Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

SWIGLAL: Python and Octave interfaces to the LALSuite gravitational-wave data analysis libraries

Karl Wette [*]

ARC Centre of Excellence for Gravitational Wave Discovery (OzGrav) and Centre for Gravitational Astrophysics, Australian National University, Canberra, ACT 2600 Australia
Max Planck Institute for Gravitational Physics (Albert Einstein Institute), D-30167 Hannover, Germany

A R T I C L E   I N F O

Article history:
Received 14 March 2020
Received in revised form 4 October 2020
Accepted 3 November 2020

Keywords:
Gravitational waves
Software wrapper
Python
Octave

A B S T R A C T

The LALSuite data analysis libraries, written in C, implement key routines critical to the successful detection of gravitational waves, such as the template waveforms describing the merger of two black holes or two neutron stars. SWIGLAL is a component of LALSuite which provides interfaces for Python and Octave, making LALSuite routines accessible directly from scripts written in those languages. It has enabled modern gravitational-wave data analysis software, used in the first detection of gravitational waves, to be written in Python, thereby benefiting from its ease of development and rich feature set, while still having access to the computational speed and scientific trustworthiness of the routines provided by LALSuite.

Code metadata

Current code version	6.21.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_119
Legal Code License	GPL-2.0
Code versioning system used	git
Software code languages, tools, and services used	C with SWIG directives, Python, Octave, LALSuite dependencies
Compilation requirements, operating environments & dependencies	SWIG $\geq$ 2.0.12, Python $\geq$ 2.6, NumPy $\geq$ 1.3, Octave $\geq$ 3.2.0
If available Link to developer documentation/manual	See https://github.com/kwwette/swiglal/blob/master/README.md
Support email for questions	See https://github.com/kwwette/swiglal/blob/master/README.md

1. Motivation and significance

The choice of programming language is a critical decision in the design of scientific software. Languages such as C provide a low level of abstraction between the programmer and the machine architecture, and are compiled to machine code for best performance. The lack of abstraction, however, places a higher burden on the developer to manually handle low-level tasks, such as memory management, which detracts from the scientific problem at hand. High-level scripting languages, of which Python [1] and Octave [2] are two examples, provide a higher level of abstraction from the machine architecture, freeing the developer to focus on the algorithm, reducing development time, and facilitating the rapid prototyping of new ideas. They also provide a richer set of features, either built into the language or else available through easy-to-install packages downloaded from a central repository. They are generally not compiled to machine code, however, and therefore performance may not match that provided by low-level machine code.

Often, a new software package will want to make use of existing libraries which provide routines which are particularly efficient, well-tested and trusted by the wider scientific community, and/or difficult to re-implement. In such cases, the developer may be constrained to use a particular language – the same language as the existing library – and therefore be forced to accept the costs and benefits of that particular language. A solution to this problem is to write a software wrapper around the existing library, which then exposes its routines so that it can be used from the programming language of choice. For example, software wrappers can enable the developer to make use of libraries written in C, while also benefiting from the ease of development and rich feature set provided by high-level languages such as Python.

* Correspondence to: ARC Centre of Excellence for Gravitational Wave Discovery (OzGrav) and Centre for Gravitational Astrophysics, Australian National University, Canberra, ACT 2600 Australia.
E-mail address: karl.wette@anu.edu.au.

https://doi.org/10.1016/j.softx.2020.100634

Table 1
Number of constants, variables, functions, and classes exported by the SWIGLAL interfaces to the libraries of LALSuite, version 6.67, and an estimate of the total (non-blank, non-comment) lines of C code (LOC) of each library. Note that SWIGLAL provides a single interface to the LAL and LALSupport libraries, which are therefore counted together.

	Constants	Variables	Functions	Classes	LOC
LAL(Support)	712	64	1745	182	38k
LALBurst	9	4	23	2	2k
LALFrame	54	4	254	12	7k
LALInference	45	41	416	22	28k
LALInspiral	204	5	358	58	33k
LALMetaio	55	4	51	18	3k
LALPulsar	156	9	623	148	32k
LALSimulation	209	42	714	12	92k
Total	1444	173	4184	454	236k

The first detections of gravitational waves, from the merger of two black holes [3] and from two neutron stars [4], were made possible through, amid many other advances, the careful implementation and rigorous testing of data analysis software. LALSuite (LSC Algorithm Library Suite; [5]) is a collection of software routines for gravitational-wave data analysis, written in C, and developed since 2000. As of version 6.67 [6], LALSuite provides, along with $\sim$ 230 executables, 9 libraries which collectively export a large number of symbols, and represents a significant code base of hundreds of thousands of lines of C code (Table 1). It provides atomic data types for fixed-width integer, floating-point, and complex numbers; and compound data types called "structs", accompanied by functions which create, destroy, and manipulate them. (Structs in C are conceptually equivalent to classes in Python and other high-level languages; this paper will hereafter use the term "class" to refer to both low-level structs and high-level classes.)

The LALSuite libraries provide extensive, well-tested routines for gravitational-wave data analysis, in particular for searches for binary black holes and binary neutron stars, which have been carefully vetted by members of the LIGO Scientific Collaboration and Virgo Collaboration. These include the template signal waveforms for such events, as predicted by general relativity, which tend to be complicated mathematical expressions [e.g. 7] which are time-consuming to implement and verify. More recent gravitational-wave data analysis software has sought to take advantage of the ease of development and extensive package library of Python; without access to LALSuite routines, however, developers would have faced a significant additional burden in re-implementing and re-verifying the routines in Python.

This paper describes SWIGLAL, which provides Python and Octave interfaces to the libraries provided by LALSuite. These interfaces have enabled modern gravitational-wave data analysis software to benefit from the advantages of programming in high-level languages, while retaining access to the trusted code base and computational efficiency of the LALSuite code base.

2. Software description and illustrative examples

Generation of the SWIGLAL interface uses SWIG (Simplified Wrapper and Interface Generator; [8]), a software development tool. SWIG parses the header files of a C/C++ library and identifies the symbols the library exports. It then generates the wrapper code required to interface the library with a variety of high-level languages, including Python and Octave. Because it takes C/C++ header files directly as input, SWIG does not require additional code to be written specifically for each exported symbol. Given the large number of symbols exported by LALSuite (Table 1), the automation provided by SWIG relieves LALSuite developers of a significant maintenance burden. SWIG wrapper code can be further customised by adding *directives* which modify the SWIG-generated wrapper code. For example, specific directives can be applied to every class in order to add constructors and destructors. SWIG does not, however, provide a general framework for automating the application of many directives to arbitrary classes of symbols. To fully automate interface generation, SWIGLAL runs SWIG twice: first as a simple C/C++ header parser, then as an wrapper code generator. The workflow is as follows:

1. For each LALSuite library, SWIGLAL generates a basic SWIG interface which simply incorporates all the C header files provided by that library.
2. The basic SWIG interface is input to SWIG with its `-xml` option, which generates an XML file containing a syntax tree of all symbols exported by the LALSuite library headers.
3. The XML syntax tree is input to a custom Python script, `generate_swig_iface.py`. It parses the XML syntax tree, gathers information about the exported symbols, and generates the full SWIG interface, which augments the basic interface with additional SWIG directives to implement desired functionalities.
4. The full SWIG interface is input to SWIG with its `-python` or `-octave` options to generate wrapper code for Python or Octave respectively, which are then compiled into dynamically loadable modules. Python modules are loaded using the **import** directive; Octave modules are loaded by simply calling the name of the library, e.g. "`lal;`" for the LAL library.

The workflow is implemented as a collection of macros and build rules in the GNU Autoconf/Automake build system used by LALSuite. Autoconf macros perform configuration tasks, e.g. finding a compatible version of the `swig` binary, and determining the C/C++ preprocessor/compiler/linker flags needed to build the Python/Octave modules. Automake macros implement the workflow to build the basic and full SWIG interfaces, and the Python/Octave modules, as described above.

A key design objective of SWIGLAL is that the interfaces should resemble and behave, in the supported high-level language, as close to native code written in that language as possible. To that end, SWIGLAL provides a library of custom SWIG directives which modify the wrapper code to mediate between the expected behaviour of native Python/Octave code and the semantics of the C-language LALSuite libraries. The interface file `SWIGCommon.i` provides common directives used in all languages, while `SWIGPython.i` and `SWIGOctave.i` provide directives specific to the Python and Octave interfaces respectively. Each LALSuite library may also provide library-specific directives.

It is also sometimes necessary to add SWIG directives directly to the C header files, in order to further modify the wrapper code for particular functions or classes. SWIGLAL provides numerous macros, defined in `SWIGCommon.i` which are then added to the C header files within **#ifdef** SWIG … **#endif** blocks and wrapped in a common macro, SWIGLAL(); Fig. 1 shows an example usage. This approach keeps SWIG-related code added to the C header files as succinct as possible. Fig. 1 provides an example: the extensive code required to expose the LAL REAL4Vector class as a native scripting-language array is hidden within the ARRAY_STRUCT_1D() macro.

The remainder of this section describes some of the issues encountered in fulfilling the objective of the SWIGLAL interface to closely resemble native Python/Octave code, and how those issues are addressed.

```
typedef struct tagREAL4Vector {
#ifdef SWIG
    SWIGLAL(ARRAY_STRUCT_1D(REAL4Vector, REAL4,
        ↪ data, UINT4, length));
#endif
    UINT4 length;
    REAL4 *data;
} REAL4Vector;
```

Fig. 1. Example usage of the SWIGLAL() macro in the wrapping of the LAL class REAL4Vector. The ARRAY_STRUCT_1D() macro exposes the "data" field of the REAL4Vector class as a native scripting-language array of length "length".

```
%extend tagLIGOTimeGPS {
  tagLIGOTimeGPS() {
    return %swiglal_new_instance(struct
        ↪ tagLIGOTimeGPS);
  }
  tagLIGOTimeGPS(const struct tagLIGOTimeGPS *src) {
    return %swiglal_new_copy(*src, struct
        ↪ tagLIGOTimeGPS);
  }
  ~tagLIGOTimeGPS() {
    %swiglal_struct_call_dtor(XLALFree, $self);
  }
}
```

Fig. 2a. Example expansion of the %swiglal_struct_extend() macro for the LAL class LIGOTimeGPS. This class contains only static fields, and so SWIGLAL provides a constructor, copy constructor, and destructor for this class. The SWIG %extend directive adds methods to an existing class; methods named after the class are interpreted as constructors, while methods named after the class with the prefix "~" are interpreted as destructors. The %swiglal_new_instance() macro allocates a new LIGOTimeGPS instance using XLALCalloc(); the %swiglal_new_copy() macro creates a copy of an existing LIGOTimeGPS instance; and the %swiglal_struct_call_dtor() macro calls the destructor function XLALFree().

```
%extend tagREAL4Vector {
  ~tagREAL4Vector() {
    %swiglal_struct_call_dtor(XLALDestroyREAL4Vector,
        ↪ $self);
  }
}
```

Fig. 2b. Example expansion of the %swiglal_struct_extend() macro for the LAL REAL4Vector class. Since this class points in dynamically-allocated memory in its "data" field (Fig. 1), only the destructor is provided, which calls the destructor function XLALDestroyREAL4Vector().

2.1. Class constructors and destructors

LALSuite classes can be separated into two groups, based on their memory requirements. Classes which contain only static fields, and do not point to dynamically-allocated memory, can be straightforwardly allocated and freed with XLALMalloc()/XLALCalloc() and XLALFree().[1] For classes which point to dynamically-allocated memory, custom constructor and destructor functions are provided; they are generally named after the class prefixed with "XLALCreate…" and "XLALDestroy…". The SWIGLAL `generate_swig_iface.py` script determines to which group each LALSuite class belongs, by using the XML parse tree to determine if a destructor "XLALDestroy…" exists for a particular class. The script then outputs calls to the macro %swiglal_struct_extend() Figs. 2a and 2b show two examples of the expansion of %swiglal_struct_extend() for a class with only static fields (LIGOTimeGPS) and a class with dynamically-allocated memory (REAL4Vector). The provision of correct destructors is necessary to free the user from manual memory management, which high-level languages are expected to handle. The provision of constructors for classes with static fields provides methods for creating new classes from high-level languages without access to low-level memory functions like XLALMalloc().

2.2. Memory ownership paradigms

LALSuite assumes that all class instances are referred to exactly once. When a class instance is destroyed, all dynamically-allocated memory associated with the instance is freed, including any instances of other classes that are pointed to by the parent instance; put another way, the parent instance "owns" the memory of the child instances it points to. High-level languages, however, allow multiple references to be taken to a particular class instance, and memory is only freed once no references to that instance remain. Class instances are responsible for freeing their own memory, but do not "own" the memory of any instances of other classes they point to.

Figs. 3a and 3b illustrate how the tension between these different paradigms of memory ownership could potentially cause problems. The LAL REAL4TimeSeries class contains a pointer to an instance of the REAL4Vector class,[2] and its constructor and destructor functions create and destroy all dynamic memory associated with a REAL4TimeSeries instance, including the REAL4Vector pointer (Fig. 3a). In Python, however, the REAL4TimeSeries and REAL4Vector instances have no parent–child relationship; the user is free to create a REAL4TimeSeries instance (Fig. 3b, line 2), store a reference to the REAL4Vector instance it points to [line 6], then delete the REAL4TimeSeries instance [line 7] and assume the REAL4Vector instance will continue to be valid [line 8]. This is incompatible with the LALSuite memory ownership model, which would destroy the REAL4Vector instance along with the REAL4TimeSeries that pointed to it, corrupting the reference stored to the REAL4Vector by the user.

To resolve this tension, the SWIGLAL interface implements a system which tracks the memory ownership relationship between instances. In Fig. 3b, line 6, SWIGLAL modifies the wrapper code for the "data" field of "ts" to record that the REAL4Vector instance "ts.data", assigned to "ts_data", is owned by the REAL4TimeSeries instance. This record is stored in an associative array called the *parent map*. The parent map also records a reference count of the number of times "ts.data" has been accessed. Then, in line 7, the Python del operator is called on "ts", which would normally immediately call the REAL4TimeSeries destructor; here SWIGLAL intervenes to check whether "ts" exists in the parent map, i.e. whether it owns the memory of another class instance. Since "ts" owns the memory of "ts.data", the destructor function XLALDestroyREAL4TimeSeries() is not called, and so the memory allocated for the REAL4Vector instance stored by "ts_data" is not destroyed. Finally, in line 10, the Python del operator is called on "ts_data"; here SWIGLAL checks who owns the memory of "ts_data" (i.e. the original "ts" object) and whether there are any outstanding references to that memory. Since both "ts" and "ts_data" have been destroyed, it is safe for SWIGLAL to call the now call the destructor function XLALDestroyREAL4TimeSeries() for the REAL4TimeSeries instance created in line 2.

[1] These are LALSuite's equivalents to the C functions malloc()/calloc() and free(), but which also provide optional memory debugging features.

[2] Strictly speaking, REAL4TimeSeries is defined with a pointer to REAL4Sequence, a synonym for REAL4Vector.

```
typedef struct tagREAL4TimeSeries {
      CHAR name[LALNameLength];
      LIGOTimeGPS epoch;
      REAL8 deltaT;
      REAL8 f0;
      LALUnit sampleUnits;
      REAL4Vector *data;
} REAL4TimeSeries;
REAL4TimeSeries *XLALCreateREAL4TimeSeries(const
      ↪ CHAR *name, const LIGOTimeGPS *epoch, REAL8
      ↪ f0, REAL8 deltaT, const LALUnit
      ↪ *sampleUnits, size_t length);
void XLALDestroyREAL4TimeSeries(REAL4TimeSeries
      ↪ *series);
```

Fig. 3a. Illustration of memory ownership tracking in SWIGLAL: Definition of the LAL REAL4TimeSeries class. The “data” field of this class points to an instance of the REAL4Vector class. The XLALCreateREAL4TimeSeries() function allocates memory for a new REAL4TimeSeries instance, and for a new REAL4Vector instance which is pointed to by the “data” field. The XLALDestroyREAL4TimeSeries() function destroys both the REAL4TimeSeries instance and the pointed-to REAL4Vector instance.

```
>>> import lal
>>> ts = lal.CreateREAL4TimeSeries("timeseries",
      ↪ 1234567890.0, 0, 1./100, lal.VoltUnit, 10)
>>> ts.data.data = range(0,10)
>>> print(ts.data.data)
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
>>> ts_data = ts.data
>>> del ts
>>> print(ts_data.data)
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
>>> del ts_data
```

Fig. 3b. Illustration of memory ownership tracking in SWIGLAL: Example usage in Python. The user creates a new REAL4TimeSeries instance at line 2, and assigns values to the data array pointed to by the REAL4Vector instance in lines 3 and 4. The user stores a reference to the “data” member of the REAL4TimeSeries instance in line 5, and attempts to delete the REAL4TimeSeries instance in line 6 using the Python del operator. This does not, however, trigger an immediate call to XLALDestroyREAL4TimeSeries(), since SWIGLAL knows that the user retains a reference to the REAL4Vector instance in the variable “ts_data”. The data contained in the REAL4Vector instance therefore remains accessible (line 8), and XLALDestroyREAL4TimeSeries() is called only when “ts_data” is destroyed (line 9).

The SWIGLAL memory ownership tracking system, combined with the native reference counting of objects in Python and Octave, completely frees the user from any manual memory management, as is appropriate for a high-level language, while respecting the LALSuite memory management paradigm. Memory allocated by LALSuite functions is only freed once it is no longer used, and conversely is retained only as long as needed, thus minimising memory usage.

2.3. Fixed-length and dynamically-sized arrays

Gravitational-wave data analysis frequently involves operations on large time- and/or frequency-domain data series, and LALSuite provides many functions and classes to represent such data, such the REAL4Vector (Fig. 1) and REAL4TimeSeries (Fig. 3a) classes. Such data should be accessible from within the SWIGLAL interface as native array objects, and in an efficient manner without copying of data between the C class instance and its high-level language representation.

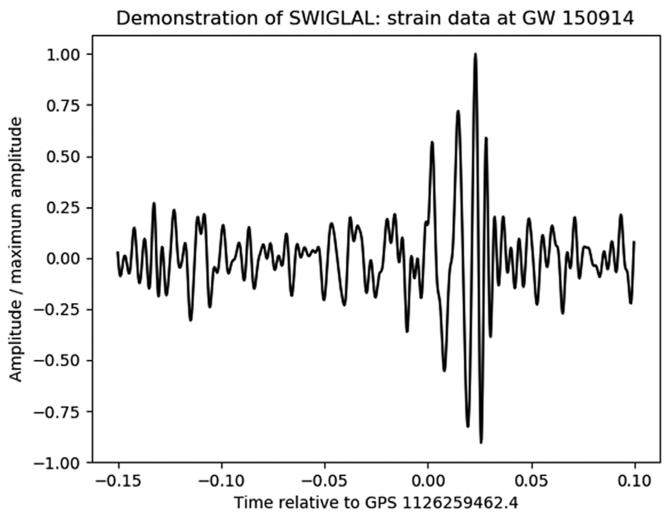

Fig. 4. Whitened, band-pass-filtered strain data from the LIGO Hanford detector at the time of the gravitational-wave event *GW 150914*, as output by the example Python script listed in Appendix.

SWIGLAL provides several typemaps for converting numerical arrays to/from native array objects; for Python, NumPy [16] arrays are used, while for Octave the native matrix type (or subclasses thereof) are used. For fixed-length C arrays, SWIGLAL supports both one- and two-dimensional arrays; typemaps are provided for both function arguments and C structure fields. Dynamically-allocated arrays are typically implemented as specific classes in LALSuite; SWIGLAL provides directives which are added to those classes to provide the type conversion. For the REAL4Vector class, for example (Fig. 1), the ARRAY_STRUCT_1D() macro modifies the wrapper code for the “data” field, so that e.g. in Python it accepts any valid sequence of floating-point numbers on assignment, and exposes the “data” field as a NumPy array [17] view which directly accesses the underlying C memory.

Some LALSuite array classes store only array data, and nothing else: REAL4Vector (Fig. 1) is such a class, while REAL4TimeSeries (Fig. 3a) contains additional fields. SWIGLAL provides additional typemaps for pure-array classes such as REAL4Vector so that functions can accept both class instances and native array objects as arguments. For example, the Python interface to a function which takes a REAL4Vector instance as an argument will also accept a NumPy array of the appropriate type.

2.4. Example: extract strain data at time of GW 150914

Fig. 4 shows the output of an example Python script, listed in the Appendix, which extracts the strain data at the time of the first detected gravitational wave event *GW 150914* [cf. Figure 1 of 3]. The script reads in strain data from the LIGO Hanford detector [18] at the time of the event, available from [19]; whitens and band-pass-filters the data so that the event is clearly visible; and plots the processed strain data in the vicinity of the event. The script is *not* intended as an example of best-practise signal processing for gravitational-wave data analysis, but as an illustration of what may be accomplished in 35 lines of Python code, by harnessing the power of LALSuite routines through the SWIGLAL interface.

3. Impact

Table 2 show the usage of the SWIGLAL interfaces by Python code within LALSuite itself, and by seven other gravitational-wave data analysis packages. The table gives, for each LALSuite

Table 2
Usage of the SWIGLAL interfaces by LALSuite itself, and by the PyCBC, GstLAL, Bilby, GWpy, PyFstat, CWInPy, and OctApps packages. The header for each table section gives the package name and version, and the percentage of (non-blank, non-comment) lines of code (LOC) written in C, Python, and/or Octave. The columns give the number of source files (out of the total in each package) which reference the SWIGLAL interfaces of each LALSuite library, as well as the number of distinct constants, variables, functions, and classes exported by the SWIGLAL interfaces that are referenced by each package. For functions, an estimate of the total (non-blank, non-comment) lines of C code (LOC) represented, including nested calls, is given in parentheses. Note that SWIGLAL provides a single interface to the LAL and LALSupport libraries, which are therefore counted together.

	Sources	Constants	Variables	Functions (LOC)	Classes
LALSuite, version 6.67 [5]. LOC: C ∼ 95%, Python ∼ 5%.					
LAL(Support)	71/222	25	3	32 (3k)	18
LALBurst	39/222	0	0	1 (0.5k)	0
LALFrame	8/222	0	0	20 (4k)	4
LALInference	27/222	7	3	24 (93k)	4
LALInspiral	7/222	0	0	5 (1k)	1
LALMetaio	8/222	0	0	0 (0)	2
LALPulsar	10/222	9	0	43 (16k)	21
LALSimulation	20/222	0	0	40 (60k)	0
PyCBC, version 1.15.4 [9]. LOC: Python ∼ 100%.					
LAL(Support)	41/309	16	0	32 (3k)	20
LALFrame	2/309	2	0	25 (6k)	1
LALPulsar	1/309	1	0	5 (2k)	2
LALSimulation	14/309	4	1	50 (61k)	1
GstLAL, version 1.5.1[a]. LOC: C ∼ 52%, Python ∼ 48%.					
LAL(Support)	41/117	6	1	16 (2k)	11
LALSimulation	9/117	0	0	7 (59k)	0
Bilby, version 0.6.5 [10]. LOC: Python ∼ 100%.					
LAL(Support)	8/83	3	0	5 (1k)	8
LALSimulation	5/83	0	0	18 (59k)	0
GWpy, version 1.0.1 [11]. LOC: Python ∼ 100%.					
LAL(Support)	10/267	10	8	3 (1k)	4
LALFrame	2/267	0	0	9 (3k)	2
PyFstat, version 1.3 [12]. LOC: Python ∼ 100%.					
LAL(Support)	6/28	8	1	5 (1k)	6
LALPulsar	6/28	10	1	22 (14k)	21
CWInPy, version 0.2.1 [13]. LOC: Python ∼ 100%.					
LAL(Support)	3/42	2	0	4 (1k)	4
LALInference	14/42	0	1	0 (0)	0
LALPulsar	2/42	0	0	1 (1k)	0
LALSimulation	1/42	0	0	9 (1k)	0
OctApps, version 0.2 [14]. LOC: Octave ∼ 100%.					
LAL(Support)	9/243	7	0	9 (1k)	5
LALPulsar	7/243	29	1	22 (13k)	20

[a]Includes the packages: GstLAL Ugly, version 1.6.6; GstLAL Inspiral, version 1.6.9; GstLAL Calibration, version 1.2.11; GstLAL Burst, version 0.2.0 [15].

library: the number of source files (out of the package total) which reference the SWIGLAL interface for that library (e.g. by importing the interface in Python using "**import**"), and the number of distinct symbols referred to by the package. The table also lists an estimate of the total lines of C code represented by the LALSuite functions referenced from each package; the estimates include any nested calls to other LALSuite functions. Python code within LALSuite is a substantial user of SWIGLAL, in terms of source files (∼ 3–30%), and lines of C code utilised (∼ 180k).

The PyCBC [20–23] and GstLAL [24,25] data analysis packages were used in the first detections of gravitational waves [3,4]. PyCBC makes use of the LAL library in over 10% of its source files, mostly for manipulating time- and frequency-domain data series. It uses 25 functions from LALFrame to read and write gravitational wave data in the standard Frame format [26] produced by gravitational-wave observatories. It uses 50 functions from LALSimulation to generate template waveforms for matched filtering of the gravitational-wave data. It uses a few functions from LALPulsar for template bank generation [27]. The total lines of LALSuite code utilised by PyCBC is ∼ 72k. While primarily written in C, GstLAL uses 16 functions from the LAL library to manipulate time- and frequency-domain data, and compute the geocentric time delay to the gravitational-wave observatories, from Python scripts. It uses 7 functions from LALSimulation to generate template waveforms in Python. The total lines of LALSuite code utilised by GstLAL from Python is ∼ 61k.

Bilby [28] aims to be a user-friendly package for Bayesian inference for use in gravitational-wave data analysis [e.g. 29]. It accesses LALSuite through the SWIGLAL interfaces in ∼ 5–10% of its source files. The LAL library is used to handle time- and-frequency domain gravitational-wave data, and LALSimulation is used to generate template waveforms for computing the Bayesian likelihood function. A total of ∼ 60k lines of LALSuite code are utilised.

GWpy [30] is a general package for easily accessing, visualising, and studying gravitational-wave data. It makes use of the LAL and LALFrame libraries, primarily for manipulating gravitational-wave data in the Frame format, in about ∼ 5% of its source files. It uses ∼ 4k lines of LALSuite code in total.

PyFstat [31], CWInPy [32], and OctApps [33] are data analysis packages focused on the search for *continuous* gravitational waves from rapidly rotating neutron stars; this class of gravitational wave signals has not yet been detected. PyFstat uses LAL and LALPulsar (in $\sim$ 20% of its source files) to compute the $\mathcal{F}$-statistic [34], a standard data analysis routine for continuous gravitational wave searches. CWInPy uses a few LALSuite routines to e.g. handle frequency-domain data, convert between time standards, and access properties of the gravitational-wave observatories. OctApps uses routines, predominately from LALPulsar, to compute the $\mathcal{F}$-statistic and its associated parameter space metric [35] for designing continuous gravitational wave searches. Both PyFstat and OctApps use $\sim$ 14–15k lines of LALSuite code, which CWInPy uses $\sim$ 3k.

4. Conclusions

LALSuite is an important, well-tested component of the gravitational-wave data analysis software stack. SWIGLAL makes innovative use of SWIG to provide automatically-generated interfaces to LALSuite for Python and Octave, with an emphasis on modelling native code behaviour in those languages. The interfaces have facilitated the development of modern gravitational-wave data analysis software written in Python, in particular PyCBC which was used in the first discovery of gravitational waves. The extensive use of the interfaces by a wide variety of Python and Octave packages for gravitational-wave data analysis demonstrates the impact and usefulness of SWIGLAL.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

I am grateful to Adam Mercer for help integrating and maintaining SWIGLAL within LALSuite, and to Kipp Cannon, Jolien Creighton, Nickolas Fotopoulos, Duncan Macleod, and Leo Singer for contributing improvements and bug fixes. I thank David Keitel and Duncan Macleod for helpful comments on the manuscript. This work was supported by the Australian Research Council through project number CE170100004, and by the Max Planck Society, Germany. This paper has document number LIGO-P2000094.

Appendix. Example Python script to extract strain data at time of GW 150914

```python
import lal
import lalframe as lalfr
import numpy as np
import matplotlib.pyplot as plt

# read strain data at time of GW 150914
#- frame file downloaded from https://www.gw-openscience.org/
frame_file = lalfr.FrFileOpenURL("./H-H1_GWOSC_4KHZ_R1-1126259447-32.gwf")
gw_strain = lalfr.FrFileReadREAL8TimeSeries(frame_file, "H1:GWOSC-4KHZ_R1_STRAIN", 0)

# compute average power spectral density of strain data
psd_segment_len = int(4.0 / gw_strain.deltaT)
psd_window = lal.CreateTukeyREAL8Window(psd_segment_len, 0.5)
fft_plan = lal.CreateForwardREAL8FFTPlan(psd_segment_len, 0)
gw_psd_length = psd_segment_len // 2 + 1
gw_psd = lal.CreateREAL8FrequencySeries("psd", gw_strain.epoch, 0, 0, lal.DimensionlessUnit,
    ↪ gw_psd_length)
lal.REAL8AverageSpectrumWelch(gw_psd, gw_strain, psd_segment_len, psd_segment_len, psd_window,
    ↪ fft_plan)
gw_psd_f = gw_psd.f0 + np.arange(gw_psd.data.length) * gw_psd.deltaF;

# transform strain data to Fourier domain
gw_fourier_length = gw_strain.data.length // 2 + 1
gw_fourier_deltaF = 0.5 / gw_strain.deltaT / (gw_fourier_length - 1)
fft_plan = lal.CreateForwardREAL8FFTPlan(gw_strain.data.length, 0)
gw_fourier = lal.CreateCOMPLEX16FrequencySeries("fourier", gw_strain.epoch, 0, gw_fourier_deltaF,
    ↪ lal.DimensionlessUnit, gw_fourier_length)
gw_fourier_f = gw_fourier.f0 + np.arange(gw_fourier.data.length) * gw_fourier.deltaF;
lal.REAL8ForwardFFT(gw_fourier.data, gw_strain.data, fft_plan)

# whiten strain data in Fourier domain
gw_psd_at_fourier_f = np.interp(gw_fourier_f, gw_psd_f, gw_psd.data.data)
gw_fourier.data.data = gw_fourier.data.data / np.sqrt(gw_psd_at_fourier_f)

# transform whitened strain data back to time domain
fft_plan = lal.CreateReverseREAL8FFTPlan(gw_strain.data.length, 0)
lal.REAL8ReverseFFT(gw_strain.data, gw_fourier.data, fft_plan)
```

```
# band—pass filter whitened time series between 50 and 300 Hz
lal.HighPassREAL8TimeSeries(gw_strain, 50, 0.1, 6)
lal.LowPassREAL8TimeSeries(gw_strain, 300, 0.1, 6)

# extract strain data [—0.15,0.10] seconds around GW 150914
time_of_GW150914 = lal.LIGOTimeGPS("1126259462.4")
first_sample = int(((time_of_GW150914 — 0.15) — gw_strain.epoch) / gw_strain.deltaT)
num_samples = int(0.25 / gw_strain.deltaT)
gw_strain = lal.CutREAL8TimeSeries(gw_strain, first_sample, num_samples)
gw_strain_t = float(gw_strain.epoch — time_of_GW150914) + np.arange(gw_strain.data.length) *
    ↪ gw_strain.deltaT

# plot strain data
plt.plot(gw_strain_t, gw_strain.data.data / max(gw_strain.data.data), "k—")
plt.title("Demonstration␣of␣SWIGLAL:␣strain␣data␣at␣GW␣150914")
plt.xlabel(f"Time␣relative␣to␣GPS␣{time_of_GW150914}")
plt.ylabel("Amplitude␣/␣maximum␣amplitude")
plt.show()
```

References

[1] Python Software Foundation. Python language reference. 2020, URL https://docs.python.org/reference/.

[2] Eaton JW, Bateman D, Hauberg S, Wehbring R. GNU Octave manual: a high-level interactive language for numerical computations. 2020, URL https://www.gnu.org/software/octave/doc/.

[3] Abbott BP, et al. Observation of gravitational waves from a binary black hole merger. Phys Rev Lett 2016;116(6):061102. http://dx.doi.org/10.1103/PhysRevLett.116.061102, arXiv:1602.03837.

[4] Abbott BP, et al. GW170817: Observation of gravitational waves from a binary neutron star inspiral. Phys Rev Lett 2017;119(16):161101. http://dx.doi.org/10.1103/PhysRevLett.119.161101, arXiv:1710.05832.

[5] LIGO Scientific Collaboration. LIGO Algorithm Library - LALSuite, free software (GPL). 2018, http://dx.doi.org/10.7935/GT1W-FZ16.

[6] Mercer A. Top-level: mark as version 6.67. GitLab 2020. URL https://git.ligo.org/lscsoft/lalsuite/-/tags/lalsuite-v6.67.

[7] Buonanno A, Chen Y, Vallisneri M. Detection template families for gravitational waves from the final stages of binary black-hole inspirals: Nonspinning case. Phys Rev D 2003;67(2):024016. http://dx.doi.org/10.1103/PhysRevD.67.024016, arXiv:gr-qc/0205122.

[8] Beazley DM. SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the 4th conference on USENIX Tcl/Tk workshop. TCLTK'96, Vol. 4, USA: USENIX Association; 1996, p. 15.

[9] Nitz A, et al. gwastro/pycbc: PyCBC release v1.15.4. Zenodo 2020. URL https://doi.org/10.5281/zenodo.3630601.

[10] Ashton G. 0.6.5 – version 0.6.5 release. GitHub 2020. URL https://github.com/lscsoft/bilby/releases/tag/0.6.5.

[11] Macleod D, et al. gwpy/gwpy: 1.0.1. Zenodo 2020. http://dx.doi.org/10.5281/zenodo.3598469, URL https://doi.org/10.5281/zenodo.3598469.

[12] Ashton G, Keitel D, Prix R. PyFstat-v1.3. Zenodo 2020. URL https://doi.org/10.5281/zenodo.3620861.

[13] Pitkin M. v0.2.1 – Release for PyPI. GitHub 2020. URL https://github.com/cwinpy/cwinpy/releases/tag/v0.2.1.

[14] Wette K. v0.2 – Release for Journal of Open Source Software paper acceptance. GitHub 2018. URL https://github.com/octapps/octapps/releases/tag/v0.2.

[15] Godwin P. README.md: update versions. GitLab 2019. URL https://git.ligo.org/lscsoft/gstlal/-/blob/b3b89bc/README.md.

[16] Oliphant TE. Guide to NumPy. 2006, URL http://www.numpy.org/.

[17] van der Walt S, Colbert SC, Varoquaux G. The NumPy array: A structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22. http://dx.doi.org/10.1109/MCSE.2011.37.

[18] Aasi J, et al. Advanced LIGO. Classical Quantum Gravity 2015;32(7):074001. http://dx.doi.org/10.1088/0264-9381/32/7/074001, arXiv:1411.4547.

[19] Abbott R, et al. Open data from the first and second observing runs of Advanced LIGO and Advanced Virgo. 2019, arXiv e-prints arXiv:1912.11716 arXiv:1912.11716.

[20] Allen B, Anderson WG, Brady PR, Brown DA, Creighton JDE. FINDCHIRP: An algorithm for detection of gravitational waves from inspiraling compact binaries. Phys Rev D 2012;85(12):122006. http://dx.doi.org/10.1103/PhysRevD.85.122006, arXiv:gr-qc/0509116.

[21] Allen B. χ^2 Time-frequency discriminator for gravitational wave detection. Phys Rev D 2005;71(6):062001. http://dx.doi.org/10.1103/PhysRevD.71.062001, arXiv:gr-qc/0405045.

[22] Nitz AH, Dent T, Dal Canton T, Fairhurst S, Brown DA. Detecting binary compact-object mergers with gravitational waves: Understanding and improving the sensitivity of the PyCBC search. Agron J 2017;849(2):118. http://dx.doi.org/10.3847/1538-4357/aa8f50, arXiv:1705.01513.

[23] Dal Canton T, Nitz AH, Lundgren AP, Nielsen AB, Brown DA, Dent T, Harry IW, Krishnan B, Miller AJ, Wette K, Wiesner K, Willis JL. Implementing a search for aligned-spin neutron star-black hole systems with advanced ground based gravitational wave detectors. Phys Rev D 2014;90(8):082004. http://dx.doi.org/10.1103/PhysRevD.90.082004, arXiv:1405.6731.

[24] Sachdev S, Caudill S, Fong H, Lo RKL, Messick C, Mukherjee D, Magee R, Tsukada L, Blackburn K, Brady P, Brockill P, Cannon K, Chamberlin SJ, Chatterjee D, Creighton JDE, Godwin P, Gupta A, Hanna C, Kapadia S, Lang RN, Li TGF, Meacher D, Pace A, Privitera S, Sadeghian L, Wade L, Wade M, Weinstein A, Liting Xiao S. The GstLAL search analysis methods for compact binary mergers in Advanced LIGO's second and Advanced Virgo's first observing runs. 2019, arXiv 1901.08580. arXiv:1901.08580.

[25] Messick C, Blackburn K, Brady P, Brockill P, Cannon K, Cariou R, Caudill S, Chamberlin SJ, Creighton JDE, Everett R, Hanna C, Keppel D, Lang RN, Li TGF, Meacher D, Nielsen A, Pankow C, Privitera S, Qi H, Sachdev S, Sadeghian L, Singer L, Thomas EG, Wade L, Wade M, Weinstein A, Wiesner K. Analysis framework for the prompt discovery of compact binary mergers in gravitational-wave data. Phys Rev D 2017;95(4):042001. http://dx.doi.org/10.1103/PhysRevD.95.042001, arXiv:1604.04324.

[26] LIGO, Virgo. Specification of a common data frame format for interferometric gravitational wave detectors. Tech. rep. LIGO-T970130-v1, VIR-067A-08, LIGO, Virgo; 2009.

[27] Wette K. Lattice template placement for coherent all-sky searches for gravitational-wave pulsars. Phys Rev D 2014;90:122010. http://dx.doi.org/10.1103/PhysRevD.90.122010, arXiv:1410.6882.

[28] Ashton G, Hübner M, Lasky PD, Talbot C, Ackley K, Biscoveanu S, Chu Q, Divakarla A, Easter PJ, Goncharov B, Hernandez Vivanco F, Harms J, Lower ME, Meadors GD, Melchor D, Payne E, Pitkin MD, Powell J, Sarin N, Smith RJE, Thrane E. BILBY: A user-friendly Bayesian inference library for gravitational-wave astronomy. Astrophys J Suppl 2019;241(2):27. http://dx.doi.org/10.3847/1538-4365/ab06fc, arXiv:1811.02042.

[29] Abbott BP, et al. GW190425: Observation of a compact binary coalescence with total mass $\sim 3.4 M_\odot$. 2020, arXiv 2001.01761. arXiv:2001.01761.

[30] Macleod D, et al. GWpy: a python package for gravitational-wave astrophysics. Softw X 2020. submitted for publication.

[31] Ashton G, Prix R, Jones DI. A semicoherent glitch-robust continuous-gravitational-wave search method. Phys Rev D 2018;98(6):063011. http://dx.doi.org/10.1103/PhysRevD.98.063011, arXiv:1805.03314.

[32] Pitkin M. Documentation for CWInPy. 2019, URL https://cwinpy.readthedocs.io/en/latest/.

[33] Wette K, Prix R, Keitel D, Pitkin M, Dreissigacker C, Whelan JT, Leaci P. Octapps: a library of Octave functions for continuous gravitational-wave data analysis. J Open Source Softw 2018;3(26):707. http://dx.doi.org/10.21105/joss.00707.

[34] Jaranowski P, Królak A, Schutz BF. Data analysis of gravitational-wave signals from spinning neutron stars: The signal and its detection. Phys Rev D 1998;58(6):063001. http://dx.doi.org/10.1103/PhysRevD.58.063001, arXiv:gr-qc/9804014.

[35] Wette K. Parameter-space metric for all-sky semicoherent searches for gravitational-wave pulsars. Phys Rev D 2015;92(8):082003. http://dx.doi.org/10.1103/PhysRevD.92.082003, arXiv:1508.02372.