2021

PINT: A Modern Software Package for Pulsar Timing

Jing Luo
The University of Texas Rio Grande Valley

S. M. Ransom

Paul B. Demorest

P. S. Ray

Anne M. Archibald

See next page for additional authors

Authors

Jing Luo, S. M. Ransom, Paul B. Demorest, P. S. Ray, Anne M. Archibald, M. Kerr, Ross J. Jennings, Matteo Bachetti, Rutger van Haasteren, and Fredrick A. Jenet

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, *] Chloe A. Champagne,[11, †] 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
[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, United Kingdom
[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
[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

ABSTRACT

Over the past few decades, the measurement precision of some pulsar-timing experiments has advanced from $\sim10\,\mu$s to $\sim10\,$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 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

Corresponding author: Jing Luo
luojing1211@gmail.com

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, usage, and validation of PINT, and we compare timing results between it and Tempo and Tempo2.

Keywords: pulsars, pulsar timing, pulsar timing software

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 (Archibald et al. 2018; Kramer et al. 2006; Damour & Taylor 1991). 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 $\lesssim 10\,\mathrm{ms}$). Via the long-term observations of high-quality MSPs, whose pulse arrival times can be measured to better than $1\,\mu s$, 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 $10\,\mathrm{ns}$. The North American Nanohertz Observatory for Gravitational Waves (NANOGrav; McLaughlin 2013) 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[1] and the 100-m Robert C. Byrd Green Bank Telescope (GBT) of the Green Bank Observatory[2]. The international effort of pulsar timing for gravitational waves is under the International Pulsar Timing

---

[*] Currently employed at Microsoft Corporation

[†] NREIP Intern at U.S. Naval Research Laboratory

[1] https://naic.edu/index_scientific.php

[2] https://greenbankobservatory.org/telescopes/gbt/

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 the 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[3] and TEMPO2[4] (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, ease of documentation, robust testing, ease of code re-use, 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 extensiblity, allowing us to break or extend 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 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[5] 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[6] has developed a TEMPO/TEMPO2-independent `Python` toolkit — the PINT software package — for high-precision pulsar timing analysis to pre-

---

[3] http://tempo.sourceforge.net
[4] https://bitbucket.org/psrsoft/tempo2
[5] https://git-scm.com/, https://github.com/
[6] Available at https://github.com/nanograv/PINT and https://pypi.org/project/pint-pulsar/

cisions of $\sim 1\,\mathrm{ns}$[7], and including known physical effects with timing amplitudes of $\sim 1\,\mathrm{ns}$ 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[8] (Harris et al. 2020; Oliphant 2015), SciPy[9] (Virtanen et al. 2019), and Astropy[10] (The Astropy Collaboration et al. 2018; Astropy Collaboration et al. 2013a). Because of their large active user and developer community, such packages are improved frequently and tested thoroughly. The dependency on such packages increases 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[11]. In §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 §3. In §4, we present one example of a PINT analysis and compare it with Tempo/Tempo2. The tests and maintenance procedures are discussed in §5. We also introduce common use cases and their command-line scripts in §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 average process, also called "folding", sums the pulse based on their pulse phases (see §2.2.1 for the definition of "pulse phase"), which are 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

---

[7] For most machines on which PINT will be run, that $\sim 1\,\mathrm{ns}$ level of precision is set by the hardware supported 80-bit floating point numbers used for many of the time-based calculations.

[8] http://www.numpy.org/

[9] https://www.scipy.org/

[10] http://www.astropy.org/

[11] https://nanograv-pint.readthedocs.io/en/latest/

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 very accurate computation of a shift, expressed in units of rotational phase from -0.5 to 0.5, of 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[12], 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, *et cetera*.

PINT does not provide functionality for measuring TOAs, that is left for codes specific to particular types of data. But, PINT can be used to compute the pulse phases for data folding or other calculations (e.g., photon phases). For instance, it 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 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 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*

Rotational phase, often referred to as simply phase, describes a pulsar's rotational status in a reference frame that is co-moving 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 is used. There is

---

[12] This has known problems; see §3.1.

some arbitrariness in the definition of phase zero; it is usually defined as the zero in 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.

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

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

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 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 $N(t_{\mathrm{TOA}})$ will be an integer value.

Practically, in order to evaluate Eqn. 1 we must transform our observed TOAs into the pulsar co-moving frame. In the next sub-section §2.2.2, these transformations, including time scale 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,

$$t_{\mathrm{e}} = t_{\mathrm{obs}} - \Delta, \tag{2}$$

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

$$\Delta = \Delta_{\mathrm{A}} + \Delta_{\mathrm{R}\odot} + \Delta_{\mathrm{E}\odot} + \Delta_{\mathrm{S}\odot} + \Delta_{\mathrm{SB}} + \Delta_{\mathrm{fd}} + \Delta_{\mathrm{binary}} + \dots, \tag{3}$$

where we have listed the most common delays in the timing process (e.g., Lorimer & Kramer 2004). The first term Δ_{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, $\Delta_{\mathrm{R}\odot}$, $\Delta_{\mathrm{E}\odot}$, and $\Delta_{\mathrm{S}\odot}$ 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 Δ_{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 since 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 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 Δ_{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, Δ_{binary}, includes the pulsar system's Rømer, Einstein[13], and Shapiro 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, Δ_{binary} needs to be evaluated at a time that needs Δ_{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 the Hobbs et al. (2006) Table 2.

Given the transformation from pulse observed time t_{obs} to pulse emission time (ignoring a constant pulsar system Einstein delay, see footnote 13)

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

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`, `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 §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 to 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

---

[13] 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.

timing model, are introduced,

$$R_{\text{time}} \equiv t_{\text{obs}} - t_{\text{model}} \tag{5}$$

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 §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:

$$R_{\text{phase}} = N(t_{\text{obs}}) - N_i(t_{\text{obs}}) \tag{6}$$

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

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

where $\nu(t_{\text{obs}})$ is the apparent pulsar pulse frequency at t_{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 the each iteration, the previous post-fit 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)[14]. 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 §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[15]. 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 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 §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 were required, for instance that of the pulsar, they could 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, since 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., Right Ascension, Declination, 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 §3.3.2. Coordinate transformations are performed using Astropy routines whose algorithms are provided via the Essential Routines for Fundamental Astronomy package (ERFA), a

---

[14] https://github.com/nanograv/enterprise
[15] Support for `Python` 2.7 was dropped in 2020, in conjunction with many other astronomical `Python` packages (see http://python3statement.org)
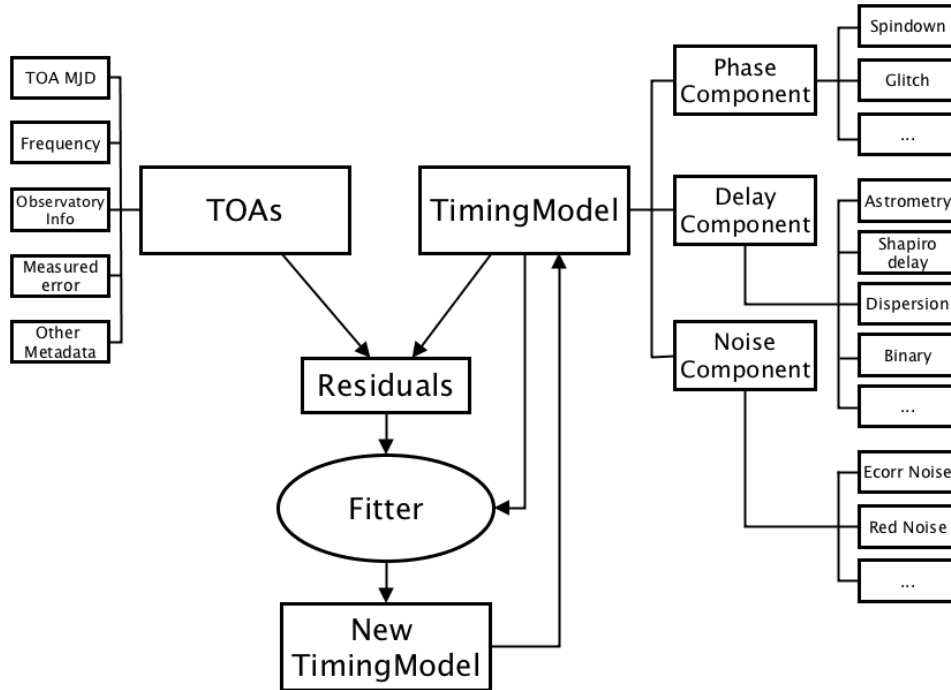
Figure 1. PINT operational model. This is a rough model as to how PINT is designed and implemented as well as how it is used for timing a pulsar. Lines without arrows indicate that the object in question contains the data; arrows indicate that results computed in one object are passed to the other. The 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.

re-branding of the Standards Of Fundamental Astronomy (SOFA) library[16] (Astropy Collaboration et al. 2013b).

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 §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[17], where two 64-bit floats represent the integer and fractional parts of each MJD. Since there is no standard way of representing UTC times on leap days as normal MJDs,[18] 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 86400. The 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.

---

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

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

[18] The precision timing community knows well that using the MJD format for UTC times is fraught with peril. There is no unique way to assign MJDs to times during days with leap seconds, and MJD1−MJD2 does not correctly give the time interval between two times, because of possible leap seconds between MJD1 and MJD2. Nevertheless, MJDs are commonly used for UTC times in many places.

In order to convert TOAs to Barycentric Dynamical Time (TDB), a sequence of clock corrections has to be applied on 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 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[19]. Whether this correction is applied can be controlled via the `observatory` API, which is discussed in §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 86400 seconds 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 seconds. 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[20]). The tick rates of the two differ by about a part in 10^8, so the value of model parameters which have a time component in the unit are different depending on the choice of barycentric timescale. Currently, TEMPO and PINT only

---

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

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 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 since the correction depends on astrometric parameters from the timing model and a solar system ephemeris, not just the TOAs themselves.
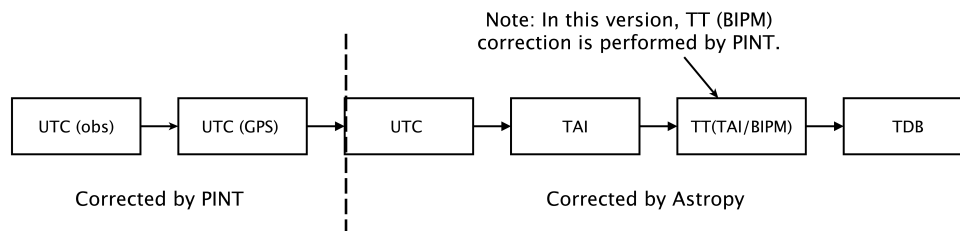

Figure 2. PINT converts TOAs from the observatory local time UTC(obs) to TDB following the steps illustrated. PINT handles the conversion from UTC(obs) to UTC(GPS) and the TT(BIPM) correction. The other part of clock corrections are performed by Astropy.

3.2. PINT *code architecture*

PINT is designed to be highly modular. According to the pulsar timing procedures introduced in §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 which vary timing model parameters to optimally fit the TOAs.

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

^{a}Time of Arrival

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, for instance, 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 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, and 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

```
>>> 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 0.00000003756850254201
  DECJ           -20:21:29.38330660 1 0.00000912542586891696
  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.

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 §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" format TOAs[21]. All 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 de-selected.

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

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. Since 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 §3.3.2. The `toa.get_toas()` method also allows the user to control the version of external data (§3.3.3 discusses the external data handling scheme) used in these wrapped functions via the input arguments. Traditionally, this information are stored in the timing model parameter (.PAR) files, which are processed by `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[22]. 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. As a 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 a `np.longdouble`[23] 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[24] the TOAs and 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 §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

---

[22] 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.

[23] The type `np.longdouble` uses the underlying C implementation's `long double` type. On most Intel machines this is 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.

[24] 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

```
>>> 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

API for obtaining observatory positions and velocities, computing the clock correction values, and calculating time transformations to TDB, with the methods `posvel()`,

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] $\rightarrow$ Observatory position vector	`np.float`	km
ssb_obs_vel	Observatory velocity (referenced to SSB)	`np.float`	km/s
obs_sun_pos	Observatory $\rightarrow$ Sun center position vector	`np.float`	km

[a]Time of Arrival

[b]Barycentric Dynamical Time

[c]Solar System Barycenter

`clock_corrections()`, and `get_TDBs()`, respectively. However, as 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 co-rotating frame to the ICRS frame (i.e., applying the clock corrections and coordinate transformations introduced in §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[25] (ITRF; Altamimi et al. 2011).

In contrast, the `SpecialLocation` class is designed to implement the observatories that are not in a fixed location co-rotating with the Earth, such as the imaginary solar system barycenter (SSB) "observatory" or an Earth-centered "observatory" (i.e., the geocenter). Another use case for the `SpecialLocation` class are space-based observatories such as *Fermi* (Atwood et al. 2009) and *NICER* (Gendreau et al. 2012), where orbital information or other spacecraft flight data is 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 pre-defined 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. 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 provide copies of these data within the packages, and Tempo developers keep them up to date. However, the upstream data are typically updated frequently, meaning that the Tempo developers must often update their packages, and their users must re-install them 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 the earth orientation parameters (i.e., IERS table A and B[26]) and solar system ephemerides, Astropy downloads and caches them when requested. However, due to the upstream issues, 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. Nonetheless, 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 programmaticallly implement a timing model are in Appendix §A. Note that this paper does not discuss the implementation of any specific timing model. For these details, please see the online documentation[27].

Following the philosophy of modularity, PINT implements different physical effects separately as model components, which are implemented independently in the `Component` class and its sub-classes. 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

---

[26] https://datacenter.iers.org/eop.php
[27] https://nanograv-pint.readthedocs.io/en/latest/api/pint.models.timing_model.html

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 sub-class of the base `Component` class; this bas 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 §2.2, modeling TOAs includes two fundamental calculations, total time delay (Δ in Eqs. 2 and 3) and total phase (Eq. 1). PINT therefore implements two explicit `Component` sub-classes, `DelayComponent` and `PhaseComponent`. The `TimingModel` class provides two corresponding methods, `.delay()` and the `.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 fitting and Bayesian timing analyses (van Haasteren 2013; Ellis 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 §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 §A.2. Since version 0.8, PINT also provides a wrapper function, `models.get_model_and_toas()`, that creates the `TimingModel` object and TOA together from the input .par and .tim files and apply the information in the .par file to `TOAs` object. 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.
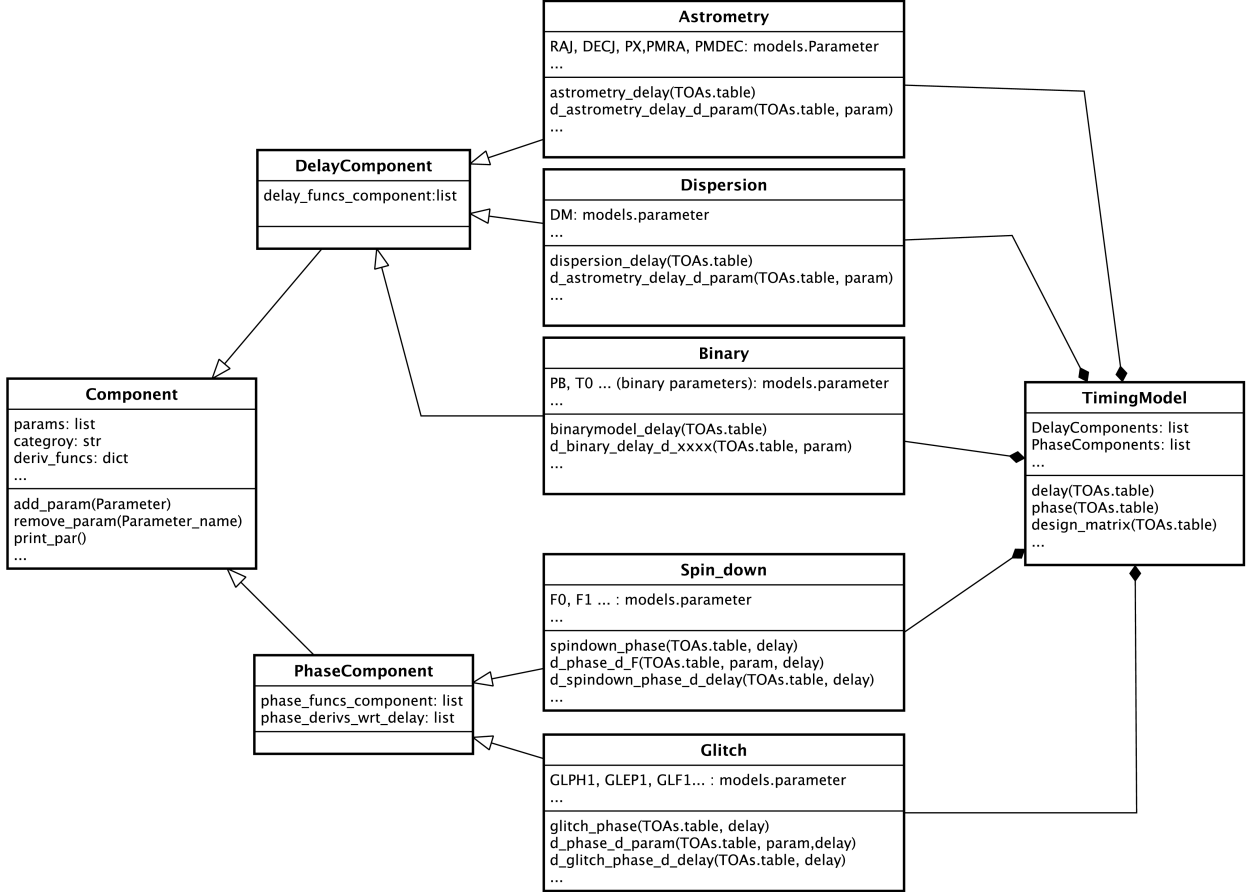
Figure 5. This figure shows an example of how PINT implements a full timing model. Hollow arrows indicate ineritance, while solid arrows indicate containment. `Astrometry`, `Dispersion` and `Binary` classes inherit from the `DelayComponent` class. `Spin_down` and `Glitch` inherit from `PhaseComponent`. Both `DelayComponent` and `PhaseComponent` inherit from the generic `Component` base class. A `TimingModel` instance manages all the specific model components needed to build the full model. Here, we only use `DelayComponent` and `PhaseComponent` as example, if the other component types (e.g., `NoiseComponent`) present, they follow the same relationship structure.

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.

3.5. *Residual Module*

Residuals between the data and timing model are key to updating model parameters and assessing goodness of fit. The `residuals` module is designed to compute the residuals using Eqs. 6 and 7. The interface class, `Residuals`, instantiated by providing the `TOAs` and `TimingModel` instance, implements the `.calc_phase_resids()` method and `.calc_time_resids()` method for computing the phase residuals and time residuals, respectively. For a better representation of the difference between

Table 3. PINT version 0.8.0 built-in TimingModel 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	Powerlaw 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

References—(1)(4)(11) Backer & Hellings (1986); (2)(5)(13)(15)(16)(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);

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 the 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 pre-defined fitter sub-classes implementing specific optimization algorithms. The general API base class `Fitter` sets up framework, and the fitter sub-classes implement the fitting algo-

Table 4. PINT implemented fitting algorithms

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

[a]The independent dispersion measurements are fitted with TOAs simultaneously using generalized least square fitting (Pennucci 2019; Alam et al. 2020a).

rithms 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 the built-in fitters in PINT 0.8.0. Note PINT implements the parameter priors (see A.1) which is used in the MCMC fitter. The constraints of parameters can be performed via the priors. However, the all other fitters do not use this Information other than the MCMC fitter, and the current fitters can not fit for the noise parameter yet. A common package to compute the noise parameter value and parameter priors is `enterprise`.

As described in the code example in Figure 3, a fitter class should be instantiated with `TOAs` object and `TimingModel` object. The `TimingModel` object will be linked to the `fitter.model_init` attribute and an extra copy will be save in the `fitter.model` attribute in order to retain initial timing model data. 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 traced back by the class method `fitter.reset_model()`. Residuals are calculated and saved in the `fitter.resids` attribute, and a copy of initial residuals will be 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 information. The `Fitter` base class already provides a set of convenience functions for this purpose. For example, the `.set_params()` class method is designed for changing parameter values and the `.set_fitparams()` method can be used for selecting the fitting parameters.

As described above the post-fitting results are returned via the `fitter.model` attribute and the `fitter.resids` attribute will be updated to post-fit residuals. This new timing model and residuals are 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-year data release (Arzoumanian et al. 2018). Here we present the results of a PINT analysis of PSR J1600−3053 from the NANOGrav 11-year 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 timespan (8 years). This comparison will also highlight some implementation differences between PINT and Tempo/Tempo2. A full scale PINT-Tempo/Tempo2 comparison using all the pulsars from NANOGrav's 12.5-year data is reported in Alam et al. (2020b)). The Jupyter notebook for this comparison is included in the PINT examples and can be view from the PINT online documentation[28].

4.1. *Comparison using PSR J1600−3053*

We used the published NANOGrav 11-year ephemeris (originally produced using the Tempo software package) as our initial timing model, fitted to TOAs from the NANOGrav 11-year data using the PINT generalized-least-square fitter `pint.fitter.GLSfitter`.

The pre-fit residuals from PINT had a weighted-root-mean-square (WRMS) value of $0.944\,\mu$s. The fitting process reported a final χ^2 value of 12368.10 for 12307 degrees-of-freedom and the post-fit residuals had a WRMS of $0.944\,\mu$s. Figure 6 shows the PINT pre-fit and post-fit 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 12368.46 and the residuals have a WRMS of $0.944\,\mu$s. We directly compared both the pre-fit and post-fit residuals between these two packages. In Figure 7, the residual differences between PINT and Tempo are presented. Note that since we dropped the constant part of absolute phase in our calculation, a constant offset in the residual differences has been ignored.

In the pre-fit 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

---

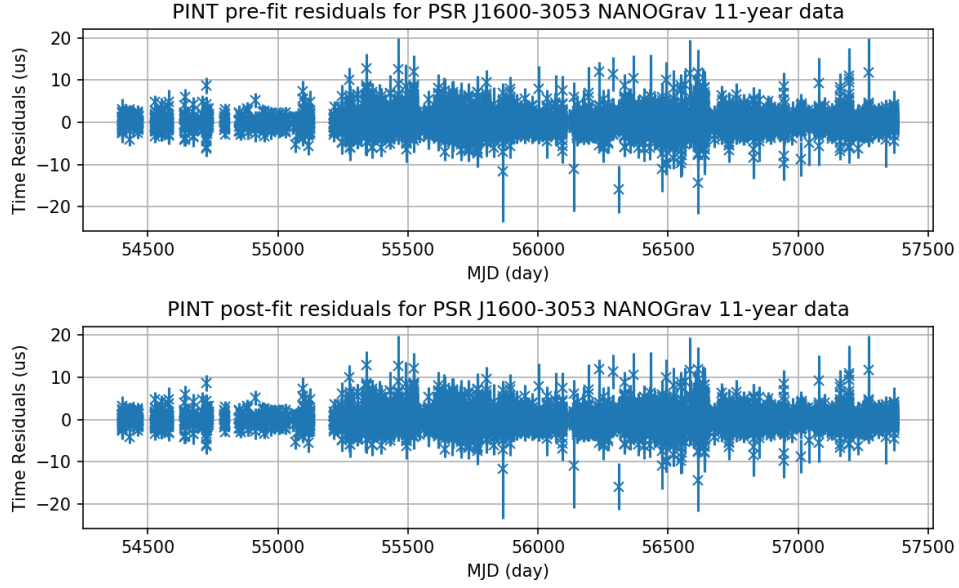[28] https://nanograv-pint.readthedocs.io/en/latest/examples-rendered/paper_validation_example.html

Figure 6. Residuals generated by PINT for PSR J1600−3053 from the NANOGrav 11-year 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.
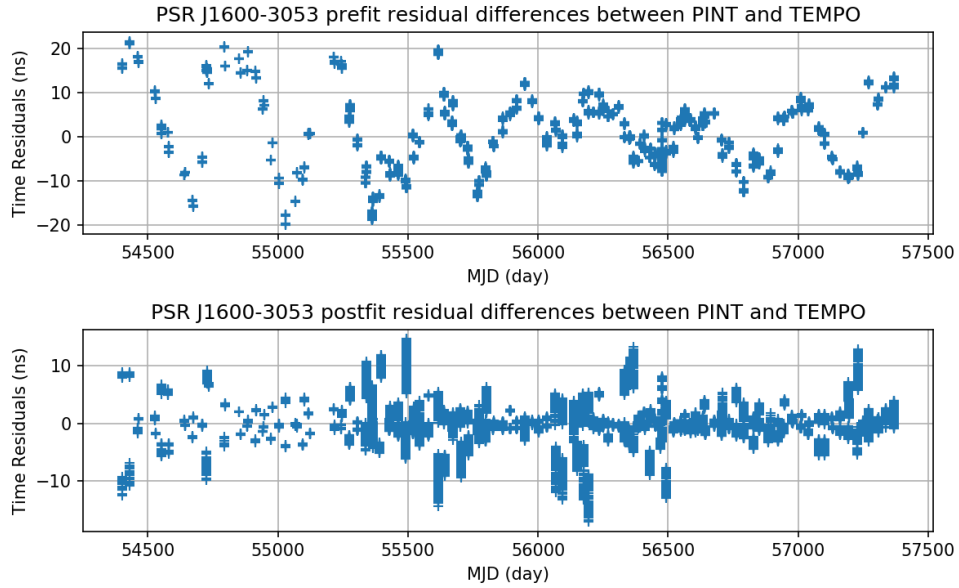

Figure 7. Residual differences between PINT and Tempo for PSR J1600−3053. The upper panel presents the difference of pre-fit residuals and the lower panel presents the post-fit residuals difference.

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.
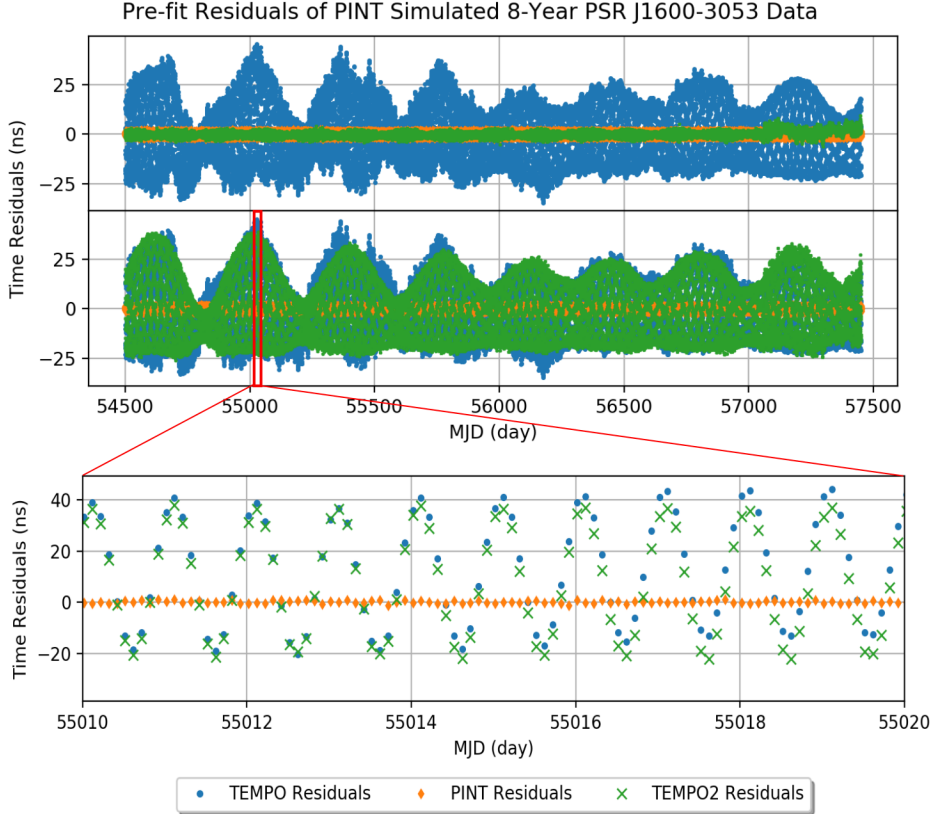
Figure 8. The residual difference due to different precession-nutation models. We use PINT to simulate an 8-year regularly sampled (2.4-hour cadence) TOAs with a simple timing model, only has a constant pulse frequency, and pulsar position. The orange marks represent the PINT residuals, the blue points are the TEMPO residuals, and green data points marks the TEMPO2 residuals. The first panel on the top shows the PINT and TEMPO/TEMPO2 residuals when TEMPO2 is under 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 has a similar signature like 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 TEMPO/TEMPO2 residuals. Given the sampling rate of NANOGrav 11-year data, the TEMPO prefit residual differences in Figure 7 is one trace of the blue dots.

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 the PINT post-fit parameters are consistent with the TEMPO parameter values to well within the 1-σ uncertainties. This shows that PINT is capable of reproducing the published result for PSR J1600−3053 in the NANOGrav 11-year 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-year data set for a more controlled compari-

Table 5. PINT parameter comparison with Tempo for PSR J1600−3053

Parameter	$V_{\rm T}$[a]	Unit	$V_{\rm T} - V_{\rm P}$[b]	$\lvert V_{\rm T} - V_{\rm P}\rvert / \sigma_{\rm T}$[c]	$\sigma_{\rm P}$[d]$/\sigma_{\rm T}$
F0	277.9377112429746(5)	Hz	-1.471×10^{-14}	0.028	1.000
F1	$-7.33874(5) \times 10^{-16}$	Hz/second	6.362×10^{-23}	0.014	1.000
FD1	$4.0(2) \times 10^{-5}$	second	-2.546×10^{-9}	0.002	1.000
FD2	$-1.5(1) \times 10^{-5}$	second	1.370×10^{-9}	0.001	1.000
JUMP	$-8.8(1) \times 10^{-6}$	second	-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/year	7.119×10^{-4}	0.068	1.003
PMELAT	$-7.16(6)$	mas/year	-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)	light-second	1.491×10^{-8}	0.018	0.984
A1DOT	$-4.0(6) \times 10^{-15}$	light-second/second	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/year	-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)	dimension less	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

[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-year 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.

son. The 11-year 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 generalized-least-squares fitting gives a final χ^2 value of 12265.16 and the post-fit residuals have a WRMS of 0.944 μs. Tempo2 residuals were also directly compared against the PINT residuals, and the comparison is shown in the Figure

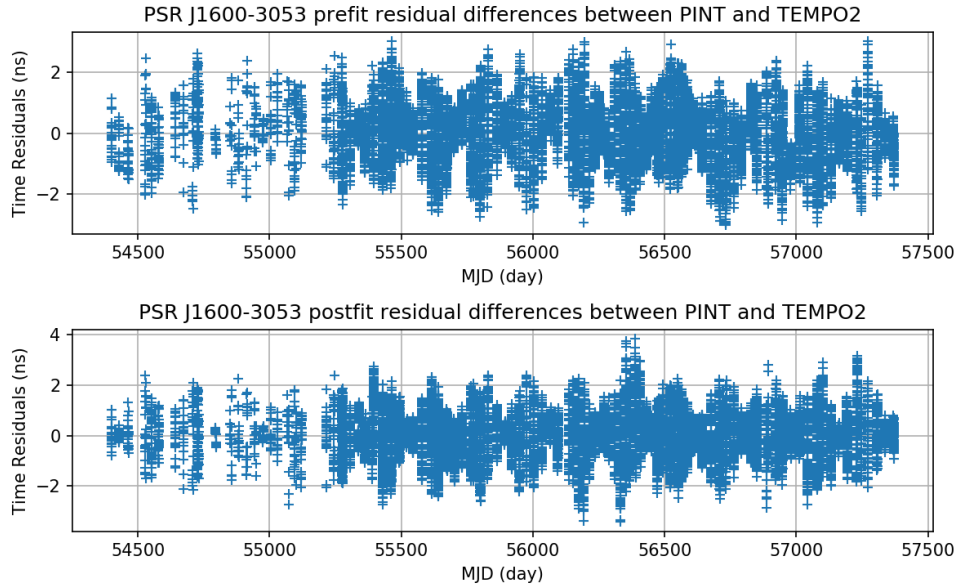9. Again a constant residual offset has been ignored here as well. Both the pre-fit

Figure 9. Residual difference between PINT and TEMPO2 for the J1600−3053 NANOGrav 11-year data. The upper panel shows the pre-fit residual difference and the lower panel shows the post-fit residual difference.

and post-fit residual differences are less than 10 ns, which is within the accuracy goal of TEMPO2 (Hobbs et al. 2006). However, the residual differences show systematic quasi-periodic signature with a semi-annual term that occurs consistently over the whole data set. The same signature presents 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 post-fit 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-year data set.

UTC(GPS) to standard UTC clock conversion (Tempo only): As described in §3.1, PINT converts UTC(GPS) time to the standard UTC timescale. However, the TEMPO package does not apply this 10-nanosecond-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.
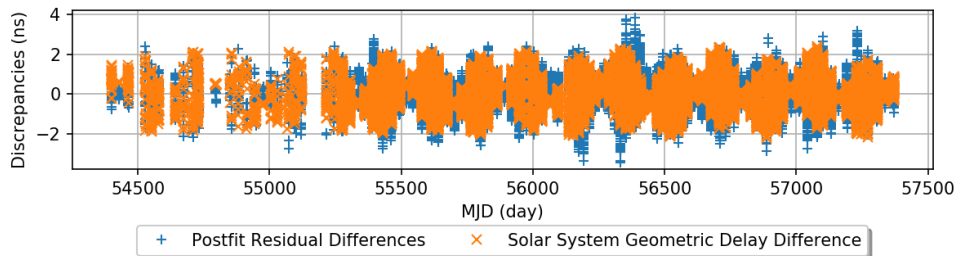
Figure 10. PINT-Tempo2 residual differences and the PINT-Tempo2 solar system geometry delay difference plotted on top of each other. The blue data points mark the difference between PINT and Tempo2 post-fit residuals and the orange points mark the difference between PINT and Tempo2 solar system geometric delay. Their envelopes trace with each other, show that the 2 ns level residual discrepancies are caused by the solar system geometric delay implementation difference of these two softwares.
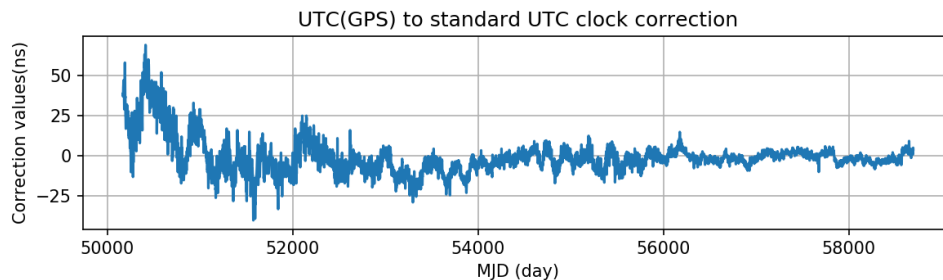

Figure 11. UTC(GPS) and standard UTC clock correction over 20 years since the GPS timescale was established.

Constant time offset between TOAs correction (JUMP): The constant time offset between TOAs, implemented as JUMPs 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). Since the first type of time offset occurs at the observatory, it should be corrected at the observatory frame (before computing the solar system barycentric TOAs). The pulse profile offset is a part of the intrinsic pulsar emission process. Thus, the second type of JUMPs is more appropriately applied under the pulsar frame. However, both Tempo and Tempo2 do not distinguish these two type of JUMPs and correct both of them under the same reference frame. Tempo corrects the JUMPs in either 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 infrastructure to apply the two types of JUMPs separately, and it is planed in the 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

Table 6. PINT fit parameter vs the Tempo2 parameter

Parameter	V_{T2}[a]	Unit	$V_{\mathrm{T2}} - V_{\mathrm{P}}$[b]	$\left| V_{\mathrm{T2}} - V_{\mathrm{P}} \right| / \sigma_{\mathrm{T2}}$[c]	σ_{P}[d]$/\sigma_{\mathrm{T2}}$
F0	277.9377112429746(5)	Hz	-6.661×10^{-16}	0.001	1.000
F1	$-7.33874(5) \times 10^{-16}$	Hz/second	-8.192×10^{-24}	0.002	1.000
FD1	$4.0(2) \times 10^{-5}$	second	-1.636×10^{-9}	0.001	1.000
FD2	$-1.5(1) \times 10^{-5}$	second	1.416×10^{-9}	0.001	1.000
JUMP	$-8.7887456483 \times 10^{-6}$	second	-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/year	7.420×10^{-6}	0.0007	1.000
PMELAT	$-7.16(6)$	mas/year	-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)	light-second	-8.197×10^{-10}	0.001	1.000
A1DOT	$-4.0(6) \times 10^{-15}$	light-second/second	-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/year	-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_0099[e]	0.0017(2)	pc/cm^3	-3.773×10^{-7}	0.002	1.000

[a]Tempo2 post-fit parameter value.

[b]PINT post-fit parameter value.

[c]Tempo2 post-fit parameter uncertainty.

[d]PINT post-fit parameter uncertainty.

[e]In the NANOGrav 11-year data, PSR J1600−3053 has 106 DMX time ranges. Here we only list the DMX parameter with the largest discrepancy between two packages.

[f]Since 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.

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 profiles 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 $\sim 10\,\mathrm{ns}$ level of residual difference, which depends on the FD parameter values (see Figure 13).
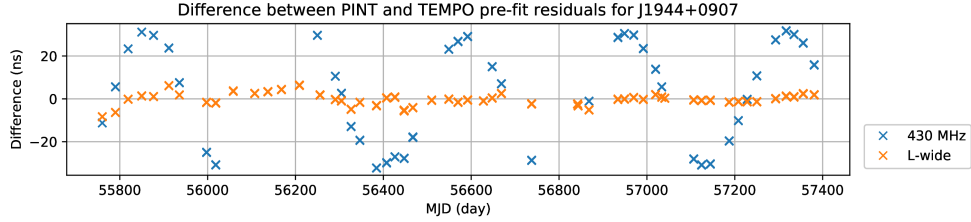
Figure 12. Residual difference between PINT and Tempo pre-fit residuals for PSR J1944+0907 NANOGrav 11-year data. This discrepancy introduced by different JUMP calculations. Since, the JUMPs in Tempo are applied on the 430 MHz receiver, the annual sinusoid variations only show up for the 430 MHz TOAs.
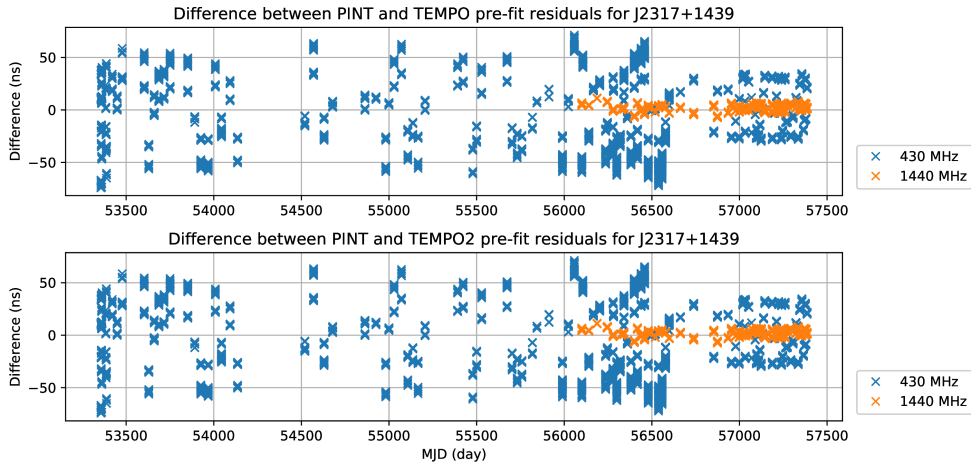

Figure 13. Residual differences between PINT and Tempo due to a discrepancy in the radio Frequency-Dependent delay (FD delay). The first panel illustrates the PSR J2317+1439 NANOGrav 11-year data PINT-Tempo residual difference, and the second panel illustrates the PINT-Tempo2 residual differences for the same data set. The radio frequency band, 1440 MHz, residual differences are marked in orange, and the band of 430 MHz residual differences are 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 a bigger discrepancies in the binary delay input TOAs. Since 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.

Aside from the difference mentioned above, PINT uses a uniform definition of the longitude of ascending node, known as the "KOM" parameter in 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 ecliptic coordinate, 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.

PINT is not a Python wrapper of other code, nor is it a Python translation of 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 re-implementation 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 post-fit 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 §4, as well as long-standing bugs in Tempo2 related to planetary Shapiro delays[29] and the solar angle calculation[30]. 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 most current standard pulsar timing conventions, including data formats and the usage 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 emphemerides series[31] (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.

---

[29] See https://bitbucket.org/psrsoft/tempo2/issues/63/incorrect-planetary-shapiro-delays.
[30] See https://bitbucket.org/psrsoft/tempo2/issues/68/sign-error-in-solar-angle-calculation.
[31] https://www.imcce.fr/recherche/equipes/asd/inpop/

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 TOAs 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 sub-section, we report the PINT run-time for a typical use case of loading a model and TOAs and fitting and compare 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 5 free parameters, and (2) a more complex timing model for PSR J1910+1256 from the NANOGrav 12.5-year data set, with 13 model components and 103 free parameters. These were run on Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, Ubuntu 20.04.1 LTS VM w/ 8GB RAM. Different computers and software libraries will give different results. To see the script used to generate these tables, please visit our GitHub page.[32]

Table 7 lists the run-time 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 Fig. 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 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 run-time for different parts of the problem. Reading from TOA object pickle files is 2–30 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-year data set's TOAs (5012 TOAs in total) and timing parameters (103 free parameters). We fit data using generalized least square (GLS) fitting with noise parameters. To test the speed of a large number of TOAs within the modeled time span, we duplicated the TOAs 2 times and 5 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, 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

---

[32] See https://github.com/nanograv/PINT/tree/master/profiling

Table 7. Performance comparison between PINT, Tempo, and Tempo2 for a simple model[a]

Number of TOAs	Tempo (second)	Tempo 2 (second)	PINT No Pickling (second)	PINT Using Pickle (second)
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

[a]Averaged over five runs.

Table 8. PINT timing breakdown[a,b]

Number of TOAs	Import Statements (second)	Loading TOAs No Pickling (second)	Loading TOAs Using Pickle (second)	Fitting WLSFitter (second)
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

[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.

fitting outperforms Tempo/Tempo2. This could be due to different linear algebra libraries, or different implementations of 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 flow charts 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

Table 9. Complex model for PSR J1910+1256 performance comparison between PINT, Tempo, and Tempo2[a,b]

Number of TOAs	Tempo (second)	Tempo2 (second)	PINT No Pickling (second)	PINT Using Pickling (second)
5,012	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

[a]GLS Fitter is used for above runs.

[b]Averaged over five runs.

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, but PINT code that is never run as part of the test suite is certainly not being checked. As version of 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. For any development and modification, running the test scripts helps detect potential bugs that may break other PINT modules, or, ideally, user code. Thus, providing testing code for new features is strongly encouraged. In order to maintain the package's stability and compatibility, the PINT project has adopted the on-line and off-line testing tools, `pytest`[33], `hypothesis`[34], `GitHub Actions`[35], and `tox`[36]. These tools execute our tests on the major UNIX based operating systems with different `Python` versions.

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 3-clause license[37]. A user can develop and modify PINT software freely as long as the copyrights are recognized.

Since PINT is an ongoing development project, it adopts a modern version control scheme using `git` and GitHub[38]. The GitHub page (https://github.com/nanograv/PINT) is where the PINT software official versions are released and where a user can

---

[33] https://docs.pytest.org/en/latest/
[34] https://github.com/HypothesisWorks/hypothesis
[35] https://github.com/features/actions
[36] https://tox.readthedocs.io/en/latest/
[37] https://opensource.org/licenses/BSD-3-Clause
[38] https://git-scm.com/, https://github.com/

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.

The documentation is compiled in Restructured Text format using standalone text files and the document strings inside the python code, using `Sphinx`[39]. 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[40] notebooks of their own creation. As such, PINT is now included as a dependency in other `Python` timing libraries (e.g. NANOGrav's `enterprise`[41]; `stingray`[42]; `HENDRICS`[43]).

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 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.

---

[39] http://www.sphinx-doc.org
[40] http://jupyter.org
[41] https://github.com/nanograv/enterprise
[42] Huppenkothen et al. (2019); github.com/stingraysoftware/stingray
[43] Bachetti (2018); github.com/stingraysoftware/HENDRICS

6.1. *High Energy Photon Timing*

PINT has a number of tools that enable 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. More, 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 `Observatory` 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 the new clock correction pipeline (Bachetti et al. in prep.). 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 millisecond pulsars for its 11-year 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 their 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 $\sim 1\,$ns and with algorithmic precision of a few ns or better.

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

- `toa` module provides the functionality of storing and pre-processing (i.e., applying clock corrections and computing the observatory location and velocity) the TOAs from different observatories.

- `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 sub-classes provide the infrastructure for implementing a new model with minimum effort and for performing pulsar data analysis smoothly.

- `fitter` module provides the infrastructures for fitting a model to a set of TOAs and allows implementing a new fitting algorithm routine without modifying the main code.

- `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 general-least-square fitting on the same test data set, PINT's post-fit parameters are consistent with the results from TEMPO/TEMPO2, within their TEMPO/TEMPO2 fit uncertainties, and PINT post-fit residuals differ from TEMPO

and TEMPO2 result at the level of $10\,\mathrm{ns}$ and $1\,\mathrm{ns}$, 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 internal calculation. Since 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 of the code.

REFERENCES

Alam, M. F., Arzoumanian, Z., Baker, P. T., et al. 2020a, arXiv e-prints, arXiv:2005.06495. https://arxiv.org/abs/2005.06495

—. 2020b, arXiv e-prints, arXiv:2005.06490. https://arxiv.org/abs/2005.06490

Altamimi, Z., Collilieux, X., & Métivier, L. 2011, Journal of Geodesy, 85, 457, doi: 10.1007/s00190-011-0444-4

Antoniadis, J., Freire, P. C. C., Wex, N., et al. 2013, Science, 340, 448, doi: 10.1126/science.1233232

Archibald, A. M., Gusinskaia, N. V., Hessels, J. W. T., et al. 2018, Nature, 559, 73, doi: 10.1038/s41586-018-0265-1

Arzoumanian, Z., Brazier, A., Burke-Spolaor, S., et al. 2018, ArXiv e-prints. https://arxiv.org/abs/1801.01837

Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013a, A&A, 558, A33, doi: 10.1051/0004-6361/201322068

—. 2013b, A&A, 558, A33, doi: 10.1051/0004-6361/201322068

Atwood, W. B., Abdo, A. A., Ackermann, M., et al. 2009, ApJ, 697, 1071, doi: 10.1088/0004-637X/697/2/1071

Bachetti, M. 2018, Astrophysics Source Code Library, ascl:1805.019

Backer, D. C., & Hellings, R. W. 1986, ARA&A, 24, 537, doi: 10.1146/annurev.aa.24.090186.002541

Backer, D. C., Kulkarni, S. R., Heiles, C., Davis, M. M., & Goss, W. M. 1982, Nature, 300, 615, doi: 10.1038/300615a0

Bailes, M., Jameson, A., Abbate, F., et al. 2020, PASA, 37, e028, doi: 10.1017/pasa.2020.19

Blandford, R., & Teukolsky, S. A. 1976, ApJ, 205, 580, doi: 10.1086/154315

Cordes, J. M., & Downs, G. S. 1985, ApJS, 59, 343, doi: 10.1086/191076

CRC Handbook. 2004, CRC Handbook of Chemistry and Physics, 85th Edition, 85th edn., ed. D. R. Lide (CRC Press)

Cromartie, H. T., Fonseca, E., Ransom, S. M., et al. 2020, Nature Astronomy, 4, 72, doi: 10.1038/s41550-019-0880-2

Damour, T., & Deruelle, N. 1986, Ann. Inst. Henri Poincaré Phys. Théor., Vol. 44, No. 3, p. 263 - 292, 44, 263

Damour, T., & Taylor, J. H. 1991, ApJ, 366, 501, doi: 10.1086/169585

Davis, J. L., Herring, T. A., Shapiro, I. I., Rogers, A. E. E., & Elgered, G. 1985, Radio Science, 20, 1593, doi: 10.1029/RS020i006p01593

Demorest, P. B., Pennucci, T., Ransom, S. M., Roberts, M. S. E., & Hessels, J. W. T. 2010, Nature, 467, 1081, doi: 10.1038/nature09466

Deng, X. P., Coles, W., Hobbs, G., et al. 2012, MNRAS, 424, 244, doi: 10.1111/j.1365-2966.2012.21189.x

Detweiler, S. 1979, ApJ, 234, 1100, doi: 10.1086/157593

Dib, R., Kaspi, V. M., & Gavriil, F. P. 2009, ApJ, 702, 614, doi: 10.1088/0004-637X/702/1/614

Donner, J. Y., Verbiest, J. P. W., Tiburzi, C., et al. 2019, A&A, 624, A22, doi: 10.1051/0004-6361/201834059

Edwards, R. T., Hobbs, G. B., & Manchester, R. N. 2006, MNRAS, 372, 1549, doi: 10.1111/j.1365-2966.2006.10870.x

Ellis, J. A. 2013, Classical and Quantum Gravity, 30, 224004, doi: 10.1088/0264-9381/30/22/224004

Fairhead, L., & Bretagnon, P. 1990, A&A, 229, 240

Fermi Science Support development Team. 2019, Fermitools: Fermi Science Tools. http://ascl.net/1905.011

Fienga, A., Deram, P., Viswanathan, V., et al. 2019, Notes Scientifiques et Techniques de l'Institut de Mecanique Celeste, 109

Folkner, W. M., Williams, J. G., Boggs, D. H., Park, R. S., & Kuchynka, P. 2014, Interplanetary Network Progress Report, 196, 1

Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. 2013, PASP, 125, 306, doi: 10.1086/670067

Foster, R. S., & Backer, D. C. 1990, ApJ, 361, 300, doi: 10.1086/169195

Freire, P. C. C., & Ridolfi, A. 2018, MNRAS, 476, 4794, doi: 10.1093/mnras/sty524

Freire, P. C. C., & Wex, N. 2010, MNRAS, 409, 199, doi: 10.1111/j.1365-2966.2010.17319.x

Gavriil, F. P., Gonzalez, M. E., Gotthelf, E. V., et al. 2008, Science, 319, 1802, doi: 10.1126/science.1153465

Gendreau, K. C., Arzoumanian, Z., & Okajima, T. 2012, in Proc. SPIE, Vol. 8443, Space Telescopes and Instrumentation 2012: Ultraviolet to Gamma Ray, 844313

Gregory, P. C. 2005, Bayesian Logical Data Analysis for the Physical Sciences: A Comparative Approach with 'Mathematica' Support

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Nature, 585, 357, doi: 10.1038/s41586-020-2649-2

Hellings, R. W., & Downs, G. S. 1983, ApJL, 265, L39, doi: 10.1086/183954

Hewish, A., Bell, S. J., Pilkington, J. D. H., Scott, P. F., & Collins, R. A. 1968, Nature, 217, 709, doi: 10.1038/217709a0

Hobbs, G., Lyne, A. G., & Kramer, M. 2010, MNRAS, 402, 1027, doi: 10.1111/j.1365-2966.2009.15938.x

Hobbs, G. B., Edwards, R. T., & Manchester, R. N. 2006, MNRAS, 369, 655, doi: 10.1111/j.1365-2966.2006.10302.x

Huppenkothen, D., Bachetti, M., Stevens, A. L., et al. 2019, ApJ, 881, 39, doi: 10.3847/1538-4357/ab258d

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, doi: 10.3847/1538-4357/aa73df

Joshi, B. C., Arumugasamy, P., Bagchi, M., et al. 2018, Journal of Astrophysics and Astronomy, 39, 51, doi: 10.1007/s12036-018-9549-y

Kiel, P. D., & Hurley, J. R. 2009, MNRAS, 395, 2326, doi: 10.1111/j.1365-2966.2009.14711.x

Kopeikin, S. M. 1995, ApJL, 439, L5, doi: 10.1086/187731

—. 1996, ApJL, 467, L93, doi: 10.1086/310201

Kramer, M., & Champion, D. J. 2013, Classical and Quantum Gravity, 30, 224009, doi: 10.1088/0264-9381/30/22/224009

Kramer, M., Stairs, I. H., Manchester, R. N., et al. 2006, Science, 314, 97, doi: 10.1126/science.1132305

Lange, C., Camilo, F., Wex, N., et al. 2001, MNRAS, 326, 274, doi: 10.1046/j.1365-8711.2001.04606.x

Lee, K. J. 2016, in Astronomical Society of the Pacific Conference Series, Vol. 502, Frontiers in Radio Astronomy and FAST Early Sciences Symposium 2015, ed. L. Qain & D. Li, 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

Makishima, K. 2016, Proceeding of the Japan Academy, Series B, 92, 135, doi: 10.2183/pjab.92.135

Manchester, R. N., & IPTA. 2013, Classical and Quantum Gravity, 30, 224010, doi: 10.1088/0264-9381/30/22/224010

Manchester, R. N., & Taylor, J. H. 1974, ApJL, 191, L63, doi: 10.1086/181549

Manchester, R. N., Hobbs, G., Bailes, M., et al. 2013, PASA, 30, e017, doi: 10.1017/pasa.2012.017

McCarthy, D. D., & Capitaine, N. 2002, IERS Technical Note, 29, 9

McLaughlin, M. A. 2013, Classical and Quantum Gravity, 30, 224008, doi: 10.1088/0264-9381/30/22/224008

Nasa High Energy Astrophysics Science Archive Research Center (Heasarc). 2014, HEAsoft: Unified Release of FTOOLS and XANADU. http://ascl.net/1408.004

Niell, A. E. 1996, Journal of Geophysical Research: Solid Earth, 101, 3227, doi: 10.1029/95JB03048

Oliphant, T. E. 2015, Guide to NumPy, 2nd edn. (USA: CreateSpace Independent Publishing Platform)

Pennucci, T. T. 2019, The Astrophysical Journal, 871, 34, doi: 10.3847/1538-4357/aaf6ef

Pletsch, H. J., & Clark, C. J. 2015, ApJ, 807, 18, doi: 10.1088/0004-637X/807/1/18

Ransom, S. M., Stairs, I. H., Archibald, A. M., et al. 2014, Nature, 505, 520, doi: 10.1038/nature12917

Ray, P. S., Kerr, M., Parent, D., et al. 2011, ApJS, 194, 17, doi: 10.1088/0067-0049/194/2/17

Sazhin, M. V. 1978, Soviet Ast., 22, 36

Seidelmann, P. K. 1982, Celestial Mechanics, 27, 79, doi: 10.1007/BF01228952

Shapiro, I. I. 1964, Physical Review Letters, 13, 789, doi: 10.1103/PhysRevLett.13.789

Standish, E. M. 1998, A&A, 336, 381

Taylor, J. H. 1992, Philosophical Transactions of the Royal Society of London Series A, 341, 117, doi: 10.1098/rsta.1992.0088

Taylor, J. H., & Weisberg, J. M. 1989,
    ApJ, 345, 434, doi: 10.1086/167917

Taylor, S. R., Vallisneri, M., Ellis, J. A.,
    et al. 2016, ApJL, 819, L6,
    doi: 10.3847/2041-8205/819/1/L6

The Astropy Collaboration,
    Price-Whelan, A. M., Sipőcz, B. M.,
    et al. 2018, ArXiv e-prints.
    https://arxiv.org/abs/1801.02634

The NANOGrav Collaboration,
    Arzoumanian, Z., Brazier, A., et al.
    2015, ApJ, 813, 65,
    doi: 10.1088/0004-637X/813/1/65

van Haasteren, R. 2013, MNRAS, 429, 55,
    doi: 10.1093/mnras/sts308

van Haasteren, R., Levin, Y., McDonald,
    P., & Lu, T. 2009, MNRAS, 395, 1005,
    doi: 10.1111/j.1365-2966.2009.14590.x

Verbiest, J. P. W., Lentati, L., Hobbs, G.,
    et al. 2016, MNRAS, 458, 1267,
    doi: 10.1093/mnras/stw347

Verbunt, F., Igoshev, A., & Cator, E.
    2017, A&A, 608, A57,
    doi: 10.1051/0004-6361/201731518

Virtanen, P., Gommers, R., Oliphant,
    T. E., et al. 2019, arXiv e-prints,
    arXiv:1907.10121.
    https://arxiv.org/abs/1907.10121

APPENDIX

A. CREATE A TIMING MODEL COMPONENT

PINT is designed to be expandable to new models and new features. We encourage our users to build custom models for their needs. Here, we present the ingredients of a new timing model component. The mechanics of automatic model building are in this section as well. 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[44].

A typical timing model component includes three major parts, model parameters (see §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 sky locations (`RAJ`, `DECJ`), the dispersion measure (`DM`) and the pulsar pulse frequency (`F0`), etc.). The model functions are where model output quantities (e.g., delay, phase, or noise effects) are computed. The derivatives of 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 to be registered. For instance, `DelayComponent` must return delays as an `astropy.units.quantity` object with time units. This allows `TimingModel.delay()` to sum all 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 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

---

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

```python
1  import numpy as np
2  import astropy.units as u
3  from pint.models.timing_model import TimingModel, Component, PhaseComponent
4  import pint.models.parameter as p
5
6
7  class PeriodSpindown(PhaseComponent):
8      """This is a simple  model component of pular spindown using spin period."""
9      register = True  # Flags for the model builder to find this component.
10     category = "spindown" # Give a category for the component sorting.
11     def __init__(self):
12         # Get the attruibutes that initilzed in the parent class
13         super().__init__()
14         # Add parameters using the add_params in the TimingModel
15         # Add spin period as parameter
16         self.add_param(p.floatParameter(name="P0", value=None, units=u.s,
17             description="Spin period", longdouble=True))
18         # Add spin period derivative P1, and default value to 0.0
19         self.add_param(p.floatParameter(name="P1", value=0.0, units=u.s / u.s,
20             description="Spin period derivative", longdouble=True))
21         # Add reference epoch time.
22         self.add_param(p.MJDParameter(name="PEPOCH_P0",  time_scale="tdb",
23                 description="Reference epoch for spin-down"))
24         # Add spindown phase model function to phase functions.
25         self.phase_funcs_component += [self.spindown_phase_period]
26         # Add the d_phase_d_delay derivative to the list.
27         self.phase_derivs_wrt_delay += [self.d_spindown_phase_period_d_delay]
28         # Setup the unique parameters for the component.
29         self.set_special_params(['P0', 'P1'])
30
31     def setup(self):
32         """Setup the model. Register the derivative functions"""
33         super().setup()  # This will run the setup in the Component class.
34         # Resgister the derivative functions to the timingmodel.
35         self.register_deriv_funcs(self.d_phase_d_P0, "P0")
36         self.register_deriv_funcs(self.d_phase_d_P1, "P1")
37
38     def validate(self):
39         """Check the parameter value."""
40         super().validate()  # This will run the parent class .validate()
41         # Check required parameters.
42         for param in ["P0"]:
43             if getattr(self, param) is None:
44                 raise ValueError("Spindown period model needs {}".format(param))
45
46     # One can always setup properties for updating attributes automatically.
47     @property
48     def F0(self):
49         # We return F0 as a parameter object, which are used in the TimingModel
50         return p.floatParameter(name="F0", value=1.0 / self.P0.quantity,
51             units="Hz", description="Spin-frequency", long_double=True)
52
53     # Defining the derivatives, a common format is d_xxx_d_xxxx
54     @property
55     def d_F0_d_P0(self):
56         return -1.0 / self.P0.quantity ** 2
57
58     @property
59     def F1(self):
60         return p.floatParameter(name="F1", description="Spin down frequency",
61             value=self.d_F0_d_P0 * self.P1.quantity, units=u.Hz / u.s, long_double=True)
62
63     @property
64     def d_F1_d_P0(self):
65         return self.P1.quantity * 2.0 / self.P0.quantity ** 3
66
67     @property
68     def d_F1_d_P1(self):
69         return self.d_F0_d_P0
70
71     def get_dt(self, toas, delay):
72         """dt from the toas to the reference time."""
73         # toas.table['tdbld'] stores the tdb time in longdouble.
74         return (toas.table["tdbld"] - self.PEPOCH_P0.value) * u.day - delay
75
76     # Defining the phase function, which is added to the self.phase_funcs_component
77     def spindown_phase_period(self, toas, delay):
78         """Spindown phase using P0 and P1"""
79         dt = self.get_dt(toas, delay)
80         return self.F0.quantity * dt + 0.5 * self.F1.quantity * dt ** 2
81
82     def d_spindown_phase_period_d_delay(self, toas, delay):
83         """This is part of the derivative chain for the parameters in the delay term.
84         """
85         dt = self.get_dt(toas, delay)
86         return -(self.F0.quantity + dt * self.F1.quantity)
87
88     def d_phase_d_P0(self, toas, param, delay):
89         dt = self.get_dt(toas, delay)
90         return self.d_F0_d_P0 * dt + 0.5 * self.d_F1_d_P0 * dt ** 2
91
92     def d_phase_d_P1(self, toas, param, delay):
93         dt = self.get_dt(toas, delay)
94         return 0.5 * self.d_F1_d_P1 * dt ** 2
```

Figure 14. Example implementation of a timing model component for pulsar spin-down.

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; 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 Markov Chain Monte Carlo algorithms, my 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, the `d_phase_d_param()` also applies the derivative chain rule (i.e., the phase is first differentiated with respect to delay, and then times the delay derivative 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 sub-classes defined in the `models.parameter` sub-module. 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 which 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`.

Table 10. Parameter class key attributes

Attribute	Description
`name`	Parameter name
`aliases`	Aliases (alternative names) for the parameter
`units`	Default unit of the parameter
`description`	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`	Post-fit 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

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., Markov chain Monte Carlo(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` sub-classes 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 the 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 format of Boolean value (e.g., 'Y/N', 'YES/NO' or '1/0')

`MJDParameter:` A parameter type created for the Modified Julian Day 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., Right Ascension or Declination). The parameter value is saved in the `astropy.coordinates.Angle` object which provides angle conversion

functions. This object accepts different input angle format 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" and "DMX_0002"). Since 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`", for example applying 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 §A.2) to select model components by comparing the parameter names.

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 §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 the components, it compares each component's parameters with the parameters in the .par file, and When they are in common, the component is selected. However, this method has two challenges that could lead to a wrong model selection: (1) One astrophysical effect can be modeled using different parametrization (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 a 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

```
>>> 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 parameter module

will be selected. For instance, even PINT has five built-in model components in the *pulsar_system* category, but one timing model can only make use of a 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.

```
>>> from pint.models import get_model
>>> m = get_model("NGC6440E.par")
WARNING: Unrecognized parfile line 'T2CMETHOD  TEMPO' [pint.models.timing_model]

Print out the read in parameters as parfile style
>>> print(m.as_parfile())
PSR                         1748-2021E
EPHEM                             DE421
UNITS                               TDB
RAJ               17:48:52.75000000 1 0.00000010000000000000
DECJ             -20:21:29.00000000 1 0.00010000000000000000
PMRA                                0.0
PMDEC                               0.0
PX                                  0.0
POSEPOCH          53750.000000000000000
F0                        61.485476554 1 1e-11
F1                          -1.181e-15 1 1e-19
PEPOCH            53750.000000000000000
TZRMJD            53801.386051182230000
TZRSITE                               1
TZRFRQ                         1949.609
PLANET_SHAPIRO                        N
NE_SW                               0.0
SWM                                 0.0
DM                              223.9 1 0.03
DM1                                 0.0

Print out the parameters used
>>> print(m.params)
['PSR', 'TRACK', 'EPHEM', 'UNITS', 'F0', 'F1', 'PEPOCH', 'TZRMJD', 'TZRSITE', 'T
ZRFRQ', 'NE_SW', 'SWM', 'POSEPOCH', 'PX', 'RAJ', 'DECJ', 'PMRA', 'PMDEC', 'PLANE
T_SHAPIRO', 'DM', 'DM1', 'DMEPOCH']

Check out the delay and phase functions
>>> m.delay_funcs
[<bound method AstrometryEquatorial.solar_system_geometric_delay of <pint.models
 .astrometry.AstrometryEquatorial object at 0x7f71640552d0>>,
 <bound method SolarSystemShapiro.solar_system_shapiro_delay of <pint.models.sol
 ar_system_shapiro.SolarSystemShapiro object at 0x7f7164055cd0>>,
 <bound method DispersionDM.constant_dispersion_delay of <pint.models.dispersion
 _model.DispersionDM object at 0x7f7164055a50>>,
 <bound method SolarWindDispersion.solar_wind_delay of <pint.models.solar_wind_d
 ispersion.SolarWindDispersion object at 0x7f71640270d0>>]

>>> m.phase_funcs
[<bound method Spindown.spindown_phase of <pint.models.spindown.Spindown object
 at 0x7f716401ef50>>]
```

Figure 16. Code example for Timing Model module