Development of KAGRA Algorithmic Library (KAGALI)

Ken'ichi Oohara

Department of Physics, Niigata University,
Niigata, 950-2181, Japan
E-mail: oohara@astro.sc.niigata-u.ac.jp

Koh Ueno[1], Hirotaka Yuzurihara[1], Yosuke Itoh[2], Hirotaka Takahasi[3], Tsukasa Arima[1], Kazunari Eda[2], Yoshinori Fujii[2], Kazuhiro Hayama[1], Yuta Hiranuma[4], Shigeki Hirobayashi[5], Nobuyuki Kanda[1], Masato Kaneyama[1], Jeongcho Kim[6], Chunglee Kim[7], Hyung Won Lee[6], Shuhei Mano[8], Kyohei Miyake[5], Akinobu Miyamoto[1], Yuta Nakanishi[5], Naoko Ohishi[9], Masaya Nakano[5], Hayato Nakao[1], Tatsuya Narikawa[1], Kazuki Sakai[3], Yukitsugu Sasaki[3], Ayaka Shoda[9], Hiroto Suwabe[4], Hideyuki Tagoshi[1], Kazuyuki Tanaka,[1] Satoshi Ueki[3], Takeshi Wakamatsu[4], Takahiro Yamamoto[1], Jun'ichi Yokoyama[2], Takaaki Yokozawa[1]

[1]*Osaka City University,* [2]*the University of Tokyo,* [3]*Nagaoka University of Technology,* [4]*Niigata University,* [5]*Toyama University,* [6]*Inje University,* [7]*Kyung Hee University,* [8]*The Institute of Statistical Mathematics,* [9]*National Astronomical Observatory of Japan*

We will report the present status of development of KAGRA Algorithmic Library (KAGALI) for data analysis of gravitational waves. It includes the concept and the basic structure of the library.

Keywords: Gravitational waves; data analysis.

1. Introduction

As for software for gravitational wave data analysis, the Data Analysis Software Working Group of the LIGO Scientific Collaboration (LSC) has already opened a fine software suite, LALSuite, to the public.[1] Nevertheless, we should prepare routines proper to KAGRA, especially for data handling tools and for new methods we are developing. The LSC Algorithm Library (LAL) prepares a clever error handling, bug tracking system.[2] But it seems to be very complicated for us to append new functions. Thus KAGRA Data Analysis Subsystem (DAS) decided to develop our own software suite that is independent to the LAL in principle. As the first step, we began to develop the KAGRA Algorithmic Library, hereafter KAGALI.

KAGALI Suite is comprised of KAGALI and KAGALI-Apps. KAGALI is a library of routines for gravitational wave data analysis and KAGALI-Apps is a data analysis application packages build upon not only KAGALI but also LALSuite and libraries developed by Virgo. We will make KAGALI Suite work independently of LAL, while any of available software including KAGALI, LALSuite, the Virgo software, etc. will be used for KAGRA data analysis.

2. KAGALI C-coding Guidelines

KAGRA DAS published "KAGALI C-coding style guide" [3] for internal use only for the time being, in order to reduce bugs, to make code-debugging easy, to produce a common properties by which the KAGRA data analysis team can develop data analysis programs efficiently and quickly, and to make programs easily re-usable and understandable by standardizing code appearances. Source codes must be easy to understand for others and you-at-some-time-later, and easy to understand without a comment.

KAGALI does not specify what language we have to use to develop them, but C language is used in the major part at the moment. The style guide specifies that KAGALI must be written in "Clean C" and C99, where "Clean C" is a common subset of the C and C++. KAGALI must be written in the way such that it can run on any computers on which main data analysis tasks are executed. However, we don't place first priority on portability. We mainly intend instead to achieve higher performance. KAGALI is assumed to be installed and work on various systems of, at least, Unix-like OS including Linux and Mac OS X anyway. It is confirmed for Mac OS X as well as for Red Hat Enterprise Linux (RHEL) 6.x, 7.x and its compatibles, for example, Scientific Linux and CentOS with the GNU C compiler (gcc) and the Intel C compiler (icc).

GNU extensions may be used if the performance is improved, but the alternative part that works on standard C99 must be added; one of them will be chosen by the preprocessor as follows,

```
#ifdef __GNUC__
  (the GNU extensions (-std=gnu99) part)
#else
  (the C99 (-std=c99) part)
#endif
```

KAGALI routines should not call file input/output routines with a few exceptions including functions that handles Frame Library. They should not call any system-dependent functions such as `system()`, `getenv()`, `srand()`, and so on. Functions in the library should not terminate the program, that is, should not call `abort()`, `exit()` or issue `signal()` except for some particular cases. They should return a status code via the designated structure `KGLStatus`, instead.

Use the atomic data type of C99 (`char`, `unsigned char`, `int`, `double`, etc.) rather than the LAL-specific atomic data types (`CHAR`, `UCHAR`, `INT4`, `REAL8`, etc). If clarification of data size of integers is necessary, use `int32_t`, `int64_t`, etc. of C99. Use `_Bool` of C99 for boolean data type. You can use `bool`, `true` and `false` defined in the C99 header stdbool.h.

Particular attention may be given to the complex data type, since some libraries including FFTW3 and GSL on which KAGALI depends define their own complex type in order to adapt to C compilers before C99. However, FFTW3 defines its own `fftw_complex` to be the native complex type of C99 if you `#include <complex.h>` before `<fftw3.h>`[4]. Representation of complex numbers of GSL is binary-compatible with that of the native complex type of C99 at least as long as you use GNU C compiler for Intel CPUs. Therefore, KAGALI will include `<complex.h>` in default and you can use `typedef` and macros defined there as well as the native complex type of C99, while you need to cast it to `gsl_complex` when you call functions of GSL.

The same rules as LAL are applied to names of functions, variables, etc., while the prefix "`KGL`" or "`kgl`" rather than "`LAL`" or "`lal`" is used. The use of Studly Caps is recommended except for macros, which are generally all in uppercase and underscores if necessary.

Every variable should have a different name from each other even if there is no overlap in the scope and/or the lifetime between any of two variables.

3. Version Control

We use "git" for the version control and the git server for KAGRA DAS is prepared at the University of Tokyo. The KAGALI sources are located in the KAGALI git repository on it. It is still a pre-alpha version and only developers can access the repository at this moment.

The version identifier of KAGALI is embedded in the library via `KGLVersion.c` and `KAGLVersion.h`. `KAGLVersion.h` will be updated automatically from the "git-log" when a developer modifies the source and commit it to git. The identifier includes the date and the name of the latest committer as well as the commit hash of git, such as

```
$KGL: 2015-06-11 18:58:51 +0900 oohara
e1c4a138e786d147e733699d97ee3a1f74f0a237 $
```

4. Analysis Tools of the Sources

Tools that can automatically detect various bugs is indispensable in developing a library. LAL provides the sophisticated mechanisms, such as the error reporting system and memory leak checking. It is a heavy task to port all of them into our system or to construct the same mechanisms by ourselves. Instead, we made a simpler mechanism of reporting error during the execution of a program. All the KAGALI functions generally have a pointer to a `KGLStatus` structure type as their first argument. Each function set an error code and message if a failure occurs. The message includes the function name and the line number to point out where the failure accrues.

For memory leak checking, we decided to use external applications rather than to use custom functions for memory management, such as `LALMalloc`, `LALFree`, etc.

One is "cppcheck",[5] which is a static analysis tool for C/C++ code. It primarily detects the types of bugs that the compiler normally do not detect, such as out of bounds checking, memory leaks checking, checking for uninitialized variables, warning if obsolete or unsafe functions are used, etc. When the sources are modified and pushed to the KAGALI git server, they will be checked by cppcheck nightly. The automatic checking system has been provided under Jenkins[6] on the server.

However, there may be memory leaks that cannot be detected by cppcheck, since it makes a static analysis without actual execution of the program. Thus we use "Valgrind" to detect dynamically memory leaks. It is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. Valgrind redirects `malloc`, `calloc`, etc. to the functions that Valgrind provides to detect memory errors. Users don't need to change the source code or link a special library.

5. The Present Status of KAGALI

The latest version of KAGALI Suite is 0.4-alpha, but it is not open to the public yet. This version requires the GNU Scientific Library (GSL, version 1.13 to 1.16), FFTW (version 3.2.0 or up) and LIGO/Virgo Frame library (FrameL, version 8.0.0 or up). Some of application programming interface of GSL were changed for gsl-2.x. Since it may cause errors in compiling KAGALI, please avoid using gsl-2.x for the time being. GNU autotools (autoconf/automake/libtools) are required for a maintainer and developer. KAGALI library calls CBLAS functions and thus it is recommended to link one of tuned CBLAS libraries in order to speed up linear algebra operations. At present, `configure` script supports OpenBLAS, ATLAS, Intel Math Kernel Library (MKL) as well as gslcblas, which is a part of GSL.

The current source tree is temporally

- kagali
  - ◇ kglcommon
    - ○ estimatepsd (averaged power spectral density)
    - ○ fft (FFTW wrappers)
    - ○ frame (FrameL wrappers)
    - ○ nha (least square fits)
    - ○ noisepsd (noise power spectral density)
    - ○ std (error report, version control, etc.)
    - ○ tools (bandpass filters, impulse response, linear algebra, mathematical functions in geometry, etc.)
  - ◇ cbc (for compact binary coalescence)
  - ◇ hht (for the Hilbert-Huang transform)
  - ◇ waveform (various wave forms)

- kagaliapps
  - ◇ cbc
  - ◇ nha

The source tree may be changed later.

6. Summary

Now we started to develop KAGALI. The basic frame work has been determined, including bug tracing mechanism, FFT wrappers and FrameL wrappers. To detect memory leak and other bugs, external applications such as cppcheck or Valgrind are used. However, it is still a pre-alpha version and we have to examine throughout the code. Some of routines in KAGALI will be applied to build the analysis pipeline for iKAGRA. The first release version will be open to public by 2017 when bKAGRA will be operated.

Acknowledgments

This work was supported by MEXT Grant-in-Aid for Scientific Research on Innovative Areas "New Developments in Astrophysics Through Multi-Messenger Observations of Gravitational Wave Sources" (Grant Number 24103005) and by MEXT Grant-in-Aid for Scientific Research(C) Grant Number 15K05071.

References

1. `https://www.lsc-group.phys.uwm.edu/daswg/`
2. B. Allen, *et. al.*, *LIGO Scientific Collaboration Algorithm Library Specification and Style Guide*, LIGO-T990030-v2, 2010, `https://dcc.ligo.org/T990030/public`
3. KAGRA Data Analysis Subsystem, *KAGALI C-coding style guide*, 2014, published internally only.
4. `http://www.fftw.org/doc/Complex-numbers.html`
5. `http://cppcheck.sourceforge.net/`
6. `https://jenkins-ci.org/`