 2004).

For traditional gravitational-wave detection projects, the residuals generated by a good deterministic timing model are the starting point of analyses (Detweiler 1979). Hellings & Downs (1983) describe the contribution of an isotropic gravitational-wave background on correlations in the timing residuals from an array of well-timed pulsars, that is, a Pulsar Timing Array (PTA; Sazhin 1978). A main objective for the PINT package is to provide high-quality timing and software tools for this type of analysis. Currently, PINT can be used by NANOGrav's gravitational-wave analysis package, the Enhanced Numerical Toolbox Enabling a Robust PulsaR Inference SuitE (ENTERPRISE).[33] In addition, PINT provides analytical derivatives of the phase with respect to most timing parameters and the capability to use numerical derivatives (i.e., finite differences) for all timing parameters (see Section 2.2). Many gravitational-wave analyses (for example, van Haasteren et al. 2009) are able to use the derivatives of the residuals with respect to the timing model parameters as a more efficient proxy for the full timing model that permits analytical marginalization.

3. PINT

PINT is a `Python` library and a set of executable scripts, compatible with `Python` 3.6 or greater.[34] In this section, we introduce the PINT software package version 0.8.0 and provide code examples. The operational model of PINT is illustrated in Figure 1. In the following subsections, the fundamental assumptions, including the coordinate definitions and the

[33] https://github.com/nanograv/enterprise

[34] Support for `Python` 2.7 was dropped in 2020, in conjunction with many other astronomical `Python` packages (see http://python3statement.org).

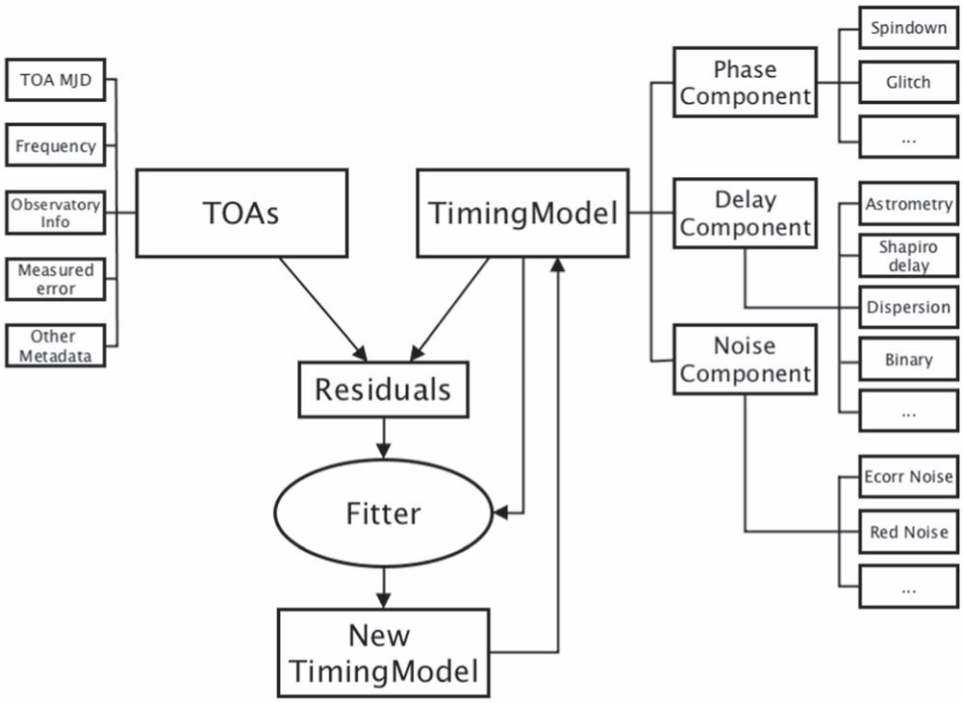

Figure 1. PINT operational model. This is a rough model as to how PINT is designed and implemented as well as how it is used for timing a pulsar. Lines without arrows indicate that the object in question contains the data; arrows indicate that results computed in one object are passed to the other. The TOAs and timing models are kept as independent as possible and only interact through other parts of PINT functionality, such as creating residuals and fitting models to data.

treatment of time, are discussed first. Code architecture details and the basic application programming interface (API) of the major modules are presented afterward.

3.1. PINT Coordinates and Time

As discussed in Section 2.2.1, the description of the pulsar signal is relatively simple in a nearly inertial frame, such as that of the SSB. As with most other timing packages, PINT uses the SSB as its reference frame for pulsar timing models. Given the design of PINT, if other reference frames are required, such as that of the pulsar, they can be added in a relatively straightforward manner.

NASA's Jet Propulsion Laboratory (JPL) has adopted the International Celestial Reference System (ICRS) J2000 reference frame as the base coordinate system for all of their solar system ephemeris calculations (Folkner et al. 2014). Therefore, because PINT uses the JPL ephemerides, all internal PINT calculations are performed in this coordinate system. A pulsar's position and velocity are generally specified by astrometric parameters (e.g., R.A., decl., and proper motions in the ICRS frame) as part of a timing model. The observatories' positions and velocities are tabulated by the PINT `observatory` module, which is discussed in Section 3.3.2. Coordinate transformations are performed using ASTROPY routines whose algorithms are provided via the Essential Routines for Fundamental Astronomy package (ERFA), a rebranding of the Standards Of Fundamental Astronomy (SOFA) library[35] Astropy Collaboration et al. (2013).

PINT assumes the TOAs it reads to be MJD values in the timescale of the observatory where they were recorded (the observatory timescales are handled in the `observatory` module; see Section 3.3.2), although PINT can also accept TOAs in other "special" reference frames, such as those at the SSB or at the geocenter. To store these MJDs at the required numerical precision of ∼1 ns, PINT uses the `astropy.time.Time` object,[36] where two 64 bit floats represent the integer and fractional parts of each MJD. Because there is no standard way of representing UTC times on leap days as normal MJDs,[37] PINT follows TEMPO and TEMPO2 in defining a custom time format called `pulsar_mjd`, in which the integer part is the normal integer MJD and the fractional part is the seconds of the day divided by 86,400. This means that MJDs "tick" at a constant rate, but there is no representation for a time during a leap second and, therefore, no way to represent a TOA during that time. This is not normally an issue as leap seconds are rare. One can usually just make the reference time for the TOA a second earlier or later so it does not occur during a leap second. In order to convert TOAs to Barycentric Dynamical Time (TDB), a sequence of clock corrections has to be applied to the TOAs. The raw TOAs are typically referenced to an observatory clock, often a GPS-disciplined rubidium clock or hydrogen maser. This timescale is denoted as UTC(obs), where "obs" is the name of the observatory. PINT applies the usually known local clock corrections to convert UTC(obs) to UTC(GPS), a timescale maintained by the U.S. Naval Observatory (USNO). Those corrections use either TEMPO or

[35] http://www.iausofa.org/

[36] http://docs.astropy.org/en/stable/time/

[37] MJDs with fractional days are commonly used for UTC times in many applications; however, there is no unique way to assign MJDs to times during days with leap seconds. Two conventions are in use: the fractional part can be seconds divided by 86,400 (in which case there is no way to represent a time during a leap second), or the fractional part can be seconds divided by 86,401 (in which case MJD advances at a different rate during days with leap seconds). In addition, for two MJDs, the difference MJD2 − MJD1 does not correctly give the time interval between two times because of possible leap seconds between MJD1 and MJD2, so care must be taken when computing time intervals.

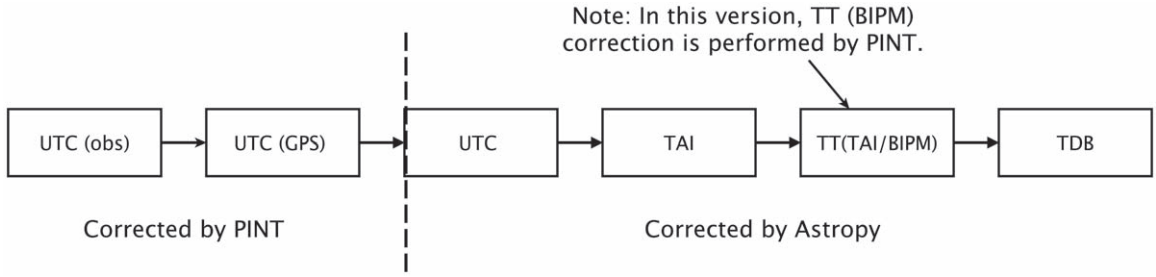

Figure 2. PINT converts TOAs from the observatory local time UTC(obs) to TDB following the steps illustrated. PINT handles the conversion from UTC(obs) to UTC (GPS) and the TT(BIPM) correction. The other parts of the clock corrections are performed by ASTROPY.

TEMPO2 format clock files, which are obtained from observatories by various means and must be kept up to date. By default, PINT uses the set of TEMPO-format clock files distributed with PINT in `src/pint/datafiles`. If needed, PINT is also able to read the clock correction files from TEMPO/TEMPO2 clock directories. A further correction can be applied to convert UTC(GPS) to the standard UTC, maintained by the International Bureau of Weights and Measurements (BIPM), using the TEMPO2-format `gps2utc.clk` file (which must also be kept up-to-date) in `pint/datafiles`. Those corrections are derived from BIPM Circular T.[38] Whether this correction is applied can be controlled via the `observatory` API, which is discussed in Section 3.3.2.

UTC is converted to International Atomic Time (TAI; using ASTROPY) by adding an integer number of leap seconds and then to Terrestrial Time (TT, also known as Terrestrial Dynamical Time, or TDT), which ticks at the same rate as TAI and UTC but, for reasons of continuity, has an offset. A TT day has a duration of 86,400 s on the geoid and is the independent argument of apparent geocentric ephemerides. The most common realization of TT is TT(TAI), which is defined as TT(TAI)=TAI + 32.184 s. However, PINT can also use TT(BIPM), which is a more accurate realization of TT published by the BIPM. In PINT, this clock correction is read from the TEMPO2-style clock file `pint/datafiles/tai2tt_bipm2015.clk` (or an alternative file based on the approximately annual publication of the BIPM timescales). Whether this correction is enabled is controlled by the `include_bipm` argument to `pint.observatory.get_observatory()`, and if it is, the version of TT(BIPM) can be selected by the `bipm_version` argument.

Finally, times are converted from TT to a barycentric time. There are two such time systems in common use. Traditionally, pulsar timing has been done using TDB, which is the independent variable of the JPL planetary ephemerides (Standish 1998). The alternative is Barycentric Coordinate Time (TCB), which is the preferred timescale according to the International Astronomical Union (IAU). TCB is a relativistic coordinate time and the modern definition of TDB is a linear scaling of TCB (IAU Resolution 3 of 2006[39]). The tick rates of the two differ by about a part in 10^8, so the value of model parameters that have a time component in the unit are different depending on the choice of barycentric timescale. Currently, TEMPO and PINT only support TDB, while TEMPO2 uses TCB as its default but allows the choice of TDB for compatibility. In the future, PINT will be extended to support TCB.

In PINT, the default conversion from TT to TDB is handled by ASTROPY, which uses the SOFA library to perform the conversion. The difference TDB–TT is quasi-periodic, dominated by an annual term of amplitude 1.7 ms. The SOFA routines implement an approximation to this function using over 800 terms from Fairhead & Bretagnon (1990) and include a location-dependent correction. PINT also provides the infrastructure to incorporate other types of TT–TDB corrections (e.g., numerical the TT–TDB difference provided by JPL ephemerides or the IF99 method; see Irwin & Fukushima 1999). The complete PINT clock correction chain is illustrated in Figure 2.

Several of the clock corrections are based on published or measured data provided by observatories or international organizations. Section 3.3.3 describes the scheme PINT uses for reading and updating these external data sets.

Note that clock corrections as described here are independent of corrections for light travel time: although the times at the end of this process are in TDB, they have not been corrected for light travel time across the solar system and are therefore not what pulsar astronomers conventionally call "barycentered." That process happens later because the correction depends on astrometric parameters from the timing model and a solar system ephemeris, not just the TOAs themselves.

3.2. PINT Code Architecture

PINT is designed to be highly modular. According to the pulsar timing procedures introduced in Section 2, PINT organizes its code in four major independent modules: `pint.toa`, `pint.models`, `pint.residuals`, and `pint.fitter`.

The `pint.toa` module provides the container classes used to store and manipulate TOAs and their corresponding metadata, while `pint.models` contains the classes that implement the various timing models to predict TOAs. The `pint.fitter` module provides classes that vary timing model parameters to optimally fit the TOAs. Typically, such a comparison between the TOAs and timing model occurs through the use of the `pint.residuals` module.

Each of these modules provides public interface classes for common usages. The classes `TOAs`, `TimingModel`, and `Residuals` are used to interface with the modules `pint.toa`, `pint.models`, and `pint.residuals`, respectively. These interface classes can be initialized independently, allowing one to analyze details of a pulsar's timing model without having TOAs from the pulsar. This flexibility is one of the key innovations of the PINT package. The interface to the `pint.fitter` module depends on the chosen fitting method (e.g., the `WLSFitter` class for a weighted least-squares fit

[38] https://www.bipm.org/en/bipm-services/timescales/time-ftp/Circular-T.html
[39] https://www.iau.org/administration/resolutions/ga2006/

Table 1
PINT Common Modules

Module Name	Provides	Reference Section
`toa`	TOA[a] container and API	2.1
`observatory`	Observatory's position, velocity, and clock corrections	3.3.2
`models`	Timing model API and built-in model components	2.2
`residual`	Residual container and API	3.5
`fitter`	Fitter API and built-in fitting algorithms	3.6
`pintk`	PINT Graphical user interface	6
`scripts`	Commonly used command-line scripts	6

Note.
[a] Time of arrival.

versus the `GLSFitter` class for generalized least squares), but all fitter classes require instances of both `TOAs` and `TimingModel`, which are compared internally using `Residuals`. Table 1 lists the frequently used modules in PINT.

One of the most common uses of PINT is to mirror the standard TEMPO functionality of updating existing timing models using newly observed data. All four modules must be used together in order to achieve this functionality. The code example in Figure 3 demonstrates how to use PINT as a substitute for TEMPO/TEMPO2. In this example, the four primary PINT classes or class types (`TOAs`, `TimingModel`, `resids`, and the fitting classes in `pint.fitter`) work together following the operation model in Figure 1.

In the following sections, these four key modules and APIs will be discussed in detail.

3.3. TOA Module

As introduced above, the `pint.toa` module provides the container class (`TOAs`) and APIs for reading, processing, storing, and interacting with TOAs. However, during TOA processing, the `pint.observatory` module also plays an essential role behind the scenes.

3.3.1. Handling TOAs

Typically, a user will read in and preprocess TOAs using the convenience function `toa.get_toas()` as shown in the code example in Figure 3 and discussed in Section 3.2. The TOAs and associated metadata (e.g., observing frequencies, TOA errors, observatories used, etc.) are typically read from a set of text files known as .TIM files. Currently, `toa.get_toas()` can read "Princeton," "Parkes," and "TEMPO2" formatted TOAs.[40] All of the TOA information is stored in the publicly accessible attribute `TOAs.table`, which is an instance of an `astropy.Table` object, allowing PINT to take advantage of the latter's high-level table access and manipulation capabilities. For example, table columns and associations can be easily defined or modified, and subsets of TOAs can easily be selected or deselected. Figure 4 provides examples of using the `TOA.table` object.

The `toa.get_toas()` function processes the raw TOAs upon reading using three `TOAs` class methods: `apply_clock_corrections()`, `compute_TDBs()`, and `compute_posvels()`. These methods transform the TOAs to the TDB timescale and compute the solar system objects' positions and velocities in the ICRS J2000 coordinate system at those times. Because the coordinate and time transformations are highly observatory dependent, these three `TOAs` class methods are actually high-level wrappers of several detailed computations provided in the `observatory` module, which is discussed in Section 3.3.2. The `toa.get_toas()` method also allows the user to control the version of external data (Section 3.3.3 discusses the external data handling scheme) used in these wrapped functions via the input arguments. Traditionally, this information is stored in the timing model parameter (.PAR) files, which are processed by the `pint.models` module. To avoid the inconvenience, PINT 0.8.0's `toa.get_toas()` accepts the `TimingModel` object, where the versions of external data are saved, as an input argument and applies them to the TOAs. The read-in and clock-corrected TOAs are stored in the `TOAs.table[''mjd'']` column as `astropy.time.Time` objects.[41] The use of tables allows for flexible organization and handling of TOAs, allowing users and developers the ability to quickly and efficiently index and select TOAs. For convenience, and with approximately the same ∼1 ns precision, the TDB times in MJD format from `compute_TDBs()` are stored in the `TOAs.table[''tdbld'']` column as an `np.longdouble`[42] array, which can be directly used in most NUMPY and SCIPY vector calculations. Some intermediary results of the time transformations (e.g., TOAs in Terrestrial Time) are saved in additional `TOAs.table` columns, allowing the user to have easy access to these results if needed. Observatories' positions and velocities, using ASTROPY quantities and units, in the ICRS J2000 frame are computed by the `compute_posvels()` class method and saved in the `TOAs.table` columns `''ssb_obs_pos''` and `''ssb_obs_vel,''` respectively. The positions of the Sun and major planets are also computed by `compute_posvels()` to enable solar system Shapiro delay calculations. Table 2 lists the `TOAs.table` columns after calling the `get_toas()` function. For efficiency, PINT can pickle[43] the TOAs and

[40] http://tempo.sourceforge.net/ref_man_sections/toa.txt

[41] The `astropy.time.Time` object uses a pair of 64 bit floating-point numbers to represent times (integer and fractional parts of the Julian Day number) and, as a result, is capable of 20 ps precision. Unfortunately, few mathematical operations can be used directly on these objects.

[42] The type `np.longdouble` uses the underlying C implementation's `long double` type. On most Intel machines, this is a hardware-supported 80 bit floating point packed into larger blocks of memory. The Microsoft Visual C runtime defines this type to have only 64 bits, and so PINT cannot run there. Other machines may define `long double` to be either software- or hardware-supported quadruple precision or software-supported double–double precision (for example Arm64, Power9, and Power7 architectures, respectively). In any case, PINT will refuse to run if this data type cannot support nanosecond precision on MJDs.

[43] Pickling is a process that serializes a `Python` object to a binary format that can be efficiently written to a file. https://docs.python.org/3/library/pickle.html.

```
>>> from pint.models import get_model
>>> from pint.toa import get_TOAs
>>> from pint.residuals import Residuals
>>> import pint.fitter
>>> import astropy.units as u
>>> # Initialize PINT TimingModel object using a TEMPO/TEMPO2 style parameter file
>>> m = get_model("NGC6440E.par")
>>> # Initialize PINT TOAs object using a TEMPO/TEMPO2 style TOAs file
>>> t = get_TOAs("NGC6440E.tim")
>>> # Create the residuals with a less accurate model
>>> rst = Residuals(t, m).time_resids
>>> # Print out the rms of the residuals.
>>> print("RMS of pre-fit time residual is {}".format(rst.std().to(u.us)))
   RMS of pre-fit time residual is 1099.12298526 us
>>> # Updating the model.
>>> # Initialize Fitter object with TimingModel object and TOAs object
>>> f = pint.fitter.WLSFitter(t, m)
>>> # Fit the data and update the model.
>>> chi2 = f.fit_toas()
>>> print("Post-fit Chi square is {}".format(chi2))
   Post-fit Chi square is 59.5742964653
>>> print("RMS of post-fit time residual is {}".format(f.resids.time_resids.std().to(u.us)))
   RMS of post-fit time residual is 33.3342840421 us
>>> print(f.model.as_parfile())
   PSR                              1748-2021E
   EPHEM                                DE421
   UNITS                                  TDB
   RAJ                     17:48:52.80034692 1 0.0000000003756850254201
   DECJ                   -20:21:29.38330660 1 0.0000091254258689169 6
   PMRA                                   0.0
   PMDEC                                  0.0
   PX                                     0.0
   POSEPOCH            53750.000000000000000
   F0                  61.485476554372500035 1 1.8086084392781505522e-11
   F1              -1.1813316309089768527e-15 1 1.4418540386147890052e-18
   PEPOCH              53750.000000000000000
   TZRMJD              53801.386051182230000
   TZRSITE                                1
   TZRFRQ                          1949.609
   PLANET_SHAPIRO                         N
   NE_SW                                  0.0
   SWM                                    0.0
   DM                  224.11379738507580495 1 0.034938980494130779386
   DM1                                    0.0
```

Figure 3. Code example showing PINT being used like TEMPO to update an existing pulsar timing model using observed TOAs.

computed data for later use if the `usepickle` flag is enabled in `get_toas()`. The performance difference between pickling and non-pickling is discussed in Section 5.1.

3.3.2. Handling Observatories

The `observatory` module stores fundamental observatory information and provides additional coordinate and time transform functionality for both stationary and moving observatories (i.e., satellites). The base class, `Observatory`, provides the unified API for obtaining observatory positions and velocities, computing the clock correction values and calculating time transformations to TDB with the methods `posvel()`, `clock_corrections()`, and `get_TDBs()`, respectively. However, because these calculations may be observatory specific, their implementations are in the various `Observatory` subclasses. This scheme allows PINT to handle TOAs from different observatories simultaneously and clearly.

There are currently two `observatory` subclasses, `TopoObs` and `SpecialLocation`. The `TopoObs` class is implemented for stationary, ground-based observatories, such as most traditional radio telescopes (e.g., Arecibo Observatory and Green Bank Observatory). Ground-based observatories follow the standard procedure of coordinate transformation and clock correction from the Earth corotating frame to the ICRS frame (i.e., applying the clock corrections and coordinate transformations introduced in Section 3.1). Creating a `TopoObs` object requires the observatory name, aliases (i.e., as often used on TOA lines), and coordinates under the International Terrestrial Reference Frame[44] (ITRF; Altamimi et al. 2011).

In contrast, the `SpecialLocation` class is designed to implement the observatories that are not in a fixed location corotating with Earth, such as the imaginary solar system

[44] http://itrf.ensg.ign.fr/

```
>>> import pint.toa as toa
>>> tim = "NGC6440E.tim"
>>> t = toa.get_TOAs(tim)
INFO: Applying clock corrections (include_GPS = True, include_BIPM = True. [pint
.toa]
INFO: Evaluating observatory clock corrections. [pint.observatory.topo_obs]
INFO: Applying GPS to UTC clock correction (~few nanoseconds) [pint.observatory.
topo_obs]
INFO: Applying TT(TAI) to TT(BIPM) clock correction (~27 us) [pint.observatory.t
opo_obs]
INFO: Computing TDB columns. [pint.toa]
INFO: Doing astropy mode TDB conversion [pint.observatory.observatory]
INFO: Computing positions and velocities of observatories and Earth (planets = F
alse), using DE421 ephemeris [pint.toa]
WARNING: No ephemeris provided to TOAs object or compute_TDBs. Using DE421 [pint
.toa]

Print out the summary
>>> t.print_summary()
Number of TOAs:  62
Number of commands:  1
Number of observatories:  1 ['gbt']
MJD span:  53478.286 to 54187.587
gbt TOAs (62):
  Min error:     13.2 us
  Max error:     118 us
  Mean error:    26.9 us
  Median error:  22.1 us
  Error stddev:  15.6 us

Print out the toa table's first 5 row.
>>> print(t.table[0:5])
index      mjd      ...              obs_sun_pos [3]
                    ...                    km
----- ------------- ... ----------------------------------------
    0 53478.2858714 ...    132300219.0054355 .. 28301415.35927446
    1 53483.2767052 ...  125950526.54693596 .. 32709720.950028352
    2 53489.4683898 ...      116811489.07975 .. 37847344.14583803
    3 53679.8756459 ... -107617035.22822961 .. -40589908.43792468
    4 53679.8756454 ... -107617036.21852377 .. -40589908.02736856

Check out the columns in the table
>>> t.table.columns
<TableColumns names=('index','mjd','mjd_float','error','freq','obs','flags','tdb
','tdbld','ssb_obs_pos','ssb_obs_vel','obs_sun_pos')>

Check out the toas stored in the table
>>> t.table[0]["mjd"]
<Time object: scale='utc' format='pulsar_mjd' value=53478.2858714>

Print out tdb time in longdouble format
>>> t.table["tdbld"][0:5]
<Column name='tdbld' dtype='float128' length=5>
53478.286614308378386
53483.277448077169016
53489.469132675783513
53679.87638877491714
53679.87638821944874
```

Figure 4. Code example for TOA module.

Table 2
Information Stored in the `TOAs.table` Object

Column Name	Descriptions	Data Type	Unit
mjd	TOA[a] at observatory in UTC	`astropy.time.Time`	MJD
error	TOA error	`np.float`	μs
freq	TOA observing frequency	`np.float`	MHz
obs	Observatory name/code	`str`	None
flags	Command flags	`dict`	None
tdb	TOA in TDB[b]	`astropy.time.Time`	MJD
tdbld	TOA in TDB in long double format	`np.longdouble`	MJD
ssb_obs_pos	SSB[c] → Observatory position vector	`np.float`	km
ssb_obs_vel	Observatory velocity (referenced to SSB)	`np.float`	km s^{-1}
obs_Sun_pos	Observatory → Sun center position vector	`np.float`	km

Notes.
[a] Time of arrival.
[b] Barycentric dynamical time.
[c] Solar system barycenter.

barycenter (SSB) "observatory" or an Earth-centered "observatory" (i.e., the geocenter). Another use case for the `SpecialLocation` class is the implementation of space-based observatories such as Fermi (Atwood et al. 2009) and NICER (Gendreau et al. 2012), where orbital information or other spacecraft flight data are required rather than ITRF coordinates. Detailed and observatory-specific calculations are provided by individual `Observatory` objects, whereas the `SpecialLocation` class implements only the high-level APIs for these calculations.

In the current PINT version, many observatories, both real and imaginary (like the geocenter and SSB), are predefined in the `observatory` module. Most users will create an observatory instance with the convenience function `get_observatory()`, which takes the observatory string name or TEMPO-style observatory code as an input argument. The special position/velocity or time transformation algorithms and their required external data sets or versions can be selected with optional arguments (e.g., the `include_gps` and the `include_bipm` arguments).

3.3.3. Handling External Data

Performing time and coordinate transformation requires external data such as JPL solar system ephemerides and observatory clock correction files. Traditionally, TEMPO provides copies of these data within the packages, and TEMPO developers keep them up to date. However, the upstream data are updated frequently, resulting in TEMPO developers frequently updating their packages. Thus, their users must reinstall their packages frequently, rather than simply updating the data directly. ASTROPY provides PINT with an easier way to keep these data up to date as many standard timing-related data sets (including but not limited to Earth rotation data, leap seconds, and JPL solar system ephemerides) are updated by ASTROPY. For Earth-orientation parameters (i.e., IERS table A and B[45]) and solar system ephemerides, ASTROPY downloads and caches them when requested. However, for ASTROPY versions earlier than 3.2, it requests an upgrade on the package itself to keep the leap seconds up to date instead of downloading the newest version of leap seconds. Data not currently handled by ASTROPY, such as observatory-specific clock corrections, are updated by the PINT development team in the traditional manner. Furthermore, there are plans for automatic updates of many of these data sets in future PINT releases.

3.4. Models Module

The PINT `models` module provides an API for implementing and interacting with pulsar timing models. In this section, the overall design of the `models` module is presented in the beginning, and the public interface object, the `TimingModel` class, is discussed after. The details of how to programmatically implement a timing model are in the Appendix. Note that this paper does not discuss the implementation of any specific timing model. For these details, please see the online documentation.[46]

Following the philosophy of modularity, PINT implements different physical effects separately as model components, which are implemented independently in the `Component` class and its subclasses. Results computed for a timing model are produced by combining the values from the selected components. The delays produced by each component are simply added together, but for components whose value depends on time—for example, the Römer delay depends on the pulsar's position in its orbit—the time at which each component is evaluated depends on the delays of other components. This requires the components to be computed in a specific order; this order is enforced by PINT but can be overridden by users if necessary (say for custom model components).

A model component implementing a particular mathematical model of a physical effect would be implemented in a subclass of the base `Component` class; this base class is where the common attributes and functionality of all model components are implemented. The `TimingModel` class is designed to manage the set of included components and provides the overall interface for collecting and returning the results from them, without requiring the calling code to know the details of the specific model.

As described in Section 2.2, modeling TOAs includes two fundamental calculations, total time delay (Δ in Equations (2) and (3)), and total phase (Equation (1)). PINT therefore

[45] https://datacenter.iers.org/eop.php

[46] https://nanograv-pint.readthedocs.io/en/latest/api/pint.models.timing_model.html

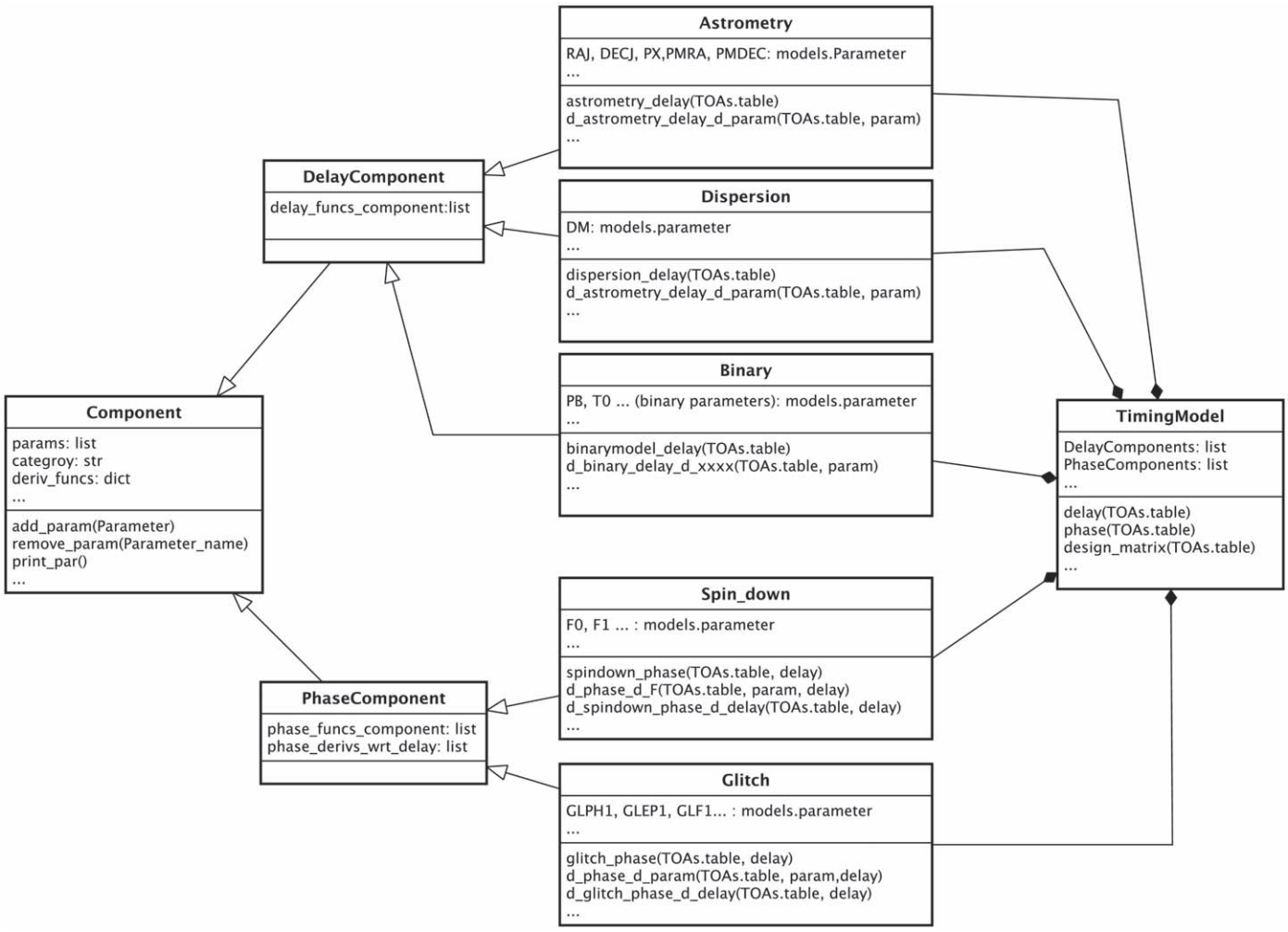

Figure 5. This figure shows an example of how PINT implements a full timing model. Hollow arrows indicate inheritance, while solid arrows indicate containment. `Astrometry`, `Dispersion`, and `Binary` classes inherit from the `DelayComponent` class. `Spin_down` and `Glitch` inherit from `PhaseComponent`. Both `DelayComponent` and `PhaseComponent` inherit from the generic `Component` base class. A `TimingModel` instance manages all of the specific model components needed to build the full model. Here, we only use `DelayComponent` and `PhaseComponent`, yet if other component types (e.g., `NoiseComponent`) are present, they follow the same relationship structure.

implements two explicit `Component` subclasses, `DelayComponent` and `PhaseComponent`. The `TimingModel` class provides two corresponding methods, `.delay()` and `.phase()`, to compute the total delay and total phase by adding the results from all the delay and phase components that are included in the model.

PINT is not limited to these component types and is completely extensible to other types. For example, PINT also provides a noise model component type, `NoiseComponent`, for handling timing noise models used in generalized least-squares (GLS) fitting and Bayesian timing analyses (Ellis 2013; van Haasteren 2013). Similarly, the `TimingModel` class also includes the APIs to compute other useful quantities. For instance, the `TimingModel` class is able to compute the design matrix, a key feature needed by the `fitter` module, via the `.designmatrix()` method. In Figure 5, the layout of the model and component class system is visually illustrated using example model components.

As described in Section 3.2, a `TimingModel` object can be initialized via the `models.get_model()` function with a TEMPO/TEMPO2-style .PAR file as input. Based on the input .PAR file, the `models.get_model()` function selects and sorts the required components, constructs the `TimingModel` object, and parses the parameter values. More details about the construction of `TimingModel` instances are discussed in Appendix A.2. Since version 0.8, PINT has also provided a wrapper function, `models.get_model_and_toas()`, that creates the `TimingModel` object and `TOAs` object together from the input .PAR and .TIM files, while applying relevant information in the .PAR file to the new `TOAs` object (e.g., which solar system ephemeris to use, for instance). Additionally, the `TimingModel` object allows users to manipulate the components interactively, beyond simply changing parameter values. For example, one can change the order of the components or disable individual components. This design facilitates interactive pulsar timing data processing, which can sometimes be difficult with compiled programs. A timing model can be adjusted and examined interactively and intermediate computational results can be accessed as needed.

The `models` module comes with commonly used timing-model components and functionality. Table 3 lists the built-in model components in PINT 0.8.0. For the most updated `model` module and built-in components information, please visit our online documentation.

Table 3
PINT Version 0.8.0 Built-in Timing Model Categories and Components

Model Category	Category Description	Component Name	Reference
astrometry	Solar system geometric effects	AstrometryEquatorial	1
		AstrometryEcliptic	2
solar_system_shapiro	Solar system Shapiro delay	SolarSystemShapiro	3
dispersion_model	Interstellar media dispersion effects	Dispersion	4
		DMX	5
pulsar_system	Pulsar system time delay	BinaryELL1	6
		BinaryELL1H	7
		BinaryDD	8
		BinaryDDK	9
		BinaryBT	10
spindown	Spindown phase	Spindown	11
glitch	Glitch phase	Glitch	12
frequency_dependent	Frequency evolution of pulsar profiles	FDdelay	13
jump	Jump phase offset	JumpPhase	14
scale_toa_error	Template fitting timing noise correction	ScaleToaError	15
ecorr_noise	ECORR type noise model	EcorrNoise	16
pl_red_noise	Power-law red-noise-type noise model	PLRedNoise	17
ifunc	Interpolated timing noise	IFunc	18
wave	Sinusoidal timing noise decomposition	Wave	19
solar_wind	Dispersion due to the solar wind	SolarWindDispersion	20
troposphere	Delay due to the local atmosphere	TroposphereDelay	21

Note. (1), (4), (11) Backer & Hellings (1986), (2), (5), (13), (15)–(17) The NANOGrav Collaboration et al. (2015), (3) Shapiro (1964), (6) Lange et al. (2001), (7) Freire & Wex (2010), (8) Damour & Deruelle (1986), (9) Kopeikin (1995, 1996), (10) Blandford & Teukolsky (1976), (12), (14) Hobbs et al. (2006), (18) Deng et al. (2012), (19) Hobbs et al. (2010), (20) Edwards et al. (2006), (21) Davis et al. (1985), Niell (1996), CRC Handbook (2004).

3.5. Residual Module

Residuals between the data (i.e., TOAs) and the timing model are key to updating model parameters and assessing the goodness of fit. The `residuals` module is designed to compute the residuals using Equations (6) and (7). The interface class, `Residuals`, instantiated by providing `TOAs` and `TimingModel` instances, implements the `.calc_phase_resids()` method and `.calc_time_resids()` method to compute the phase residuals and time residuals, respectively. For a better representation of the difference between the timing model and the TOAs, the residuals are by default weighted by the TOA uncertainty, but this feature can be switched off in the class method argument. In addition, if specific pulse numbers are provided, the residuals can be calculated based on those, rather than using the calculated nearest integer pulse. Together with the residual calculation methods, a handful of convenience methods for computing statistics of the residuals are provided (e.g., the χ^2 and reduced χ^2 values).

3.6. Fitter Module

The updating of timing models is performed by the `pint.fitter` module, which includes a general API base class `fitter.Fitter` and a set of predefined fitter subclasses implementing specific optimization algorithms. The general API base class `Fitter` sets up the framework, and the fitter subclasses implement the fitting algorithms under the `.fit_toas()` class method. This setup allows the user to implement a new fitting algorithm with minimum code modifications (only overwriting the `.fit_toas()` method), but using the same interface. Table 4 lists all of the built-in fitters in PINT 0.8.0. PINT implements priors for parameters (see Appendix A.1) that are used in the Markov Chain Monte Carlo (MCMC) fitter (and which can be used also to effectively constrain the values of parameters). However, the other fitters currently do not use priors or constraints, and no fitters can currently fit for noise parameters. A common package used to compute noise parameter values is `enterprise`.

Table 4
PINT Implemented Fitting Algorithms

Fitter Name	Algorithm
PowellFitter	SCIPY Powell minimizing
WLSFitter	Weighted least-squares fitting
GLSFitter	Generalized least-squares fitting
MCMCFitter	Markov Chain Monte Carlo optimization fitting
WidebandTOAFitter	TOAs and independent dispersion measurements joint fitting[a]

Note.
[a] The independent dispersion measurements are fitted with TOAs simultaneously using generalized least-squares fitting (Pennucci 2019; Alam et al. 2021b).

As described in the code example in Figure 3, a fitter class should be instantiated with `TOAs` and `TimingModel` objects. The `TimingModel` object will be linked to the `fitter.model_init` attribute, and an extra copy will be saved in the `fitter.model` attribute in order to retain the initial timing model. During fitting, the `fitter.model` attribute will be updated, but the `fitter.model_init` stays the same. Under this scheme, the original timing model can be easily reset using the class method `fitter.reset_model()`. Residuals are calculated and saved in the `fitter.resids` attribute, and a copy of the initial residuals is saved to `fitter.resids_init` using the same scheme.

One of the most important functionalities of the fitter API is to alter the model parameter values. The `Fitter` base class already provides a set of convenience functions for this

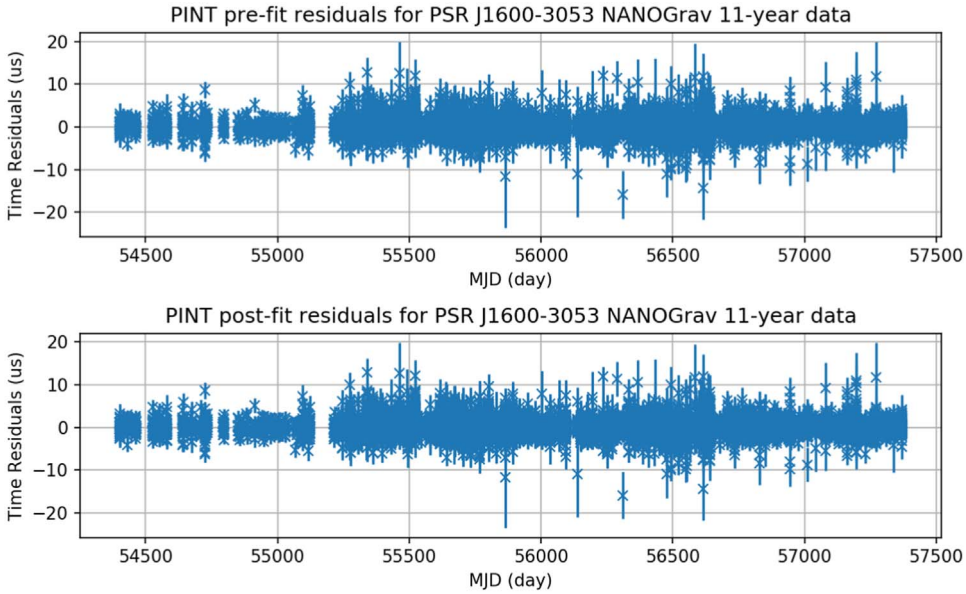

Figure 6. Residuals generated by PINT for PSR J1600−3053 from the NANOGrav 11 yr data set. The top panel shows residuals before performing a generalized least-squares fit based on the published TEMPO-based timing solution. The bottom panel shows the residuals after the fit using PINT. The rms of the residuals are nearly identical.

purpose. For example, the `.set_params()` class method is designed for changing parameter values, and the `.free_params()` method (originally the `.set_fitparams()` method) can be used for selecting the parameters to be fitted.

As described above, the postfit results are returned via the `fitter.model` attribute, and the `fitter.resids` attribute will be updated with the postfit residuals. This newly fitted timing model and residuals are then ready for the next iteration.

4. Comparison of PINT with TEMPO/TEMPO2

One way to validate PINT is to compare its results with those from the existing high-precision pulsar timing software packages (i.e., TEMPO version 13.101 and TEMPO2 version 2019.01.1). In addition to validating PINT, such a comparison checks the accuracy and precision limitations of the various software packages. As of version 0.8.0, PINT is capable of analyzing the TOAs from most pulsars, including the 45 pulsars from the NANOGrav 11 yr data release (Arzoumanian et al. 2018). Here we present the results of a PINT analysis of PSR J1600−3053 from the NANOGrav 11 yr data set, using the DD binary model, including a detailed comparison between PINT and TEMPO results. PSR J1600−3053 was chosen for this comparison because it has a large number of TOAs (12433) with sub-microsecond timing precision over a long time span (8 yr). This comparison will also highlight some implementation differences between PINT and TEMPO/TEMPO2. A full-scale PINT-TEMPO/TEMPO2 comparison using all of the pulsars from NANOGrav's 12.5 yr data is reported in Alam et al. (2021a). The Jupyter notebook for this comparison is included in the PINT examples and can be viewed from the PINT online documentation.[47]

4.1. Comparison Using PSR J1600−3053

We used the published NANOGrav 11 yr ephemeris (originally produced using the TEMPO software package) as our initial timing model, fitted to TOAs from the NANOGrav 11 yr data using the PINT GLS fitter `pint.fitter.GLSfitter`.

The prefit residuals from PINT had a weighted rms (WRMS) value of 0.944 μs. The fitting process reported a final χ^2 value of 12368.10 for 12,307 degrees of freedom, and the postfit residuals had a WRMS of 0.944 μs. Figure 6 shows the PINT prefit and postfit residuals.

In the following subsections, the results of a detailed comparison between PINT and TEMPO/TEMPO2 are presented.

4.1.1. Comparison with TEMPO Results

The TEMPO-based fitting for the same data set returns a χ^2 value of 12,368.46 and the residuals have a WRMS of 0.944 μs. We directly compared both the prefit and postfit residuals between these two packages. In Figure 7, the residual differences between PINT and TEMPO are presented. Note that because we dropped the constant part of the absolute phase in our calculation, a constant offset in the residual differences has been ignored.

In the prefit residual differences, a distinct annual periodic signature, with a peak amplitude of about 20 ns, is present throughout the whole data set. This discrepancy is due primarily to different precession–nutation models used in PINT and TEMPO. PINT uses ASTROPY's built-in precession–nutation model (see the IAU 2000 resolution; McCarthy & Capitaine 2002), while TEMPO uses much older models, the IAU 1976 precession (Lieske et al. 1977) and IAU 1980 nutation (Seidelmann 1982) models. The difference between these models and their impact on timing residuals has been discussed in Hobbs et al. (2006). Due to a lack of polar motion in the TEMPO-style precession–nutation model, the expected timing residual differences should have an amplitude near ±30 ns with a diurnal signature that is modulated by annual and 435 day periodicities. Figure 8 illustrates the residual discrepancies due to the different precession–nutation models.

We compared the parameters resulting from GLS fits using TEMPO and PINT as well. The timing model parameter differences are listed in Table 5. All of the PINT postfit parameters are consistent with the TEMPO parameter values to

[47] https://nanograv-pint.readthedocs.io/en/latest/examples-rendered/paper_validation_example.html

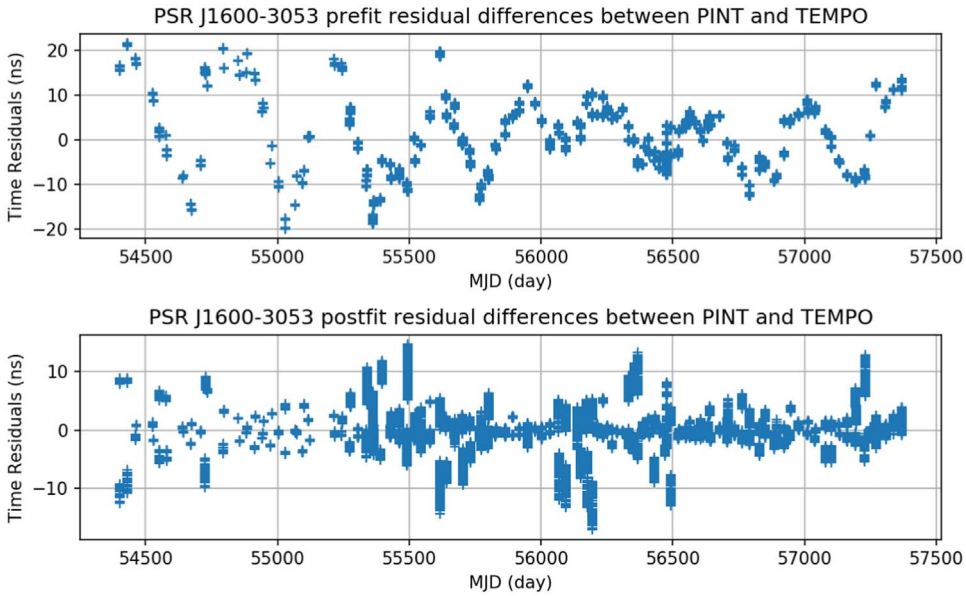

Figure 7. Residual differences between PINT and TEMPO for PSR J1600−3053. The upper panel presents the difference of prefit residuals, and the lower panel presents the postfit residuals difference.

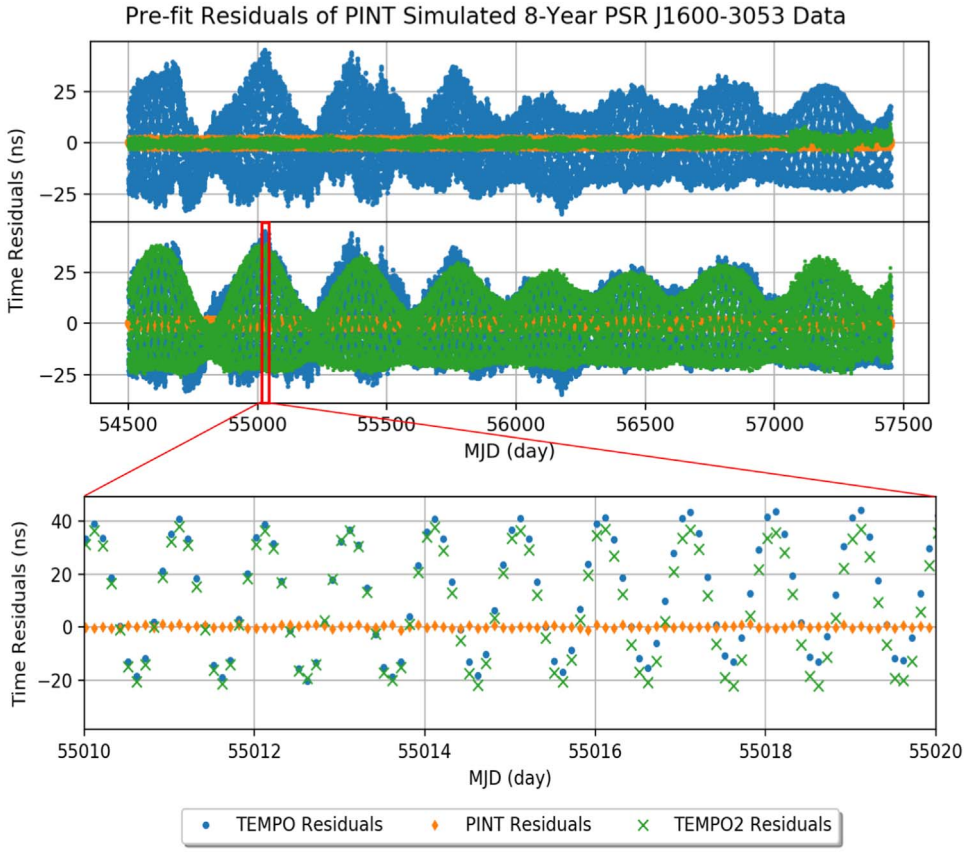

Figure 8. The residual difference due to different precession–nutation models. We use PINT to simulate 8 yr regularly sampled (2.4 hr cadence) TOAs with a simple timing model, only has a constant pulse frequency, and pulsar position. Orange marks represent the PINT residuals, the blue points are the TEMPO residuals, and green data points mark the TEMPO2 residuals. The first panel on the top shows the PINT and TEMPO/TEMPO2 residuals when TEMPO2 is under the IAU 2000 resolution of precession and nutation. The second panel displays the same results with TEMPO2's old precession and nutation mode, and TEMPO2's residuals have a similar signature to TEMPO residuals. The third panel is a zoomed-in version of the second panel on days from MJD 55010 to MJD 55020. We can see the diurnal sinusoidal oscillation from the TEMPO/TEMPO2 residuals. Given the sampling rate of NANOGrav 11 yr data, the TEMPO prefit residual differences in Figure 7 is one trace of the blue dots.

Table 5
PINT Parameter Comparison with TEMPO for PSR J1600−3053

Parameter	V_T[a]	Unit	$V_T - V_P$[b]	$\lvert V_T - V_P\rvert/\sigma_T$[c]	σ_P[d]$/\sigma_T$
F0	277.9377112429746(5)	Hz	-1.471×10^{-14}	0.028	1.000
F1	$-7.33874(5) \times 10^{-16}$	Hz s^{-1}	6.362×10^{-23}	0.014	1.000
FD1	$4.0(2) \times 10^{-5}$	s	-2.546×10^{-9}	0.002	1.000
FD2	$-1.5(1) \times 10^{-5}$	s	1.370×10^{-9}	0.001	1.000
JUMP	$-8.8(1) \times 10^{-6}$	s	-4.650×10^{-10}	0.004	1.004
PX	0.50(7)	mas	-2.070×10^{-3}	0.028	1.000
ELONG	244.347677844(6)	deg	-5.924×10^{-10}	0.099	1.000
ELAT	−10.07183903(3)	deg	-3.191×10^{-9}	0.095	1.000
PMELONG	0.46(1)	mas yr^{-1}	7.119×10^{-4}	0.068	1.003
PMELAT	−7.16(6)	mas yr^{-1}	-5.048×10^{-4}	0.009	0.999
PB	14.348466(2)	day	-3.457×10^{-8}	0.016	1.000
A1	8.8016531(8)	lt-s	1.491×10^{-8}	0.018	0.984
A1DOT	$-4.0(6) \times 10^{-15}$	lt-s s^{-1}	8.913×10^{-18}	0.014	1.000
ECC	$1.73729(9) \times 10^{-4}$	dimensionless	-2.386×10^{-10}	0.027	1.002
T0	55878.2619(5)	day	-1.051×10^{-5}	0.020	0.991
OM	181.85(1)	deg	-2.638×10^{-4}	0.020	0.991
OMDOT	$5(1) \times 10^{-3}$	deg yr^{-1}	-2.211×10^{-5}	0.016	1.000
M2	0.27(9)	solar mass	-1.641×10^{-3}	0.018	0.979
SINI	0.91(3)	dimensionless	5.436×10^{-4}	0.016	0.984
DMX_0010[e]	$6(2) \times 10^{-4}$	pc cm^{-3}	-5.089×10^{-6}	0.025	1.000

Notes.
[a] TEMPO fit parameter value.
[b] PINT fit parameter value.
[c] TEMPO fit parameter uncertainty.
[d] PINT fit parameter uncertainty.
[e] In the NANOGrav 11 yr data, PSR J1600−3053 has 106 DMX time ranges. Here we only list the one DMX parameter that has the largest difference between PINT and TEMPO.

well within the 1σ uncertainties. This shows that PINT is capable of reproducing the published result for PSR J1600−3053 in the NANOGrav 11 yr data set.

4.1.2. Comparison with TEMPO2 Results

Prior to the PINT-TEMPO2 comparison, we modified the timing model parameter files from the published NANOGrav 11 yr data set for a more controlled comparison. The 11 yr data set timing models used TEMPO, which has adopted the ecliptic coordinate frames with the 2010 IAU value of the obliquity (The NANOGrav Collaboration et al. 2015). However, TEMPO2 implements the ecliptic coordinate frame using the 2003 IAU obliquity value. Thus, we chose to use the 2003 IAU obliquity value in this comparison. Another modification is due to the discrepancy in the precession and nutation model mentioned in the previous section. Fortunately, TEMPO2 allows for the user to choose between the IAU 2000 resolution and the TEMPO-style precession–nutation model (Hobbs et al. 2006). Naturally, we decided to run TEMPO2 under the same precession–nutation model (IAU 2000 resolution) as PINT.

TEMPO2's GLS fitting gives a final χ^2 value of 12,265.16 and the postfit residuals have a WRMS of 0.944 μs. TEMPO2 residuals were also directly compared against the PINT residuals, and the comparison is shown in Figure 9. Again, a constant residual offset has been ignored here as well. Both the prefit and postfit residual differences are less than 10 ns, which is within the accuracy goal of TEMPO2 (Hobbs et al. 2006). However, the residual differences show a systematic quasi-periodic signature with a semiannual term that occurs consistently over the whole data set. The same signature is present in the PINT-TEMPO2 solar system geometric delay (i.e., Rømer Delay) difference as well. In Figure 10, the solar system geometric delay difference and the residual differences are plotted together. This common signature indicates that the 2.5 ns level residual discrepancies are due to a difference in the solar system geometric delay calculation (e.g., observatory position or pulsar sky location). We also compared the postfit parameters between PINT and TEMPO2 and all agree within the TEMPO2-reported parameter uncertainties (see Table 6).

4.2. Other Known Implementation Differences between PINT and TEMPO/TEMPO2

In this section, we present four major known implementation differences between PINT and TEMPO that could cause substantial differences in the results. We show differences in the timing between PINT and TEMPO for several other pulsars presented in the NANOGrav 11 yr data set.

*UTC(GPS) to standard UTC clock conversion (*TEMPO *only).* As described in Section 3.1, PINT converts UTC(GPS) time to the standard UTC timescale. However, the TEMPO package does not apply this 10 ns-level clock correction to the TOAs. In Figure 11, the UTC(GPS) and standard UTC clock correction values over the past two decades are plotted. *Constant time offset between TOAs correction (JUMP).* The constant time offset between TOAs, implemented as `JUMP`s in the timing model, can be introduced from two major effects: (1) a constant time delay from different instruments (e.g., different cable length), and (2) pulse-profile evolution delays (e.g., from the frequency evolution of the intrinsic pulse profile). Because the first type of time offset occurs at the observatory, it should be corrected at the

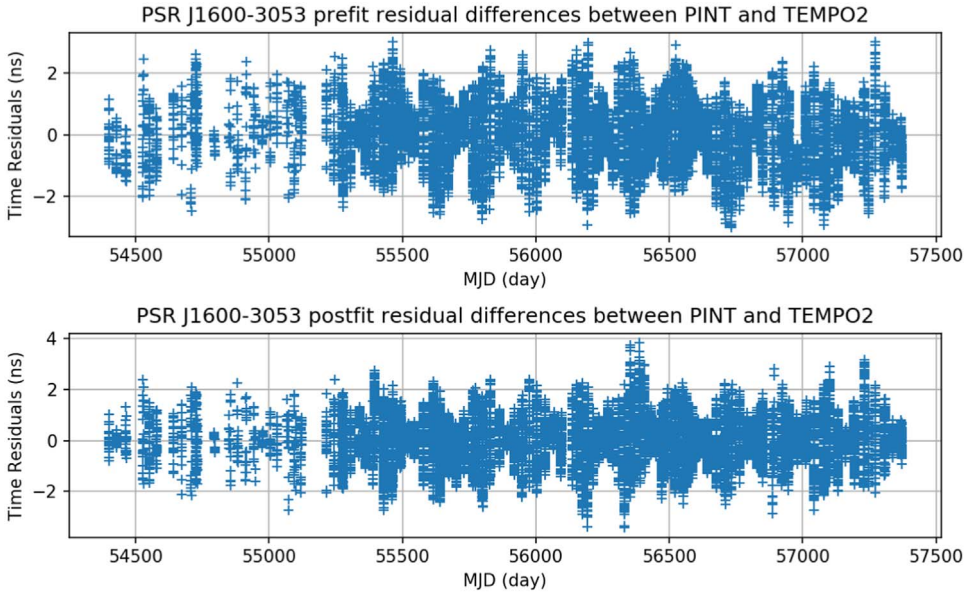

Figure 9. Residual difference between PINT and TEMPO2 for the J1600−3053 NANOGrav 11 yr data. The upper panel shows the prefit residual difference, and the lower panel shows the postfit residual difference.

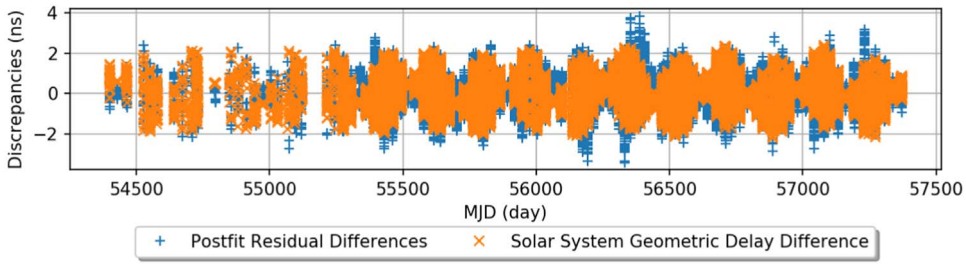

Figure 10. PINT-TEMPO2 residual differences and the PINT-TEMPO2 solar system geometry delay difference plotted on top of each other. The blue data points mark the difference between THE PINT and TEMPO2 postfit residuals, and the orange points mark the difference between THE PINT and TEMPO2 solar system geometric delay. Their envelopes trace each other and show that the 2 ns level residual discrepancies are caused by the solar system geometric delay implementation difference of these two software.

observatory frame (before computing the solar system barycentric TOAs). The pulse-profile offset is part of the intrinsic pulsar emission process. Thus, the second type of JUMPs is more appropriately applied under the pulsar frame. However, neither TEMPO nor TEMPO2 distinguishes these two types of JUMPs, and they correct both of them under the same reference frame. TEMPO corrects the JUMPs in either the observatory frame or the pulsar frame (TEMPO gives the options to the user). TEMPO2 applies the JUMP corrections in the pulsar frame in terms of phase offset. In this release of PINT, the JUMPs are applied in the same way as the TEMPO2 method. However, PINT has the infrastructure to apply the two types of JUMPs separately, and it is planned in future releases. Therefore, if TEMPO corrects the JUMPs at the observatory, a highly radio-frequency-dependent residual discrepancy with a period of one year will be present in the PINT-TEMPO residuals difference (see Figure 12). The peak value of this yearly signature is dependent on the JUMP offset values.

Frequency-dependent delay (FD delay). The frequency-dependent delay is implemented for modeling the pulse-profile variation at different radio frequencies by NANOGrav (The NANOGrav Collaboration et al. 2015). Instead of applying the FD delay before the pulsar binary correction like TEMPO/TEMPO2, PINT applies it to the TOAs after the binary model in the pulsar frame. This delay introduces an offset in the binary model input TOAs, which leads to a ∼10 ns level of residual difference, which depends on the FD parameter values (see Figure 13).

Aside from the difference mentioned above, PINT uses a uniform definition of the longitude of ascending node, known as the "KOM" parameter in the DDK binary model (Kopeikin 1995, 1996), which is measured with respect to equatorial north. In TEMPO/TEMPO2, the KOM parameter is defined with respect to the north of the reference frame under which the pulsar position is given (i.e., if the pulsar position is given as an ecliptic coordinate, the KOM parameter is measured from ecliptic north).

4.3. Independence from TEMPO/TEMPO2

One of the motivations of the PINT project is to provide independent (or as independent as is reasonably possible) cross-checks and/or validation of the timing results from other pulsar timing packages. For high-impact precision timing programs, such as gravitational-wave detection efforts, it is critical to compare results from more than a single data analysis pipeline.

Table 6
PINT Fit Parameter vs. the TEMPO2 Parameter

Parameter	V_{T2}[a]	Unit	$V_{T2} - V_P$[b]	$\lvert V_{T2} - V_P \rvert / \sigma_{T2}$[c]	σ_P[d]$/\sigma_{T2}$
F0	277.9377112429746(5)	Hz	-6.661×10^{-16}	0.001	1.000
F1	$-7.33874(5) \times 10^{-16}$	Hz s^{-1}	-8.192×10^{-24}	0.002	1.000
FD1	$4.0(2) \times 10^{-5}$	s	-1.636×10^{-9}	0.001	1.000
FD2	$-1.5(1) \times 10^{-5}$	s	1.416×10^{-9}	0.001	1.000
JUMP	$-8.7887456483 \times 10^{-6}$	s	-4.904×10^{-11}	0.0004[f]	N/A
PX	0.50(7)	mas	1.878×10^{-5}	0.0003	1.000
ELONG	244.347677843(6)	deg	9.123×10^{-12}	0.002	1.000
ELAT	−10.07183905(3)	deg	-1.449×10^{-11}	0.0004	1.000
PMELONG	0.46(1)	mas yr^{-1}	7.420×10^{-6}	0.0007	1.000
PMELAT	−7.16(6)	mas yr^{-1}	-7.171×10^{-5}	0.001	1.000
PB	14.348466(2)	day	-1.924×10^{-09}	0.0009	1.000
A1	8.8016531(8)	lt-s	-8.197×10^{-10}	0.001	1.000
A1DOT	$-4.0(6) \times 10^{-15}$	lt-s s^{-1}	-1.034×10^{-18}	0.002	1.000
ECC	$1.73730(9) \times 10^{-4}$	dimensionless	4.159×10^{-11}	0.005	1.000
T0	55878.2619(5)	day	4.883×10^{-7}	0.0009	1.000
OM	181.84(1)	deg	1.226×10^{-5}	0.0009	1.000
OMDOT	0.005(1)	deg yr^{-1}	-1.229×10^{-6}	0.0009	1.000
M2	0.27(9)	solar mass	1.043×10^{-4}	0.001	1.000
SINI	0.91(3)	dimensionless	-4.278×10^{-5}	0.001	1.000
DMX_0098[e]	0.0013(2)	pc cm^{-3}	-5.204×10^{-7}	0.003	1.000

Notes.
[a] TEMPO2 postfit parameter value.
[b] PINT postfit parameter value.
[c] TEMPO2 postfit parameter uncertainty.
[d] PINT postfit parameter uncertainty.
[e] In the NANOGrav 11 yr data, PSR J1600−3053 has 106 DMX time ranges. Here we only list the DMX parameter with the largest discrepancy between two packages.
[f] Because this version of TEMPO2 did not report the JUMP uncertainty. The relative difference is computed using the PINT fit uncertainty, and the uncertainty division is not applicable.

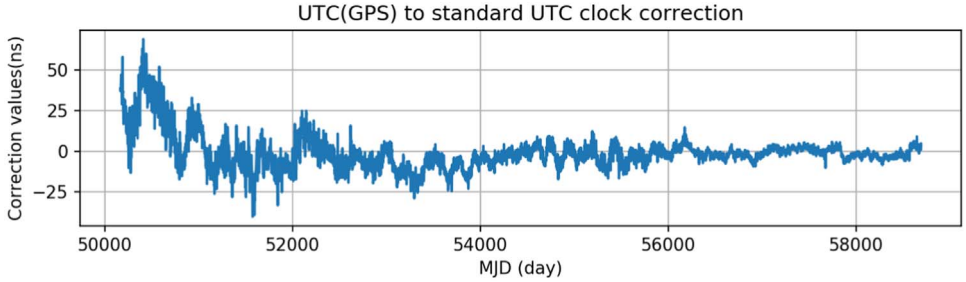

Figure 11. UTC(GPS) and standard UTC clock correction over 20 yr since the GPS timescale was established.

PINT is not a `Python` wrapper of other code, nor is it a `Python` translation of a C or FORTRAN code from previous timing packages. The framework, APIs, and internal data storage are implemented independently. The fundamental algorithms, such as linear algebra, solar system coordinate transformations, and unit conversions, are from widely used and well-tested public `Python` packages (e.g., NUMPY, ASTROPY). PINT's built-in models are implemented based on the physical formulas from their respective publications, and the detailed references are incorporated in the code documentation (e.g., the equation numbers from the papers and necessary derivations are documented in the documentation strings and/or source code). This reimplementation automatically provides a cross-check to the same models as implemented in, for example, TEMPO/TEMPO2. When validating the built-in models, we compare PINT's results (e.g., residual and postfit parameter values and uncertainties, or direct calculations of delay times, for example) with TEMPO/TEMPO2 and attempt to resolve all the discrepancies by auditing both packages' code and their references carefully. This is how we identified implementation differences described in Section 4, as well as long-standing bugs in TEMPO2 related to planetary Shapiro delays[48] and the solar angle calculation.[49] Aside from comparing the same physical model with different implementations, PINT's flexibility, such as being able to call model components from the `Python` command line, enables the user to easily test or compare algorithms and implementations with other versions in PINT or with other software.

Despite these differences in implementation, PINT adopts the most current standard pulsar timing conventions, including data

[48] See https://bitbucket.org/psrsoft/tempo2/issues/63/incorrect-planetary-shapiro-delays.
[49] See https://bitbucket.org/psrsoft/tempo2/issues/68/sign-error-in-solar-angle-calculation.

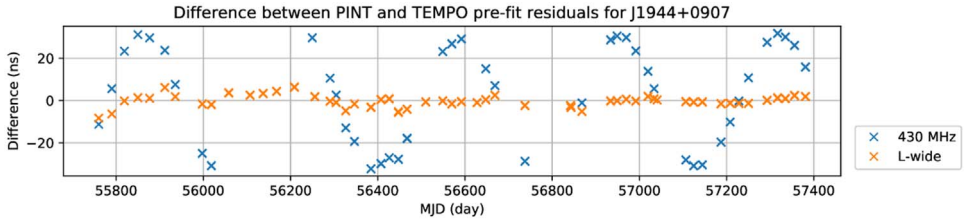

Figure 12. Residual difference between PINT and TEMPO prefit residuals for PSR J1944+0907 NANOGrav 11 yr data. This discrepancy is introduced by different JUMP calculations. Because the JUMPs in TEMPO are applied on the 430 MHz receiver, the annual sinusoid variations only show up for the 430 MHz TOAs.

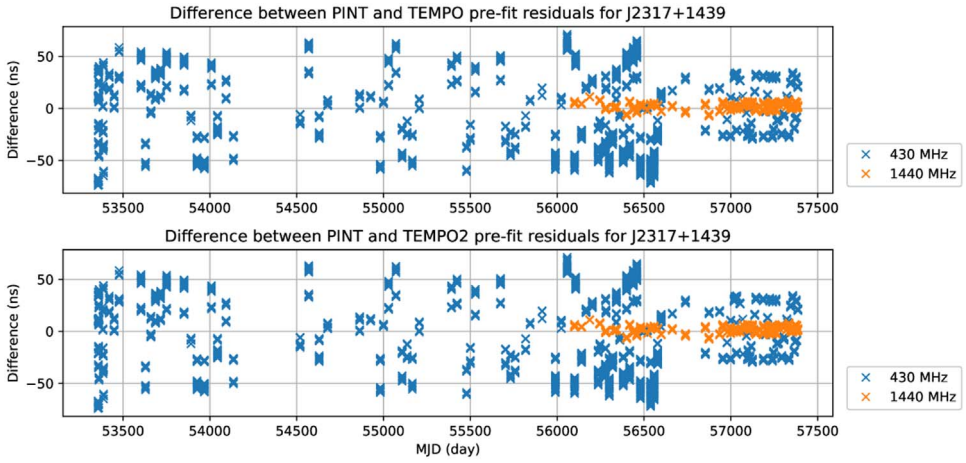

Figure 13. Residual differences between PINT and TEMPO due to a discrepancy in the radio-frequency-dependent delay (FD delay). The first panel illustrates the PSR J2317+1439 NANOGrav 11 yr data PINT-TEMPO residual difference, and the second panel illustrates the PINT-TEMPO2 residual differences for the same data set. In the radio-frequency band, 1440 MHz, residual differences are marked in orange, and the band of 430 MHz residual differences is marked in blue. The 430 MHz shows a higher variation on the difference plot. Because the FD delay is higher at the lower frequency band, this leads to bigger discrepancies in the binary delay input TOAs. Because TEMPO and TEMPO2 both apply the FD delay before the binary correction, these two results are very similar, so that both panels show almost identical plots.

formats and the use of external data (e.g., the JPL solar system ephemerides and standard clock correction files). PINT supports most TEMPO/TEMPO2-accepted styles of TOA and parameter files, and attempts to provide as much backwards compatibility as is reasonably possible. This allows users to cross-check or reproduce earlier results without changing their input data formats. There are plans to include additional compatibility options in future releases of PINT, such as timing using the INPOP solar system ephemerides series[50] (Fienga et al. 2019) or with reference to TCB rather than TDB time.

5. Performance, Testing, and Maintenance

The PINT project's goal is to provide a high-precision, reliable, relatively efficient (i.e., fast), and user-friendly software package. To achieve this goal, we require a comprehensive test suite, profiling, effective version control and other development practices, and good documentation. In this section, we discuss the PINT's performance, testing, and maintenance in detail.

5.1. Performance

Compared to compiled languages, one potential drawback of using a high-level interpreted language like Python is execution speed. In particular, there is a substantial startup cost for a Python script as all the necessary packages are imported, and portions of the code that do a lot of looping and object creation are slower than for compiled languages. However, PINT makes use of highly optimized vectorized code from NUMPY and SCIPY for array and linear algebra operations, and can save intermediate results, such as the TOA table as a Python "pickle" file, which can be loaded very quickly. Thus, the relative performance depends on the particular problem and how PINT is used. In this subsection, we report the PINT runtime for a typical use case of loading a model and TOAs and fitting and comparing it with that of TEMPO and TEMPO2. We chose two test cases: (1) a simple timing model for PSR NGC6440E, which includes astrometry, dispersion, and spindown components, comprising five free parameters, and (2) a more complex timing model for PSR J1910+1256 from the NANOGrav 12.5 yr data set, with 13 model components and 103 free parameters. These were run on Intel(R) Core(TM) i7-10510U CPU @ 1.80 GHz, Ubuntu 20.04.1 LTS VM with 8 GB RAM. Different computers and software libraries will give different results. The script used to generate these tables is available in the PINT GitHub repository.[51]

Table 7 lists the runtime of PINT and TEMPO/TEMPO2 for the case of PSR NGC6440E (with the same timing model and the same fitter as the code example in Figure 3) with different numbers of simulated TOAs. Given the efficiency of FORTRAN and C/C++, TEMPO and TEMPO2 are faster and more RAM efficient than PINT for small problems dominated by reading TOAs from text files and doing preprocessing (applying clock corrections, and computing positions and

[50] https://www.imcce.fr/recherche/equipes/asd/inpop/

[51] See the Python notebook at https://github.com/nanograv/PINT/tree/master/profiling/paper_timing_tables.

Table 7
Performance Comparison between PINT, TEMPO, and TEMPO2 for a Simple Model[a]

Number of TOAs	TEMPO (s)	TEMPO 2 (s)	PINT No Pickling (s)	PINT Using Pickle (s)
100	0.250	1.194	2.174	1.894
1,000	0.288	1.320	3.346	1.954
10,000	0.426	1.680	17.020	3.054
100,000	1.972	6.370	151.170	12.734

Note.
[a] Averaged over five runs.

velocities of the observatory and solar system bodies). The PINT `TOAs` object's pickling functionality allows users to read in TOAs and process them once, save the results to a binary file, and then perform multiple fits or other operations. Table 8 shows the breakdown of the PINT runtime for different parts of the problem. Reading from TOA object pickle files is 10–50 times faster than parsing the TOA text files.

For the case of PSR J1910+1256 with the complicated timing model, we use the NANOGrav 12.5 yr data set's TOAs (5012 TOAs in total) and timing parameters (103 free parameters). We fit data using GLS fitting with noise parameters. To test the speed of a large number of TOAs within the modeled time span, we duplicated the TOAs two and five times. As seen in Table 9, the GLS fitting in TEMPO/TEMPO2, coupled with a more complex model, can increase runtime significantly. When using the GLS fitter, the execution time will depend on the linear algebra libraries (i.e., LAPACK) installed and the configuration of the respective software packages. In the case of large numbers of TOAs, PINT GLS fitting outperforms TEMPO/TEMPO2. This could be due to different linear algebra libraries or different implementations of the GLS fitting algorithm in these packages.

To aid current and future optimization efforts, PINT comes with a folder of profiling code, allowing users and developers to see both a general summary and a detailed report of how long it takes PINT to perform tasks. These files make use of cProfile, `Python`'s built-in profiling tool. Users and developers can produce flowcharts to visualize where PINT spent the most time and find bottlenecks in the code. An HTML viewer (independent of PINT and cProfile) for the cProfile output is also available, allowing the user to click into a function and see the subsequent functions called. Thus, the user can find the root function consuming the most time, or a function taking an unexpectedly long time, and optimize the embedded code. It is our hope that with these features, PINT will become faster as more and more people use the profiling features. The authors themselves have been able to reduce certain benchmark speeds by over 15% using these features.

5.2. Testing

PINT provides various scripts for testing the package, most of which are systematically executed before incorporating any change into the code base. The aim of this testing is to ensure reliability and reproducibility, and ensure that code changes or updates to external packages do not introduce errors. As of version 0.8.0, 58.05% of the code is executed during these tests, and increasing this fraction, as well as ensuring that tests check essential properties, is a goal for future releases. In the development process, providing testing code for new features is strongly encouraged for code contributions to be merged into the main code base. In order to maintain the package's stability and compatibility, the PINT project has adopted the online and offline testing tools `pytest`,[52] `hypothesis`,[53] `GitHub Actions`,[54] and `tox`.[55] These tools execute our tests on major Unix-based operating systems with different `Python` versions.

Table 8
PINT Timing Breakdown[a,b]

Number of TOAs	Import Statements (s)	Loading TOAs No Pickling (s)	Loading TOAs Using Pickle (s)	Fitting WLSFitter (s)
100	1.476	0.471	0.010	0.120
1,000	1.476	2.098	0.096	0.143
10,000	1.476	14.961	1.037	0.432
100,000	1.476	162.165	12.332	2.818

Notes.
[a] These times were recorded separately from the runs in Table 7, and there are additional, smaller operations not displayed. Thus, there may be small disparities in timing between the summation of these individual parts and the total runtime recorded in Table 7.
[b] Averaged over five runs.

Table 9
Complex Model for PSR J1910+1256 Performance Comparison between PINT, TEMPO, and TEMPO2[a,b]

Number of TOAs	TEMPO (s)	TEMPO2 (s)	PINT No Pickling (s)	PINT Using Pickling (s)
5012	32.644	24.630	42.636	35.972
10,024	249.492	52.394	60.458	47.206
25,060	3695.400	211.972	119.190	79.730

Notes.
[a] GLSFitter is used for the above runs.
[b] Averaged over five runs.

5.3. PINT Maintenance

Following the design philosophy of "for and by the user," the PINT software package is an open-source project under the BSD three-clause license.[56] A user can develop and modify PINT software freely as long as the copyrights are recognized.

Because PINT is an ongoing development project, it adopts a modern version control scheme using `git` and GitHub.[57] The GitHub page (https://github.com/nanograv/PINT) is where the PINT software official versions are released and where a user can communicate with the development team, open issues, and propose changes through pull requests. The PINT user manual can be found at the link above as well. We encourage the user community to contribute to the PINT project by submitting pull requests and reporting issues.

[52] https://docs.pytest.org/en/latest/
[53] https://github.com/HypothesisWorks/hypothesis
[54] https://github.com/features/actions
[55] https://tox.readthedocs.io/en/latest/
[56] https://github.com/nanograv/PINT/blob/master/LICENSE.md
[57] https://git-scm.com/, https://github.com/

The documentation is compiled in Restructured Text format using standalone text files and the document strings inside the Python code, using Sphinx.[58] Each time a change is merged into the master branch, the documentation is deployed to readthedocs.io, where it is automatically compiled and made available as a website (https://nanograv-pint.readthedocs.io).

6. Example PINT Use Cases

Fundamentally, PINT is a Python library that users can employ to do pulsar timing calculations in Python scripts or Jupyter[59] notebooks of their own creation. As such, PINT is now included as a dependency in other Python timing libraries (e.g., NANOGrav's enterprise[60]; stingray[61]; HENDRICS[62]).

However, several common use cases have been implemented as command-line Python scripts that are distributed with PINT, serving as examples and allowing many users to employ PINT without needing to explicitly write Python code:

- pintempo: a command-line script that provides similar functionality to the TEMPO and TEMPO2 programs. It reads a timing model and TOAs from specified files and fits parameters, optionally making a residuals plot.
- pintbary: a simple script for barycentering (i.e., converting to TDB timescale and applying solar system delays) specified times, allowing specification of the observatory and observation frequency.
- pintk: a graphical user interface inspired by the plk plugin for TEMPO2. Users can modify the model and TOAs, perform fits, revert to previous fits, and view the results on a residuals plot with a choice of axes. The interface is highly interactive, and subsets of TOAs can be selected for fitting. In addition, JUMPs and phase wraps can be easily added and removed without changing the parfile or timfile. As an aid for phase connection, pintk can also plot sets of random models with parameters drawn from the covariance matrix of each fit to see how well a model extrapolates across data gaps.
- zima: a script to generate a set of simulated TOAs based on an input timing model.

In addition to these applications, there are also scripts included that are specific to handling high-energy (X-ray, γ-ray) photon data, as described below.

6.1. High-energy Photon Timing

PINT has a number of tools that enable the processing of photon data by treating the arrival time of each photon event as a TOA. These are often from space-borne X-ray and γ-ray telescopes. The biggest difference between these events and traditional TOAs is that they are not expected to have occurred at a fiducial phase; they have some distribution in phase, and the goal of the project may even be to determine whether there is any evidence of phase dependency in this distribution. Moreover, these events are often taken from an observatory that is in orbit and thus not at a fixed ITRF coordinate like a ground-based observatory. PINT's observatory module smoothly handles these cases, as described in Section 3.3.2.

PINT is able to handle events from FITS files that contain unmodified spacecraft times, or those that have been barycentered or geocentered by mission-specific software such as gtbary (Fermi Science Support development Team 2019) or barycorr (NASA High Energy Astrophysics Science Archive Research Center (Heasarc) 2014). For unmodified spacecraft times, the relevant SatelliteObs class is initialized with a (mission-specific) orbit file that contains data on the position of the spacecraft as a function of time. PINT builds a univariate spline interpolator that allows for easy computation of the spacecraft position (and velocity) at the precise time of any photon event. Given this, the rest of the PINT machinery can be used on these data. Such data sets often contain large numbers of events, so this often puts a premium on efficient, vectorized computations, made possible by the NumPy arrays that PINT uses.

Here again, these functions are available for use as Python modules, but several common use cases have been implemented as command-line scripts distributed with PINT:

- photonphase: a code that reads common X-ray event data (e.g., from NICER, XMM-Newton, NuSTAR, RXTE) from FITS files and computes the pulse phase of each event using a provided timing model. The output can be plotted or written back out to a column in a FITS file.
- fermiphase: a code similar to photonphase that is specific to Fermi γ-ray data. One addition is the ability to handle photon weights.
- event_optimize: a code that demonstrates fitting a pulsar timing model to photon data, using PINT to compute model phases and emcee (Foreman-Mackey et al. 2013) to perform an MCMC maximum-likelihood optimization.

The NuSTAR team is using PINT for their new clock correction pipeline (Bachetti et al. 2021). Recently, the Very-High-Energy (VHE) γ-ray community has been investigating the use of PINT as part of their processing pipelines. Their data are photon events from ground-based observatories.

7. Conclusion and Discussion

High-precision pulsar timing experiments, including ground-based and space-based projects, are now monitoring a large number of pulsars regularly (for example, NANOGrav monitored 45 ms pulsars for its 11 yr data release). Around the globe, thousands of precisely measured TOAs are generated using high-sensitivity radio telescopes and their modern receivers and backends (wideband receivers and GPU-based backends, etc.) every year. These efforts aim to detect new, extreme astrophysical signals, like the low-frequency stochastic gravitational-wave background. However, it has been very challenging to analyze these large and intricate data sets and share them between international pulsar timing groups (see, e.g., Verbiest et al. 2016 as each group uses its own tools to record and analyze data). In addition, historical data sets are still very valuable for current and future timing projects (e.g., comparing the differences between instruments). This requires that an analysis pipeline has sufficient backwards compatibility.

We present the PINT software package, which provides a platform to overcome these challenges by using an object-oriented and modular design, adopting well-debugged Python libraries, and incorporating the modern version control tools git and GitHub. The PINT package is capable of processing high-precision pulsar timing data with a numerical precision of

[58] http://www.sphinx-doc.org
[59] http://jupyter.org
[60] https://github.com/nanograv/enterprise
[61] Huppenkothen et al. (2019); github.com/stingraysoftware/stingray.
[62] Bachetti (2018); github.com/stingraysoftware/HENDRICS.

~1 ns and with algorithmic precision of a few nanoseconds or better.

We briefly summarize the code architecture and four core modules: the `toa`, `models`, `fitter`, and `residuals` modules.

1. `toa` module provides the functionality of storing and preprocessing (i.e., applying clock corrections and computing the observatory location and velocity) the TOAs from different observatories.
2. `models` module maintains a set of built-in model components and the public interface class, `TimingModel`, for interacting and organizing the model components. The model component class, `Component`, and its subclasses provide the infrastructure for implementing a new model with minimum effort and for performing pulsar data analysis smoothly.
3. `fitter` module provides the infrastructures for fitting a model to a set of TOAs and allows a new fitting algorithm routine to be implemented without modifying the main code.
4. `residuals` module implements the container class, `Residuals` class, for storing timing residuals and their statistical attributes and methods.

A comparison between PINT and TEMPO/TEMPO2 packages is presented in this paper. After the GLS fitting on the same test data set, PINT's postfit parameters are consistent with the results from TEMPO/TEMPO2, within their TEMPO/TEMPO2 fit uncertainties, and PINT postfit residuals differ from TEMPO and TEMPO2 result at the level of 10 ns and 1 ns, respectively. Some known sources of the discrepancies are described.

We also demonstrate the unique features of PINT. PINT modules and functions are designed as an interactive data analysis platform where the user has access to each step of the internal calculation. Because PINT is a `Python`-based package, importing other packages provided by the `Python` community becomes extremely simple. This innovation creates the possibility for applications or features that are hard to implement with the traditional software packages. Using the modern version control tool `git` and the powerful online interface of GitHub, PINT developers are able to communicate with PINT users and provide technical support. Along with the package, some convenient command-line scripts are also provided for the common use cases. In future releases, the PINT project will keep providing new features and improvements to the code.

This project was initiated and supported by the NANOGrav collaboration, which receives support from NSF Physics Frontiers Center award number 1430284. The National Radio Astronomy Observatory is a facility of the National Science Foundation operated under cooperative agreement by Associated Universities, Inc. Portions of this work performed at NRL were supported by Office of Naval Research 6.1 funding. Student research at NRL was sponsored by the Office of Naval Research NREIP program. S.M.R. is a CIFAR Fellow. R.v.H. was supported by NASA Einstein Fellowship grant PF3-140116.

Software: `Astropy` (Astropy Collaboration et al. 2013), `emcee`, Tempo, Tempo2 (Hobbs et al. 2006), `git`, `NumPy`.

Appendix
Creating a Timing Model Component

PINT is designed to be expandable to incorporate new models and new features, and we encourage users to build custom models that suit their needs. Here, we present both the ingredients needed for a new timing model component and the mechanics of automatic model building. A brief code example is provided in Figure 14 to illustrate how to implement a complete PINT model component that can interact with the `TimingModel` class. Along with the descriptions, a detailed example for composing a model component is included in our online documentation.[63]

A typical timing model component includes three major parts: model parameters (see Appendix A.1 for more details), model functions, and derivative functions. Model parameters, implemented by the `Parameter` class, represent the astrophysical quantities the model depends on, e.g., the pulsar position (`RAJ`, `DECJ`), the dispersion measure (`DM`), and the pulsar pulse frequency (`F0`). The model functions then compute needed outputs, e.g., delay, phase, or noise contributions. The derivatives of the modeled quantities with respect to the parameters are required for many fitting algorithms, and so the derivative functions are provided to compute these.

To allow the `TimingModel`'s high-level methods to collect the result from the model component, two API conventions must be followed: (1) the returned result has to be in the accepted format, and (2) the model function must be registered. For instance, `DelayComponent` must return delays as an `astropy.units.quantity` object with time units. This allows `TimingModel.delay()` to sum all of the delays correctly without explicit unit conventions needing to be followed in the code. For `PhaseComponent`, the final result should be a `pint.phase.Phase` object, which represents pulse phase at the required precision. In addition, the model functions must be added to the appropriate function lists. The `TimingModel` computes the modeled quantity by sequentially summing the results of the functions in these lists. Taking the same example, the delay/phase model functions should be added to `.delay_funcs_component` or `.phase_funcs_component` lists in the `delayComponent` or `phaseComponent` classes, respectively.

The model component class is also responsible for providing derivative functions with respect to the parameters. To enable the `TimingModel` class to compute the derivatives using high-level wrapper functions, `d_delay_d_param()` and `d_phase_d_param()`, for example, PINT implements a registration scheme for derivative functions. This scheme requires all derivative functions to follow a consistent API; that is, these functions should have specific input arguments and return values, e.g., the phase derivatives should have the TOA table, parameter name, and total delay as the input arguments. When setting up a model component, derivative functions should be registered using the `Component.register_deriv_funcs()` class method which maps the parameter to its derivatives. The `TimingModel` class computes the derivatives by enumerating the derivative functions with respect to the target parameter from all the model components and then summing the result from these derivative functions. Users are encouraged to provide accurate derivative functions;

[63] https://nanograv-pint.readthedocs.io/en/latest/examples/How_to_build_a_timing_model_component.html

```python
import numpy as np
import astropy.units as u
from pint.models.timing_model import TimingModel, Component, PhaseComponent
import pint.models.parameter as p


class PeriodSpindown(PhaseComponent):
    """This is a simple  model component of pular spindown using spin period."""
    register = True  # Flags for the model builder to find this component.
    category = "spindown" # Give a category for the component sorting.
    def __init__(self):
        # Get the attruibutes that initilzed in the parent class
        super().__init__()
        # Add parameters using the add_params in the TimingModel
        # Add spin period as parameter
        self.add_param(p.floatParameter(name="P0", value=None, units=u.s,
            description="Spin period", longdouble=True))
        # Add spin period derivative P1, and default value to 0.0
        self.add_param(p.floatParameter(name="P1", value=0.0, units=u.s / u.s,
            description="Spin period derivative", longdouble=True))
        # Add reference epoch time.
        self.add_param(p.MJDParameter(name="PEPOCH_P0",  time_scale="tdb",
                description="Reference epoch for spin-down"))
        # Add spindown phase model function to phase functions.
        self.phase_funcs_component += [self.spindown_phase_period]
        # Add the d_phase_d_delay derivative to the list.
        self.phase_derivs_wrt_delay += [self.d_spindown_phase_period_d_delay]
        # Setup the unique parameters for the component.
        self.set_special_params(['P0', 'P1'])

    def setup(self):
        """Setup the model. Register the derivative functions"""
        super().setup()  # This will run the setup in the Component class.
        # Resgister the derivative functions to the timingmodel.
        self.register_deriv_funcs(self.d_phase_d_P0, "P0")
        self.register_deriv_funcs(self.d_phase_d_P1, "P1")

    def validate(self):
        """Check the parameter value."""
        super().validate()  # This will run the parent class .validate()
        # Check required parameters.
        for param in ["P0"]:
            if getattr(self, param) is None:
                raise ValueError("Spindown period model needs {}".format(param))

    # One can always setup properties for updating attributes automatically.
    @property
    def F0(self):
        # We return F0 as a parameter object, which are used in the TimingModel
        return p.floatParameter(name="F0", value=1.0 / self.P0.quantity,
            units="Hz", description="Spin-frequency", long_double=True)

    # Defining the derivatives, a common format is d_xxx_d_xxxx
    @property
    def d_F0_d_P0(self):
        return -1.0 / self.P0.quantity ** 2

    @property
    def F1(self):
        return p.floatParameter(name="F1", description="Spin down frequency",
            value=self.d_F0_d_P0 * self.P1.quantity, units=u.Hz / u.s, long_double=True)

    @property
    def d_F1_d_P0(self):
        return self.P1.quantity * 2.0 / self.P0.quantity ** 3

    @property
    def d_F1_d_P1(self):
        return self.d_F0_d_P0

    def get_dt(self, toas, delay):
        """dt from the toas to the reference time."""
        # toas.table['tdbld'] stores the tdb time in longdouble.
        return (toas.table["tdbld"] - self.PEPOCH_P0.value) * u.day - delay

    # Defining the phase function, which is added to the self.phase_funcs_component
    def spindown_phase_period(self, toas, delay):
        """Spindown phase using P0 and P1"""
        dt = self.get_dt(toas, delay)
        return self.F0.quantity * dt + 0.5 * self.F1.quantity * dt ** 2

    def d_spindown_phase_period_d_delay(self, toas, delay):
        """This is part of the derivative chain for the parameters in the delay term.
        """
        dt = self.get_dt(toas, delay)
        return -(self.F0.quantity + dt * self.F1.quantity)

    def d_phase_d_P0(self, toas, param, delay):
        dt = self.get_dt(toas, delay)
        return self.d_F0_d_P0 * dt + 0.5 * self.d_F1_d_P0 * dt ** 2

    def d_phase_d_P1(self, toas, param, delay):
        dt = self.get_dt(toas, delay)
        return 0.5 * self.d_F1_d_P1 * dt ** 2
```

Figure 14. Example implementation of a timing model component for pulsar spindown.

Table 10
`Parameter` Class Key Attributes

Attribute	Description
`name`	`Parameter` name
`aliases`	Aliases (alternative names) for the parameter
`units`	Default unit of the parameter
`simple`	Description of the parameter
`quantity`	`Parameter` quantity (with units)
`value`	`Parameter` numerical value in the default unit
`prior`	Prior probability distribution for the parameter
`uncertainty`	Postfit parameter uncertainty (with units)
`uncertainty_value`	`Parameter` uncertainty numerical value in the default unit
`frozen`	Boolean flag for turning on/off fitting of the `Parameter`

fitters that depend on these derivatives may fail completely or converge very slowly if they are wrong or inaccurate. Other fitters, like those based on MCMC algorithms, may not use the derivatives at all but often run much more slowly. However, if analytic derivatives are not provided, approximate derivatives can be obtained automatically by numerical methods in `TimingModel.d_delay_d_param_num()` or `TimingModel.d_phase_d_param_num()` with appropriate differential steps. In the case of phase derivatives, `d_phase_d_param()` also applies the derivative chain rule, i.e., the phase is first differentiated with respect to delay, and then multiplied by the derivative of the delay with respect to the parameter. If applicable, the phase derivative with respect to delays should be provided in the phase component.

A.1. Parameter Module

Information about the parameters of a timing model is stored in instances of the `Parameter` class and its subclasses defined in the `models.parameter` submodule. These collect all information relevant to a specific model parameter, including its value, uncertainty, units, and description (see Table 10 for a list of key attributes). There is a profusion of subclasses of `Parameter` in order to handle the variety of different types and formats that parameters can have (for example, strings, right ascensions, floating point) and also to handle extensible families of parameters like the pulse frequency derivatives F0, F1, …, or like JUMP parameters that select subsets of the arrival time measurements to apply time delays to.

One of the innovative features of the `Parameter` class is programmatic integration between a parameter's value and its units. The `.quantity` attribute saves the parameter value as an `astropy.unit.Quantity` object, or compatible type of object (e.g., `astropy.time.Time`), which contains the physical units and allows automatic unit conversions when performing arithmetic with other quantities. This feature avoids confusion and errors arising from unit conversions having to be manually implemented in the code. Each parameter's uncertainty is saved in the `.uncertainty` attribute using the same scheme. For calculations that do not require unit information, the raw numerical parameter and uncertainty values can still be accessed via the `.value` and `.uncertainty_value` properties; these are always guaranteed to return the numerical value in the units specified in the `.units` attribute. The parameter value and uncertainty can be changed by setting the `.quantity` and `.uncertainty` attribute, with unit conversions handled automatically, or `.value` and `.uncertainty_value`.

To read a parameter's information from a .PAR-style parameter file, the `Parameter` class provides the `.from_parfile_line()` method, which parses the parameter file line that has the matching parameter name. The `Parameter` class also implements the `.as_parfile_line()` method to write a parameter as a .PAR-style string line.

Another advanced feature is that the parameter's prior probability density function can be set at the `.prior` attribute for Bayesian timing parameter estimation (e.g., MCMC fitting; Gregory 2005).

In pulsar timing analysis, timing model parameters are applied to more use cases than typical numerical parameters. For instance, the "BINARY" parameter represents the binary model name as a string. Thus, in PINT, a set of `Parameter` subclasses for different use cases are also implemented. In the section below, the parameter types provided in this release are listed.

`floatParameter`: a parameter type for storing floating-point values. The data are stored as an `astropy.units.quantity` object, and the precision can be either 64 bit float or `np.longdouble`.
`strParameter`: a parameter object to store a string value.
`boolParameter`: a type of parameter object used as Boolean flags. It is able to recognize different expressions of Boolean value (e.g., "Y/N," "YES/NO" or "1/0").
`MJDParameter`: a parameter type created for the MJD time values. In order to keep the precision and allow a convenient timescale transformation, it is stored as the `astropy.time.Time` object.
`AngleParameter`: a parameter type implemented for the astronomical angle parameters (e.g., R.A. or decl.). The parameter value is saved in the `astropy.coordinates.Angle` object which provides angle conversion functions. This object accepts different input angle formats as well (e.g., "hour:minute:second" or "degree:minute:second").
`PrefixParameter`: a parameter type designed for parameters that have the same name prefix but a different suffix, e.g., "DMX_0001," "DMX_0002," etc. Because this object is implemented according to the parameter name, not the value type, it is able to store any other `Parameter` types (e.g., MJDParameter, AngleParameter). These internal parameter types can be specified via its input argument `parameter_type`.
`maskParameter`: This parameter object provides functionality for parameters that apply only to a subset of TOAs (e.g., a JUMP). It accepts different parameter values like the `PrefixParameter` object as well. It is able to handle a parameter that has a key value pair for selecting TOAs (e.g., "`ECORR -f Rcvr1_2_GASP 0.00370`," which applies an ECORR value only to TOAs with a particular flag).

Although the `Parameter` objects introduced above can be initialized and used independently (see the code example in Figure 15), it is recommended to use the `Component.add_param()` class method to add the `Parameter` object into the `Component` object and register it to the parameter name space. This allows the automatic model builder (discussed below in Appendix A.2) to select model components by comparing the parameter names.

```
>>> import pint.models.parameter as p
>>> import astropy.units as u
>>> # Create a new floatParameter type class.
>>> param = p.floatParameter(name="F0", value=0.0, units="Hz",
>>>                          description="Spin-frequency", long_double=True)
>>> # Read parameter from a .par style file line.
>>> param.from_parfile_line("F0  61.485476554000000001 1 1e-12")
True
>>> # Print the parameter information.
>>> print(param)
F0 (Hz) 61.485476554000000001 +/- 1e-12 Hz
>>> # Print the parameter information as parfile style.
>>> param.as_parfile_line()
'F0  61.485476554000000001 1 1e-12\n'
>>> # Access the parameter quantity with unit.
>>> param.quantity
<Quantity 61.485476554000000001 Hz>
>>> # Access the parameter unit
>>> param.units
Unit("Hz")
>>> # Access the parameter pure value, without unit.
>>> param.value
61.485476554000000001
>>> # The parameter value can be changed via .quantity or .value
>>> param.quantity = 120.0 * u.Hz
>>> print(param.quantity, param.value)
(<Quantity 120.0 Hz>, 120.0)
>>> param.value = 100.0
>>> print(param.quantity, param.value)
(<Quantity 100.0 Hz>, 100.0)
>>> # Access the parameter uncertainty.
>>> param.uncertainty
<Quantity 1e-12 Hz>
>>> # Check if the parameter fittable or not
>>> param.frozen
False
>>> # This is a fittable parameter.
```

Figure 15. Code example for the `Parameter` module.

A.2. Connecting Components to the `TimingModel`

In order to properly instantiate the various timing model components, including, for example, properly registering the partial derivative functions used by PINT for fitting, a user will typically use the `get_model()` function (introduced in Section 2.2), which utilizes the `model_builder` module and associated `ModelBuilder` class behind the scenes. The `model_builder` selects the correct model components and sorts them into a preferred order and reads the input parameter values. The `model_builder` searches for all registered model components, whose attribute `.register` is set to be True, as demonstrated in the code example in Figure 14 (see line 10). After listing all of the components, it compares each component's parameters with the parameters in the .PAR file and selects matching components. However, this method has two challenges that could lead to an incorrect model selection: (1) the same astrophysical effect can be modeled using different parameterizations (e.g., the DM variation can be modeled by a Taylor expansion or a set of discrete DM values); (2) different components may share a set of common parameters (e.g., some more complicated components are derived from simple components). To help the `model_builder` filter the components, PINT implements a component category system and a special parameter identifier. `model_builder` reads the component's category from the component attribute `.category`, and only one component from the same category will be selected. For instance, even though PINT has five built-in model components in the $\text{pulsar}_{\text{system}}$ category, a timing model can only make use of only one pulsar binary component. As of PINT 0.8.0, we classify all the components in the categories listed in Table 3. Each model component specifies its unique parameters in the `.component_special_params` attribute, and the `model_builder` will first check if these unique parameters are specified in the .PAR file. In the end, the selected components are sorted by category, and model parameter values are read in.

ORCID iDs

Jing Luo https://orcid.org/0000-0001-5373-5914
Scott Ransom https://orcid.org/0000-0001-5799-9714
Paul Demorest https://orcid.org/0000-0002-6664-965X
Paul S. Ray https://orcid.org/0000-0002-5297-5278
Anne Archibald https://orcid.org/0000-0003-0638-3340
Matthew Kerr https://orcid.org/0000-0002-0893-4073
Ross J. Jennings https://orcid.org/0000-0003-1082-2342
Matteo Bachetti https://orcid.org/0000-0002-4576-9337
Rutger van Haasteren https://orcid.org/0000-0002-6428-2620
Camryn Phillips https://orcid.org/0000-0002-2099-0254
Kevin Stovall https://orcid.org/0000-0002-7261-594X
Michael T. Lam https://orcid.org/0000-0003-0721-651X

References

Alam, M. F., Arzoumanian, Z., Baker, P. T., et al. 2021a, ApJS, 252, 4
Alam, M. F., Arzoumanian, Z., Baker, P. T., et al. 2021b, ApJS, 252, 5
Altamimi, Z., Collilieux, X., & Métivier, L. 2011, JGeod, 85, 457
Antoniadis, J., Freire, P. C. C., Wex, N., et al. 2013, Sci, 340, 448
Archibald, A. M., Gusinskaia, N. V., Hessels, J. W. T., et al. 2018, Natur, 559, 73
Arzoumanian, Z., Brazier, A., Burke-Spolaor, S., et al. 2018, ApJS, 235, 37
Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013, A&A, 558, A33
Atwood, W. B., Abdo, A. A., Ackermann, M., et al. 2009, ApJ, 697, 1071
Bachetti, M. 2018, HENDRICS: High ENergy Data Reduction Interface from the Command Shell, v0.5, Astrophysics Source Code Library, ascl:1805.019
Bachetti, M., Markwardt, C. B., Grefenstette, B. W., et al. 2021, ApJ, 908, 184
Backer, D. C., & Hellings, R. W. 1986, ARA&A, 24, 537
Backer, D. C., Kulkarni, S. R., Heiles, C., Davis, M. M., & Goss, W. M. 1982, Natur, 300, 615
Bailes, M., Jameson, A., Abbate, F., et al. 2020, PASA, 37, e028
Blandford, R., & Teukolsky, S. A. 1976, ApJ, 205, 580
Cordes, J. M., & Downs, G. S. 1985, ApJS, 59, 343
CRC Handbook 2004, in CRC Handbook of Chemistry and Physics, ed. D. R. Lide (85th ed.; Boca Raton, FL: CRC Press)
Cromartie, H. T., Fonseca, E., Ransom, S. M., et al. 2020, NatAs, 4, 72
Damour, T., & Deruelle, N. 1986, AIHPA, 44, 263
Damour, T., & Taylor, J. H. 1991, ApJ, 366, 501
Davis, J. L., Herring, T. A., Shapiro, I. I., Rogers, A. E. E., & Elgered, G. 1985, RaSc, 20, 1593
Demorest, P. B., Pennucci, T., Ransom, S. M., Roberts, M. S. E., & Hessels, J. W. T. 2010, Natur, 467, 1081
Deng, X. P., Coles, W., Hobbs, G., et al. 2012, MNRAS, 424, 244
Detweiler, S. 1979, ApJ, 234, 1100
Dib, R., Kaspi, V. M., & Gavriil, F. P. 2009, ApJ, 702, 614
Donner, J. Y., Verbiest, J. P. W., Tiburzi, C., et al. 2019, A&A, 624, A22
Edwards, R. T., Hobbs, G. B., & Manchester, R. N. 2006, MNRAS, 372, 1549
Ellis, J. A. 2013, CQGra, 30, 224004
Fairhead, L., & Bretagnon, P. 1990, A&A, 229, 240
Fermi Science Support development Team 2019, Fermitools: Fermi Science Tools, Astrophysics Source Code Library, ascl:1905.011
Fienga, A., Deram, P., Viswanathan, V., et al. 2019, NSTIM, 109
Folkner, W. M., Williams, J. G., Boggs, D. H., Park, R. S., & Kuchynka, P. 2014, IPNPR, 196, 1
Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. 2013, PASP, 125, 306
Foster, R. S., & Backer, D. C. 1990, ApJ, 361, 300
Freire, P. C. C., & Ridolfi, A. 2018, MNRAS, 476, 4794
Freire, P. C. C., & Wex, N. 2010, MNRAS, 409, 199
Gavriil, F. P., Gonzalez, M. E., Gotthelf, E. V., et al. 2008, Sci, 319, 1802
Gendreau, K. C., Arzoumanian, Z., & Okajima, T. 2012, Proc. SPIE, 8443, 844313
Gregory, P. C. 2005, Bayesian Logical Data Analysis for the Physical Sciences: A Comparative Approach with "Mathematica" Support (Cambridge: Cambridge Univ. Press)
Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Natur, 585, 357
Hellings, R. W., & Downs, G. S. 1983, ApJL, 265, L39
Hewish, A., Bell, S. J., Pilkington, J. D. H., Scott, P. F., & Collins, R. A. 1968, Natur, 217, 709
Hobbs, G., Lyne, A. G., & Kramer, M. 2010, MNRAS, 402, 1027
Hobbs, G. B., Edwards, R. T., & Manchester, R. N. 2006, MNRAS, 369, 655
Huppenkothen, D., Bachetti, M., Stevens, A. L., et al. 2019, ApJ, 881, 39
Irwin, A. W., & Fukushima, T. 1999, A&A, 348, 642
Jones, M. L., McLaughlin, M. A., Lam, M. T., et al. 2017, ApJ, 841, 125
Joshi, B. C., Arumugasamy, P., Bagchi, M., et al. 2018, JApA, 39, 51
Kiel, P. D., & Hurley, J. R. 2009, MNRAS, 395, 2326
Kopeikin, S. M. 1995, ApJL, 439, L5
Kopeikin, S. M. 1996, ApJL, 467, L93
Kramer, M., & Champion, D. J. 2013, CQGra, 30, 224009
Kramer, M., Stairs, I. H., Manchester, R. N., et al. 2006, Sci, 314, 97
Lange, C., Camilo, F., Wex, N., et al. 2001, MNRAS, 326, 274
Lee, K. J. 2016, in ASP Conf. Ser. 502, Frontiers in Radio Astronomy and FAST Early Sciences Symp. 2015, ed. L. Qain & D. Li (San Francisco, CA: ASP), 19
Lieske, J. H., Lederle, T., Fricke, W., & Morando, B. 1977, A&A, 58, 1
Lorimer, D. R., & Kramer, M. 2004, Handbook of Pulsar Astronomy, Vol. 4 (Cambridge: Cambridge Univ. Press)
Makishima, K. 2016, PJAB, 92, 135
Manchester, R. N., Hobbs, G., Bailes, M., et al. 2013, PASA, 30, e017
Manchester, R. N., & IPTA 2013, CQGra, 30, 224010
Manchester, R. N., & Taylor, J. H. 1974, ApJL, 191, L63
McCarthy, D. D., & Capitaine, N. 2002, ITN, 29, 9
McLaughlin, M. A. 2013, CQGra, 30, 224008
NASA High Energy Astrophysics Science Archive Research Center (Heasarc) 2014, HEAsoft: Unified Release of FTOOLS and XANADU, Astrophysics Source Code Library, ascl:1408.004
Niell, A. E. 1996, JGR, 101, 3227
Pennucci, T. T. 2019, ApJ, 871, 34
Pletsch, H. J., & Clark, C. J. 2015, ApJ, 807, 18
Ransom, S., Brazier, A., Chatterjee, S., et al. 2019, BAAS, 51, 195
Ransom, S. M., Stairs, I. H., Archibald, A. M., et al. 2014, Natur, 505, 520
Ray, P. S., Kerr, M., Parent, D., et al. 2011, ApJS, 194, 17
Sazhin, M. V. 1978, SvA, 22, 36
Seidelmann, P. K. 1982, CeMec, 27, 79
Shapiro, I. I. 1964, PhRvL, 13, 789
Standish, E. M. 1998, A&A, 336, 381
Taylor, J. H. 1992, RSPTA, 341, 117
Taylor, J. H., & Weisberg, J. M. 1989, ApJ, 345, 434
Taylor, S. R., Vallisneri, M., Ellis, J. A., et al. 2016, ApJL, 819, L6
The Astropy Collaboration, Price-Whelan, A. M., Sipőcz, B. M., et al. 2018, AJ, 156, 123
The NANOGrav Collaboration, Arzoumanian, Z., Brazier, A., et al. 2015, ApJ, 813, 65
van Haasteren, R. 2013, MNRAS, 429, 55
van Haasteren, R., Levin, Y., McDonald, P., & Lu, T. 2009, MNRAS, 395, 1005
Verbiest, J. P. W., Lentati, L., Hobbs, G., et al. 2016, MNRAS, 458, 1267
Verbunt, F., Igoshev, A., & Cator, E. 2017, A&A, 608, A57
Virtanen, P., Gommers, R., Oliphant, T. E., et al. 2020, NatMe, 17, 261