Sol^2 - Analysis of and Comparisons on an Efficient C++-Lua Wrapper

ThePhD
Lounge<C++> - Library Developer
Columbia University - School of Engineering and Applied Sciences
New York, NY 10027
Email: phdofthehouse@gmail.com

***Abstract*—Lua is a lightweight scripting language that has grown exponentially in usage, featured in code bases for humanoid robots, video games engines, servers, and many more systems and applications. C and C++ can levy Lua within itself using its C API. However, the API is a low-level and stack-based, making it cumbersome to manage. It also does not handle many language features, library abstractions, and features of languages that are more robust than C, leading many developers to attempt both wrapping and extending the API in various ways, with varying degrees of success and popularity.**

I present a novel header-only C++ library Sol^2 that achieves an easy to use standard library-complimenting API, while still maintaining the equal or near-equal overall performance as compared to using the plain C API. I explore Sol^2's merits and design as a C++ inter-operation library, and compare both its feature set and performance to other existing libraries, highlighting some of the useful features Sol^2 provides.

1. Introduction

Sol^2 [1] is a binding between the Lua language [2] and C++. Lua's canonical implementation have seen widespread use throughout several different industries. Due to its incredibly small binary size but relative power, it has found use in games [3], robotics [4], servers [5] and many more applications. Of particular note is that it is ubiquitous for scripting in the video game industry, with titles such as World of Warcraft, AngryBirds, and several others [6] lending it practical credence. It builds into a tiny library or executable, making it suitable for even the most demanding of environments, and has a lean runtime with library features and sandboxing that can be easily enforced at the user's request.

1.1. Lua's Problems

As a C++ or C developer, the Lua C API is the offered interface. It is low-level, stack-based, and primitive in its operations. This makes it prone to off-by-one, copy-paste errors like most low-level code tends toward [7] [8] [9]. Many have attempted to wrap up the API, but have often invoked pain with their abstractions and caused many angry blogs and confused questions [10] [11]. Even experienced Lua users tend to diverge from the original C API when wrapping things up, because C is verbose at its best, and a kludgy soup at its worst [12].

Lua's developers have introduced many breaking changes between the Lua 5.1 implementation that became ubiquitous in many places and its later iterations, such as LuaJIT, Lua 5.2, 5.3. Dealing with source-level incompatibility is a problem for many developers [13]. Developing large systems on top of the Lua 5.1 API has left large codebases where refactors of all of the C code would be exceptionally painful and is thus avoided at all costs.

Many libraries were created by hobbyist programmers and corporations alike over the last two decades to try and address the idea of wrapping up the Lua C API. One of the oldest and longest-standing is luabind, which has survived to the time of writing and is still being used, with over 140 forks on Github alone [14]. With the introduction of C++11 and the resurgent interest in the C++ programming Language[1], many developers attempted to use C++ to create libraries better than their predecessors. The original Sol was no exception to this.

1.2. Sol^2's History

Sol^2 was originally just Sol, started by Dr. Danny Y. (Rapptz) [15]. It grew out of the Lounge<C++> shortly after a similar library Lundi [16] had been abandoned. Sol came mostly from Rapptz's need to create a tiny metabuild system around "ninja" [17] for his academic assistance endeavors. During Sol's development, I joined the project and saw its potential, desiring to not only strengthen its implementation but heavily upgrade its feature set. Rapptz no longer had the time to maintain Sol, so after almost two years of repository and feature stagnation, alongside keeping my own personal fork alive, I officially endeavored to heavily improve and optimize Sol from January 2016 onward. The rest of this paper discusses design choices for, implementation trials of, and comparisons to Sol^2.

1. According to the TIOBE index since 2011, after the introduction of C++11 and C++14. http://www.tiobe.com/tiobe_index?page=index

```
    lua_getglobal( L, "my_number_cruncher" );
    lua_pushstring( L, "test 40" );
    lua_pushnumber( L, 56 );
    lua_pushnumber( L, 280 );
    lua_call( L, 3, 1 );
    int value = (int)lua_tonumber( L, -1 );
    lua_pop( L, 1 );
```

Snippet 1. Lua C API to call a function with 3 arguments and a single return value.

```
    int value = lua["my_number_cruncher"]("test 40", 56. 280);
```

Snippet 2. Use of Sol^2 to do exactly the same as snippet 1.

2. Design and Methods

There were two tasks necessary to conduct this design study and comparison. The first was figuring out what a good API for Sol^2 is, and iterating to improve it. The second was taking that API and comparing it to the vast majority of APIs currently present, their feature set, and user desires (through issue/bug trackers and feature requests).

2.1. Designing Sol^2

Designing an API is both important and difficult. Much research, inquiry and study has been done on the subject and a good API has clear, tangible benefits [18] [19]. Since Sol^2 came from its predecessor Sol, the API was mostly complete. Rapptz's original design for Sol is well-segmented and separated, abstracting away stack operations like push (...) , get, check, and pop. The next part was realizing that Lua was a language built on garbage collection and reference counting, so the next logical primitive structure became reference. These then served as the building blocks for higher level primitives such as function, table, and usertype. These three types sum up almost exactly what Lua has to offer, and are essentially core to any higher-level wrapper that abstracts up Lua.

Next, features had to be chosen. Of importance in feature determination was using the work done by Sol^2's predecessors, Lundi and Rapptz's Sol, and their various issues and pull requests on their Github repositories. This made it clear what would be required to hit the Sol^2 milestone, and what features should be implemented for user desires beyond the typical abstractions over tables, functions, and getters/setters. Easy user-defined type support became a core feature that drove much of the innovation. LuaJIT support also became extremely important in the effort to make Sol^2 truly Lua-version-agnostic and to attract a large swath of users from becoming locked into other libraries that did not support it, such as Selene [20] and luacppinterface [21].

Finally, tantamount to Sol^2's design was the realization that we could replace the chatty, verbose plain C API with just-as-efficient wrappers. Consider the code in snippet 1.

Notice that both getting and calling the function is less lines of code. The compression ratio between cognitive overhead and stack management becomes even bigger when users want to start calling Lua functions with variables that are inside of the Lua runtime and not present in C++. "Spaghetti code" does not even begin to describe when users have to start nesting into tables even just 2 levels deep to get the things users want. This became a very important goal of Sol^2: how would Sol^2 reduce this cognitive overhead to an absolute minimum, so that a function feels like a function, a table can be accessed like a table, and a userdata behaves more like a class type?

To do this, evaluation of several different libraries, browsing dozens of code repositories, and combing through hundreds of messages on the Lua Mailing List were done. Direct developer surveys were not done, because access to a large and diverse pool of developer specifically working with the Lua C API was not something obtainable even within a few months: rather, it is much more convincing to look at how individuals actually used the Lua C API and what users have done with it to build up the library, rather than try to meet the needs of a survey [22].

2.2. Evaluation of Libraries

13 different libraries, including Sol^2 and the plain Lua C API, were used and check for some of the most common features users expect out of a Lua wrapper/binding library. Feature and API comparisons for both were done by attempting to use the code to write benchmarks and by examining the library's documentation and examples for additional cases. A feature table was composed for many of the desired API functionality demonstrated by looking at personal use, example code in books [23], issue trackers [2], game(engine)s [24], and reading *Programming in Lua* [25]. The following features were examined.

1 optional: Support for getting an element, or potentially not (and not forcing the default construction of what amounts to a bogus/dead object). Usually comes with std (:: experimental) :: optional . It's a fairly new class, so a hand-rolled class internal to the library with similar semantics is also acceptable

2 tables: Some sort of abstraction for dealing with tables. Ideal support is mytable["some_key"] = value, and everything that syntax implies (setting to create a key, for example)

2. For example, https://github.com/jeremyong/Selene/issues has many opened issues of users looking to get a better idea of how use the library

3 table chaining: In conjunction with tables, having the ability to do nest deeply into tables `mytable["key 1"]["key2"]["key3"]`. Note that this becomes a tripping point for some libraries: crashing if `"key1"` does not exist while trying to access `"key2"` (Sol^2 avoids this specifically when users use `sol::optional`), and sometimes it's also a heavy performance bottleneck as expressions are not lazy-evaluated by a library
4 arbitrary keys: Letting C++ code use userdata, other tables, integers, etc. as keys for into a table without dropping to the plain API
5 user-defined types (udts): C++ types given form and function in Lua code
6 udts - member functions: C++ member functions on a type, usually callable with `my_object:foo(1)` or similar in Lua
7 udts - variables: C++ member variables, manipulated by `my_object.var = 24` and similar syntax within Lua scripts
8 function binding: Support for binding all types of functions. Lambdas, member functions, free functions, in different contexts, etc...
9 protected function: Use of `lua_pcall` to call a function, which offers error-handling and trampolining
10 multi-return: returning multiple values from and to Lua
11 variadic/variant argument: being able to accept "anything" from Lua, and even return anything to Lua
12 inheritance: allowing some degree of subtyping or inheritance on classes / userdata from Lua
13 overloading: the ability to call overloaded functions, matched based on argument arity or type (`foo(12)` from lua calls a different function then `foo("bark")`).
14 lua thread: basic wrapping of the lua thread API; ties in with coroutine.
15 coroutines: allowing a function to be called multiple times, resuming the execution of a Lua coroutine each time

We used a very simple structure and functions to examine these benefits plus several defined features. The basic structure and functions used to extract these benefits are equivalent to snippet 3, with a myriad of additional code being tested and applied to see if the target library could perform as desired.

3. Results

Sol^2 was the most fully-featured library present, because as the paper author I had the ability to use the well-designed API of Sol^2 to quickly and swiftly implement several features that the others had, and more. The following are the lessons and caveats of the various Lua bindings we uncovered, and their design merits.

3.1. Feature Discussion

The following sections list the developer ease of use compared to the Lua plain C API, which is considered. Subjective classifications, from worst to best, are used: **Painful** - exceedingly difficult to learn and comes with its own serious pitfalls and traps; **Hard** - more difficult to learn than the plain C API; **Normal** - about as hard as the plain C API in learning and usage; **Easy** - easier to learn than the C API but with some usage caveats; **Cakewalk** - an entirely smooth experience in learning and using the library. Despite subjective labels, justification is provided for various library classifications below.

3.1.1. Kaguya - Cakewalk. A rather well-designed header-only library, Kaguya [26] inspired many of Sol^2's features and optimizations, including the `sol::coroutine` object and the `sol::table` Performance improvements. It features efficient `operator[]` syntax on its tables and comes with a variety of well-thought-out and useful features, such as intuitive multiple returns using `std::tuple` from C++ functions, argument-based overloading support, user-defined type handling, and much more. The structures used to wrap the `lua_State*` into something more usable provide script execution, global table handling, and various utility functions. The library is being actively developed at the time of writing. It employs a readme-style tutorial for its features, but lacks formal documentation. One of its strengths is that it features both a C++03 and a C++11 implementation, making it suitable to use in older settings if a developer is willing to take a dependency on the Boost [27] library for some portions (`Boost.Preprocessor` specifically).

Of note is that author *satoren* also implemented his own benchmarks [28] to help guide the improvement of his library. A bit later, Sol^2 and its previous version were added to the benchmarks. More libraries are being added as issues with builds get sorted out and developers make pull requests. The library author is responsive, addressing pull requests and issues within a few days, frequently in as little as 24 hours. This library had efficient abstractions for tables, coroutines, functions, and more, covering almost all of the desired features, save for *optional* support. Like some other wrappers discussed here, Kaguya makes *C++ member variables* bound to Lua accessible with Lua self-call syntax `my_object:var()` to get the value and `my_object:var( ... )` to set the value.

Performance measurements indicate Kaguya is among the faster of the libraries present. Kaguya can also be introduced to code bases incrementally: its `kaguya::State` type can be created from an existing `lua_State*` and provide access to all of the abstractions therein without destroying the state when it goes out of scope. This allows all of the power from Kaguya to be used without having to commit to a full-scale rewrite of the code.

Recommendation: This library should be used if C++11 or better is not an option, and is still a good library for C++11 and better (though there is better for those versions of the standard).

3.1.2. Lua-API++ - Normal. Lua-API++ [29] is compile-in-the-sources library. It was created and even presented with slides at a Lua user group. It would stand to reason

```cpp
#include <tuple>
#include <string>

struct basic {
    int var;
    basic() : var() {}
    int get() const { return var; }
    void set(int x) { var = x; }

    std::tuple<int, std::string> multi_return(int a, std::string b) { return{ a, b }; }
};

int basic_call(int x) { return x; }

std::string basic_multi_param(std::string a, bool x) {
    if (x) {
        return a;
    }
    return a + " (not true)";
}
```

Snippet 3. Basic classes and functions that were attempted to be bound and integrated into various libraries.

that its one of the more popular libraries, and claims to use C++11 internally to simplify both its interface and implementation. Unfortunately, its facilities as a Lua C API wrapper library are unintuitive and make little sense and are less easy than almost all the other libraries. This feeling is contributed to by incredibly sparse tutorials and murky documentation, which discusses internal classes and implementation details while not explaining simple things like how to properly bind functions that are not tied to a metatable entry with the `LUAPP_USERDATA` macro.

It somehow adds more boilerplate than it removes: in the end, users end up writing wrapper functions that take a `self` argument for the class type users wish to bind, creating menial wrappers that other libraries have proven time and time again can be done without such tedium. A user not only has to know about the underlying C API to an extent, they must also endure learning how Lua-API++ thinks the API should look, combining some of the worst parts of both worlds for dealing with binding C++ classes.

Thankfully, a `globalIndexer` structure was provided that supported table chaining and nested gets, making the library very usable in terms of being a good abstraction of the Lua table primitive.

Recommendation: There is better out there than this library. Were it the only library, it might be considered a decent alternative, but there are 11 other choices here, many with vastly better syntax, interface and quality.

3.1.3. Luabind - Easy. Luabind is one of the oldest frameworks around created by Rasterbar Software. It has survived over a decade of time and is one of the most forked frameworks on github as users and corporations from all around pepper it with their own implementations and features. This comparison uses a special version of luabind that had boost removed and some C++11 features added to it.

Unlike many other libraries, Luabind supports *member variable syntax* and even has wrappers to allow getter/setter functions to be turned into variables in Lua. This feature inspired support for similar in Sol^2, helping to provide a compatibility connection between users transitioning from Luabind to Sol^2. Another user opened up another feature request in Sol^2 from the way Luabind created userdata in Lua[3]: the desired syntax was `local object = classname()`, where as the Programming in Lua book advocated usage of `local object = classname.new()`. Ultimately, Sol^2 defaults to the Programming in Lua style, but provides the ability to create the Luabind-style one to ease transitioning over.

```cpp
lua_State* L = luaL_newstate();
luaL_openlibs(L);
luabind::open(L);
luabind::module(L)[
luabind::class_<basic>("basic")
    .def(luabind::constructor<>())
    .def_nonconst("set", &basic::set)
    .def_const("get", &basic::get)
    .def_readwrite("var", &basic::var)
];
if (luaL_dostring(L, "b = basic()\n"
                     "b.var = i\n"
                     "x = b.var\n")) {
    lua_error(L);
}
```

Snippet 4. A complete example for binding a class to luabind. Notice that users can access member variables with the 'a.b' syntax in Lua scripts.

Luabind has a few quirks in its API, like setting up a keyword to easily define "classes" in Lua script and using "." for nested table queries, requiring the user compose a full lookup string before passing it to Luabind to process and traverse down a table. It also seems to lack a dedicated table abstraction, but has several kinds of indexers on some of its types. This ultimately makes it easy to use, but certainly not as easy as Kaguya, Selene or Sol^2.

One unmentioned caveat is that Luabind ***requires*** the "require" Lua package when users call `luabind::open` to get started with the framework. This is why the call to `luaL_openlibs(L)` is present. This was an undocumented caveat

3. https://github.com/ThePhD/sol2/issues/75

Library	plain C	luawrapper	lua-intf	luabind	Selene	Sol2	oolua
optional	✳	✗	✓	✗	✗	✓	✗
tables	✳	✳	✳	✓	✓	✓	✳
table chaining	✳	✳	✳	✓	✓	✓	✗
arbitrary keys	✳	✓	✓	✓	✓	✓	✗
user-defined types (udts)	✳	✓	✓	✓	✓	✓	✳
udts: member functions	✳	✓	✓	✓	✓	✓	✳
udts: variables	✳	✳	✳	✳	✳	✓	✳
stack abstractions	✳	✓	✓	✓	✓	✓	✓
lua function from C(++)	✳	✓	✓	✓	✓	✓	✓
function binding	✳	✓	✓	✓	✓	✓	✓
protected function	✳	✗	✳	✳	✳	✓	✳
multi-return	✳	✗	✓	✓	✓	✓	✳
variadic/variant argument	✳	✓	✓	✓	✓	✓	✳
inheritance	✳	✓		✓	✓	✓	✳
overloading	✳	✗	✗	✗	✗	✓	✗
lua thread	✳	✗	✗	✗	✗	✓	✓
coroutines	✳	✗	✗	✓	✓	✓	✗
no-rtti support	✓	✗	✓	✗	✗	✓	✓
no-exception support	✓	✗	✓	✳	✗	✓	✓
Lua 5.1	✓	✓	✳	✓	✗	✓	✓
Lua 5.2	✓	✓	✓	✓	✓	✓	✓
Lua 5.3	✓	✓	✓	✓	✓	✓	✓
luajit	✓	✓	✓	✓	✳	✓	✓
distribution	compile	header	both	compile	header	header	compile

Library	lua-api-pp	kaguya	SLB	SWIG	luacppinterface	luwra
optional	✗	✗	✗	✗	✗	✗
tables	✓	✓	✗	✗	✳	✓
table chaining	✓	✓	✗	✗	✳	✓
arbitrary keys	✳	✓	✗	✗	✗	✗
user-defined types (udts)	✓	✓	✓	✓	✓	✓
udts: member functions	✓	✓	✓	✓	✓	✓
udts: variables	✳	✳	✗	✓	✗	✳
stack abstractions	✓	✓	✳	✗	✳	✓
lua function from C(++)	✓	✓	✓	✓	✓	✳
function binding	✓	✓	✓	✓	✓	✓
protected function	✓	✳	✳	✳	✳	✳
multi-return	✓	✓	✳	✓	✳	✳
variadic/variant argument	✓	✓	✳	✳	✳	✗
inheritance	✳	✓	✳	✓	✳	✗
overloading	✗	✓	✓	✓	✗	✗
lua thread	✗	✓	✗	✗	✓	✗
coroutines	✗	✓	✗	✗	✓	✗
no-rtti support	✗	✓	✓	✳	✓	✓
no-exception support	✗	✓	✓	✳	✓	✓
Lua 5.1	✓	✓	✓	✓	✗	✓
Lua 5.2	✓	✓	✓	✓	✓	✓
Lua 5.3	✓	✓	✓	✗	✓	✓
luajit	✓	✓	✓	✓	✗	✓
distribution	compile	header	compile	generated	compile	header

TABLE 1. FEATURES FOR 12 DIFFERENT LUA LIBRARIES: ✓ - COMPLETELY SUPPORTED — ✗ - NOT SUPPORTED — ✳ - PARTIALLY SUPPORTED

Partial support often means that it allows access in some obscure way or only supports parts of a feature, e.g. not allowing function objects (lambdas) but working with free functions. Support was determined by using the frameworks, reading issue lists, contacting developers and checking documentation. An updated version is kept it Sol^2's documentation: http://sol2.readthedocs.io/en/latest/features.html#category-explanations

and it caused quite the crash for a while before figuring it out. Somebody somewhere should document it, but the docs are not hosted on a place where pull requests can be sent.

Recommendation: This code base is good and individual open-source contributors have improved parts of it and remade forks over time, but best left to handle cases of legacy where transition or use of something better is entirely out of the question.

3.1.4. Simple Wrapper and Interface Generator (SWIG 3.0) - Normal. SWIG [30] is a generator that deals with parsing source files. It does not technically qualify as a Lua API wrapper because it wraps nothing about the API into an abstraction: everything users do with the Lua C API applies here, and it offers no abstractions on coroutines, threads, tables, or the stack. What SWIG does do *extremely well* is making available C++ classes and objects in Lua code. It does this by essentially requiring a pre-build step to generate the files (which only has to be done once, unless users change the source interface and modules definitions).

SWIG's abstractions for userdata and types in C++ are feature-rich and incredibly efficient. For that reason alone it is Normal to deal with, because once a user's classes are set in stone in C++ there is very little work users need to do for utilizing their classes in Lua script. However, it still does not provide much in the way of a basic hook for opening the modules users define for their C++ code, and after that the user is back to struggling with the Lua C API.

Recommendation: Use SWIG when there is a large body of preexisting C++ code and a handful of well-defined classes must be made available to Lua scripts. Avoid use if an actual featured API wrapper is the goal.

3.1.5. luacppinterface - Easy. Started a number of years ago, luacppinterface [21] is a fairly good inter-operation library that supports the concept of a `LuaTable`, `LuaFunction`, and several other useful primitives. It's only flaw is that the primitives have their constructors hidden from the user, meaning that manual operations done by utilizing and combining the basic primitives becomes a bit difficult to do. Nevertheless, the library author has created quite a useful

The library's more updated branches also suffer from the same problems as § 3.1.7 luwra's tables: the initial type checkers do not accommodate the fact that metatables can make tables and userdata behave like functions. Therefore, luacppinterface's own implementation of functions (a userdata with a `"__call"` metatable key overriden) crashes the code on that branch. Reverting back to the `master` branch allowed benchmarking and test functionality to continue, but that has filled the build with yet more warnings.

While it struggles with performance in some areas, in others the library is surprisingly fast, particularly regarding its implementation of metatables for its userdata. It uses a self-referencing table trick to keep the current lookup table in the cache for when the table looks for the key of the value: this leads to an incredibly fast lookup for member functions. Unfortunately, the price of this implementation is that member variable support becomes impossible. The library also has not updated itself to handle LuaJIT, cutting it out of a fairly large portion of user's desires.

Recommendation: luacppinterface is superseded by other libraries, despite its usefulness. It does not have enough features and does not implement them in a novel enough or efficient enough way to justify it over any of the other choices here. Still, it is not the worst choice a user can make.

3.1.6. luawrapper - Easy. luawrapper [31] is the most minimal implementation here, even compared to § 3.1.7 luwra. It has a more functional API, relying on function calls and constructors of `std::` types to create and manipulate tables. The primary functions are `LuaContext::readVariable` and `LuaContext::writeVariable`. It's approach to standard types means that, because there is no "object" abstraction, it suffers from a heavy dose of verbosity that can only be changed by `typedef`s (see snippet 5).

Nevertheless, it tends to be simple for basic use cases, and supports a wide variety of features, including variant arguments and variant return types (with `boost::variant`). The code is handled by PowerDNS since the company took a dependency on it long ago before the original author dropped the project entirely. Sol^2's developer fixed an internal compiler error when using luawrapper with Visual Studio and shipped the fix to them in a pull request. Hopefully, it will be integrated back into the original code.

Recommendation: It is a nice library, but with it being held onto by a company that puts its needs over that of the general userbase and has deadlines and goals not related to making it the best it could be, it is best to just leave this one as support for the developers who used it before its abandonment. It is effectively in "maintenance mode" at time of writing.

```
LuaContext lua;
lua.writeVariable("ulahibe",
    std::map<std::string, std::map<std::string
        , int>> {
        { "warble",
            std::map<std::string, int>{
                { "value", 24 }
            }
        },
    }
);
```

Snippet 5. luawrapper code to write a nested table. This does not compose well or scale up.

3.1.7. luwra - Easy. Previously, luwra [32] was too lightweight to be considered a wrapper. It more-so just had a few macros implemented to make calling functions easier and a slim state wrapper, did not provide any additional functionality and did not accommodate complex scenarios. Over the last 3 months, author Ole Krüger added several more features. It has the basics of a good Lua wrapper and a few others, maintaining very fast performance but sacrificing some useful features to do so.

For example, C++ function objects (e.g., Lambdas, or anything that overrides operator()) that have state are not usable unless they're made `constexpr`. Multiple argument push and pop are still in flux. And, it is impossible to get out a `std::function` without first getting an intermediate, making it hard to interop with functions that take `std::function` as a parameter. Variant returns are missing and multiple returns do not compile.

Still, the library has potential to be good. A good core design is there, even if there are several corner cases and caveats the library doesn't consider. As an example mistake made by more than just luwra, when creating a `luwra::Table` type, the internal check only asks if it is a table. However, due to metatables in Lua, those same abstractions and Lua C API functions called by luwra work on userdata as well. If someone wanted to treat a userdata as a table and index into it, luwra would not let them do it. This is a mistake many more libraries that implement type checking encounter (such as snippet 3). It has basic documentation as well with usage examples, which make utilizing the library simple. Nevertheless, it's speed is still within a fraction of Sol^2 and Kaguya.

Recommendation: The developer is active (as recent as May 15th, 2016) and is adding features. Its performance is good (except for nested table keys and values), but that seems due to the fact that it is ignoring some of the harder parts of creating a Lua wrapper (such as not supporting arbitrary functors and lambdas). It requires C++11, but not having support for the most notable feature of C++11 (lambdas) dissuades from usage.

3.1.8. OOLua - Hard. OOLua [33] is a well-meaning library that suffers from trying to use a Domain-Specific Language to control much of the functionality of the framework. The framework has configuration flags for turning off C++11 features, and in doing so must be able to compile with C++98. Therefore, the author willingly chose to invent a domain-specific language (DSL) that requires a significant amount of cognitive overhead to learn and use.

This DSL must be registered in the top-level namespace and is macro-based. Thankfully, for all of the trouble it puts the user through, the framework and DSL are powerful enough to accommodate member variables, which means already it supports more than several other libraries. However, even the DSL requires a degree of repetition that is not fully justifiable when there are other frameworks that do it in less code and with less weird oddly specific macros. For example:

```
OOLUA_PROXY(basic)
        OOLUA_MGET_MSET(var)
        OOLUA_MFUNC_CONST(get)
        OOLUA_MFUNC(set)
OOLUA_PROXY_END

OOLUA_EXPORT_FUNCTIONS_CONST(basic, get)
OOLUA_EXPORT_FUNCTIONS(basic, set)

OOLUA_CFUNC(basic_call, oo_basic_call)
```

Snippet 6. Macros to setup usertype bindings and a function binding in OOLua

Even then, one must deliberately call `OOLua::set_global( oolua_script, "my_func", oo_basic_call)` to have the function you just declared further up as a type. luwra may have macros, but those macros understand their purpose: less typing, less cognitive overhead, more efficiency. OOLua demands you create elaborate setups to set up your functions and userdata, but does not provide performance more competitive than either luwra, Kaguya, or Sol^2. It is efficient, yes. While it is *technically* less written work than specifying the metatable by yourself in Lua script or the Lua C API, the convoluted DSL is not helpful. Consider that the "const-ness" of a member function can be determined at compile-time VIA the templates used underneath in OOLua, and yet the export macros and member function macros still require that you specify it manually.

It has a table abstraction, but its API is so poor it may as well not have one at all. Not only does OOLua demand that any variable it gets is default-initialized first, the library also retrieves it as an out-parameter:

```
using namespace OOLUA;
Script vm;
// initialize some table in lua,
// e.g. ulahibe = {warble={value=24}}
// ...

int v; // what we want to get
Table one_deep;
Table two_deep;
get_global(vm, "ulahibe", one_deep);
tu.at("warble", two_deep);
tw.at("value", v);
```

This verbosity is entirely unnecessary and also incredibly error-prone. There's no way to compose the function calls together with output parameters, and if someone accidentally reuses a table they could very well corrupt memory. Contacting the OOLua developer, he has explained me that he advises individuals do not use the Table abstraction at all; this is fine advice since the `OOLUA::Table` class is neither efficient and operates in a dated fashion. The library asserts that it efficiently "[covers] the problem with clever template metaprogramming and macros"[4]. OOLua's approach is quite demonstrably too clever, and this library only manages to justify the gripes of the article the website links to that uses those very same words.

Recommendation: Avoid using the library. While it is functional and has a few features, it is not so overly efficient overall or novel that it deserves the learning time required to get used to its heavily quirky DSL that goes a lot deeper and becomes quite verbose with its tags and syntax. Nevertheless, it is worth noting it is performant in the case of member functions. It's table abstraction, however, can be considered a way to *NOT* to wrap up a Lua API, and the DSL, while being performant, is not helpful for developers who do not want to repeat themselves twice or more for OOLua.

3.1.9. Selene - Normal. Selene [20] is by far the most popular of the wrapping and binding libraries to date. Started by Jeremy Ong, it's been around for quite some time and has achieved quite a high number of followers and more than half the number of forks of Luabind. Its interface is easy to understand, and there is a noticeable improvement in the way it handles class bindings over Lua-Intf (see snippet 7): in fact, Selene's class bindings are nearly identical to Kaguya.

Selene takes the interesting approach of making everything based on `operator[]` with a `sel::Selector` class that the author wrote many blogposts about. This makes everything feel and index like a table.. even if there is *absolutely no table type*. So, Selene can still claim that it supports tables even if it has no dedicated table class, because nearly everything that's set and returned from Selene is something that users can index into. Selene would be a perfectly fine framework, but its performance is dismal.

4. urlhttps://oolua.org, May 17th, 2016

This is one of those frameworks that looks very nice but has truly heaped so much boilerplate under the hood (or made the code so convoluted not even a powerful implementation has the power to) that the one true optimization – inlining – cannot save its performance metrics. It is perplexing that this has gone unnoticed for such a long time and the library has become so wildly popular: almost nobody seems to question that Selene's performance is nearly the worst for the majority of its categories, ***especially*** for table-chaining. The expectation is that the indexing type `sel::Selector`, which Jeremy Ong spent an entire blog post writing about in the context of r-values and traversal, would be efficient. But it is the single most boilerplated piece of code in the entire library.

Thusly, because of this gross degree of over-engineering without thought to the actual performance of the system, Selene is ranked as Normal, if only to dissuade individuals from using it. It does not support LuaJIT. In the library's defense, however, its design has inspired many other libraries to consider the merits of a cleaner interface using C++11: Rapptz knew of Selene's existence as he continued to work on the original Sol, and other library writers have taken note of Selene's interface. As an academic exercise, it succeeded in getting individuals to think about how users could have a clean interface on top of the Lua C API.

Recommendation: As nice as the API is, use something else. While a nice idea, the implementation suffers from over-engineering and handcrafted overhead that other C++ libraries like Kaguya, luwra and Sol^2 solve, much more efficiently and much more practically.

3.1.10. Lua-Intf - Easy.

Created by Steve K. Chiu, Lua-Intf [34] is a rather functional library that can be seen as the API design bridge between older libraries like Luabind and newer libraries like Selene or Kaguya. It features a module userdata binding syntax that's easy enough to use, even if the syntax is not as terse as it could possibly be due to the usage of a builder-style "begin module / end module" syntax.

```cpp
using LuaIntf;
LuaContext lua;
LuaBinding(lua).beginClass<basic>("basic")
	.addConstructor(LUA_ARGS())
	.addFunction("set", &basic::set)
	.addFunction("get", &basic::get)
	.addVariable("var", &basic::var)
.endClass();
```

Snippet 7. Binding the basic example from snippet 3 in Lua-Intf

While these bindings are verbose, they do enable the user to cover a lot of use cases, even if there's a few features missing from the Feature Table (see table 1). The framework stumbles over itself in the case of getting variables, requiring a `LuaContext.getGlobal( ... )` call before a user can begin to use `operator[]`. This incongruence is strange, but certainly not impossible to deal with. It also supports getting variables using the "." syntax within a string, giving it the same string-processing overhead luabind suffers as it has to read the string. This implementation does build optional support into LuaRef: traversal down a chained query and returns a `nil` reference object.

It is also lacking in abstractions for Lua's threads and coroutines, with issues stating that another abstraction can be used to obtain the necessary features. Nevertheless, in the code, there are functions for checking if a `LuaRef` that may be pointing at a thread is yieldable, and a resume call as well, so there is still some low-level support.

Recommendation: A good framework, but Kaguya and Sol^2 still offer more features with better performance. It is not the worst choice one can make, but it is certainly not the best and those class type bindings can get quite heavy.

3.1.11. Simple Lua Binding (SLB3) - Painful.

SLB3 was not going to be compared and contrasted here until an e-mail from OOLua developer Liam Devine came in about the feature table constructed for Sol^2. He asserted that SLB3, SWIG, and OOLua were among the fastest in his tests and could be used to great achievement. Thusly, here is the analysis of SLB3:

The worst of all the libraries present, SLB3 [35] is a derelict codebase from several years ago, abandoned and provided no fixes for the many bugs opened over 6 years ago. The code is found on a repository automatically exported from Google Code. Despite having documentation, inability to compile in any recent iteration of the major compilers or with the latest version of Lua caused a number of problems even benchmarking or using SLB3. Tweaking compiler flags and changing how exports were done, the code was able to compile but even after being re-compiled for the target platform alongside the application and with Lua, the library inexplicably crashes on both GCC's g++ and Microsoft's Visual C++ without any kind of indication as to what went wrong. Specifically, it bypasses even Visual C++'s debug message boxes when `abort` is called and no exception can be trapped. The code simply just crashes and exits. Not even Lua's `at_panic` function gets called. Debugging the problem requires stepping through the code to figure out what went wrong. At this point, it is not worth working with the library for the instances where it does this.

Investigating the crashes gave the opportunity to also read through the code and try to glean proper usage from its outdated Wiki. Even when following the Wiki, crashes happened in the middle of said example code. What is more baffling is the usage of raw pointers with unclear ownership semantics for things. Coupled with a lack of descriptive names that offer little to no understanding of the functionality of the class itself (the notorious "enterprise-ready" `SLB::Manager` class, that tells a user nothing about what it manages).

It has no notion of a table either, providing basic set/get functions. How a user accesses nested values is anyone's guess: the documentation does not say, and various attempts resulted in crashes, wasting a significant chunk of research time trying to get the thing to work.

SLB3 is also incredibly ashamed of Lua itself: it attempts to hide Lua's primitives from users everywhere in the API. Large chunks of functionality are stored in the `SLB`

::Script abstraction, which has its own lua_State* that it keeps protected from the user. This means that in order to extend the functionality with one's own lua hooks, a user needs to first derive from SLB::Script and either expose the data or play entirely by the hooks, setup, and rules of SLB3. In the vein of setup, multiple setup functions are required before using SLB3 if a user is trying to plug it into an existing system, adding more boilerplate. This makes SLB3 the wrong choice for incremental and small cleanup of C++-Lua inter-operation code without incurring severe developer and maintenance overhead.

Recommendation: *Do not use*. There are vastly more well-maintained libraries. SLB3 deserves to fade into the obscurity it has achieved over the years. A user could be forgiven for developing a bad stereotype of all Lua API wrappers based solely on this one.

3.1.12. Sol^2 - Cakewalk.

As this paper is authored by the developer of Sol^2, this section will obviously be contain bias. However, it must be said that Sol^2 provides a number of benefits in API design and efficiency that other libraries do not, and remains one of the most efficient libraries among all categories for the group.

With fully-featured abstractions such sol::table; sol::function and sol::protected_function ; sol::coroutine; sol::thread; sol::userdata; and sol::lightuserdata , it is fairly clear Sol^2 covers almost every single specific type that Lua offers with a higher-level covering. Abstractions for the debug framework are missing, and a potential TODO would be the at least providing enough of an abstraction on top of that for a lua-based stack-trace when the default panic function is called. Its simplicity is proven by the praise that Lua veteran Sean Conner on the Lua mailing list gave Sol^2 when he was given an example [36]. Jason Turner commented during a C++ Now 2016 scripting discussion that he was "super impressed".

"For ease of use and performance it's really quite good. I'd say it's the second best scripting option for C++ :wink:" – Jason "Lefticus" Turner, creator of ChaiScript

May 16, 2016

Reception on reddit was also overwhelmingly positive [37]. Sol^2 even attracted users away from other libraries once enough documentation and tutorials existed to demonstrate just how fully featured Sol^2 is. Our usertype abstraction is also one of the most terse, going with a key-value style of additions with various wrappers to provide functionality akin to Luabind's properties, Luabind's factory/constructor calls, and read/write only memory from SWIG.

```
sol::state lua;
lua.new_usertype<basic>("basic",
      "var", &basic::var,
      "get", &basic::get,
      "set", &basic::set
);
```

Snippet 8. Sol^2 usertype example, performing the same binding for the basic class as shown with many of the other libraries. Noticeably more terse.

Libraries that have started 2-4 years before Sol^2 have less stars and less watchers, with Sol^2 breaking 160 stars with small, steady growth continuing (with the exception of the extremely popular Selene). Additional advertising on the lua mailing list and perhaps a talk about the optimization and implementation of Sol^2 are in the works. Documentation hosted at ReadTheDocs [38] has proven immensely useful, allowing users to make simple pull requests to change documentation with automatic updates if they accept. Iris Zheng, a student at Columbia University, was extraordinarly helpful in improving Sol^2's documentation for beginners, making it much easier to get those who were not entirely used to C++ and its many environments to pull the header-only library and get started.

Sol^2 is Lua-version-agnostic. So long as some form of LuaJIT or Lua 5.1 or above are in place, Sol^2 will work without problem. This is ensured by tests written under the Catch [39] framework.

Recommendation: Use Sol^2. Issues are gladly accepted at the repository. Turnaround time for responses is less than 24 hours, sometimes with fixes and implementations coming around that fast as well. Development is active and additional performance enhancements are already planned.

3.2. Performance Comparisons

Aside from satoren's benchmarks (see fig. 1), there is only raw data available at the moment concerning performance of *all* thirteen APIs. The data and benchmarks are available online[5]. The additional data needs to be massaged into a clean representation using python's matplotlib from the "lua crunch" project in that repository (pull requests welcome). From both satoren's benchmarks and lua-bench's initial CSV numbers generated from statistical microbenchmarking framework nonius [40], Kaguya, luwra, and Sol^2 seem to be the most highly performing across all areas, with Selene quite demonstrably being the worst optimized of the group.

4. Conclusion

From our analysis of over a dozen wrapper APIs, it is clear that there are really only 2 choices to use when developing in C++ and one needs to reach out for Lua wrapper to help them embed the language in their application/system. Kaguya and Sol^2 provide the most features, abstractions, and speed to date, with Kaguya filling the niche of compatibility and Sol^2 filling the niche of heavily

5. Additional benchmark data that has not yet been massaged into graphs and covers greater detail than the benchmarks shown in this paper are at https://github.com/ThePhD/lua-bench/tree/master/lua-results. The overall benchmarking code and projects can be found at https://github.com/ThePhD/lua-bench.

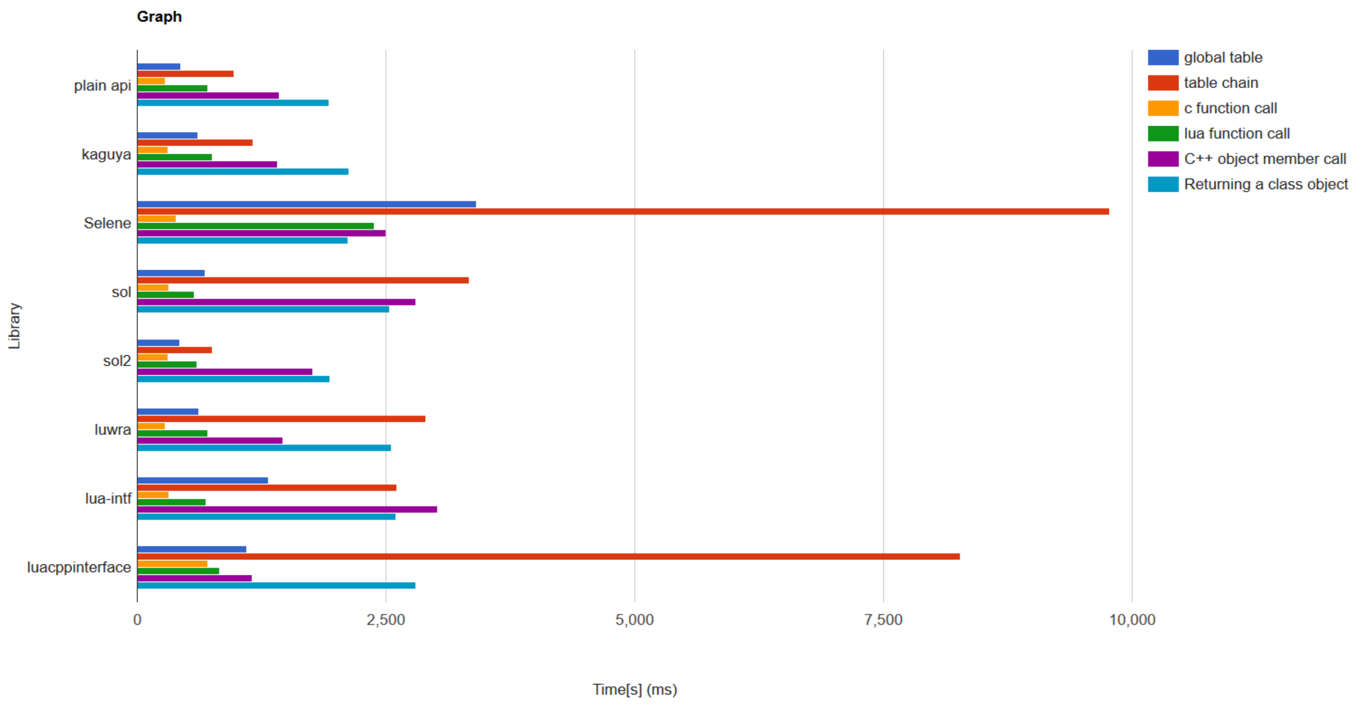


Figure 1. EARLY BENCHMARKS created by satoren, developer of Kaguya. Here, various libraries for a given color-coded task are shown, giving the total execution time for 5,000,000 (5 million) iterations. We are doing incredibly well compared to many other libraries and wrappers: http://satoren.github.io/lua_binding_benchmark/.

featured. Both are efficient and performant enough, even if there is some evidence for Sol^2 being faster.

The most important part of this project has been learning that ***documentation is key***. The reason SLB3 is a pain to use is because the documentation is outdated and sparse, along with the developer dropped off the face of the earth. luacppinterface was equally difficult to grok until looking at its tests to understand how it was used, as well as other individuals who used it. Selene and Kaguya had readme-style, shamelessly quick, code-snippet-examples style documentation that had users up and running in minutes, resulting in a confidence that kept users coming to both wrappers and continuing to use them. Usable, helpful documentation is a premium in the software development world, far more than unit tests or continuous integration build badges.

Creating those docs three months ago from the time of writing (May 2016) was a massive boon. This user friendliness was further compounded by Iris Zheng's help, alongside several Github user's input that truly shaped up a decent tutorial. At one point some of our potential users *thought* that Sol^2 had more features, but they were not sure because they could not get to our documentation. This proved an important lesson on making sure users got to our documentation, and hosting it on ReadTheDocs helped that happen swiftly. Sol^2 is ready to change how developers look at scripting language APIs in C++ for a long time to come, and I sincerely hope it becomes the golden standard for Lua inter-operation in C++.

Acknowledgments

A humongous Thank You to: Professor Gail Kaiser to letting the focus of COMS E6156's Topics in Software Engineering be about the use and improvement of Sol^2; Ms. Iris Zhang for using, improving, and vetting the documentation; Kevin Brightwell (Nava2) for Travis CI build improvements; Sean Conner for his help on the Lua Mailing List; the various Github issue openers and contributors for improving documentation, suggesting features, and catching bugs in Sol^2; Danny Y. (Rapptz) for starting and implementing Sol; and, the Lounge<C++>.

References

[1] ThePhD. Sol2. [Online]. Available: https://github.com/ThePhD/sol2

[2] P. C. U. of Rio de Janeiro in Brazil. Lua. [Online]. Available: https://www.lua.org

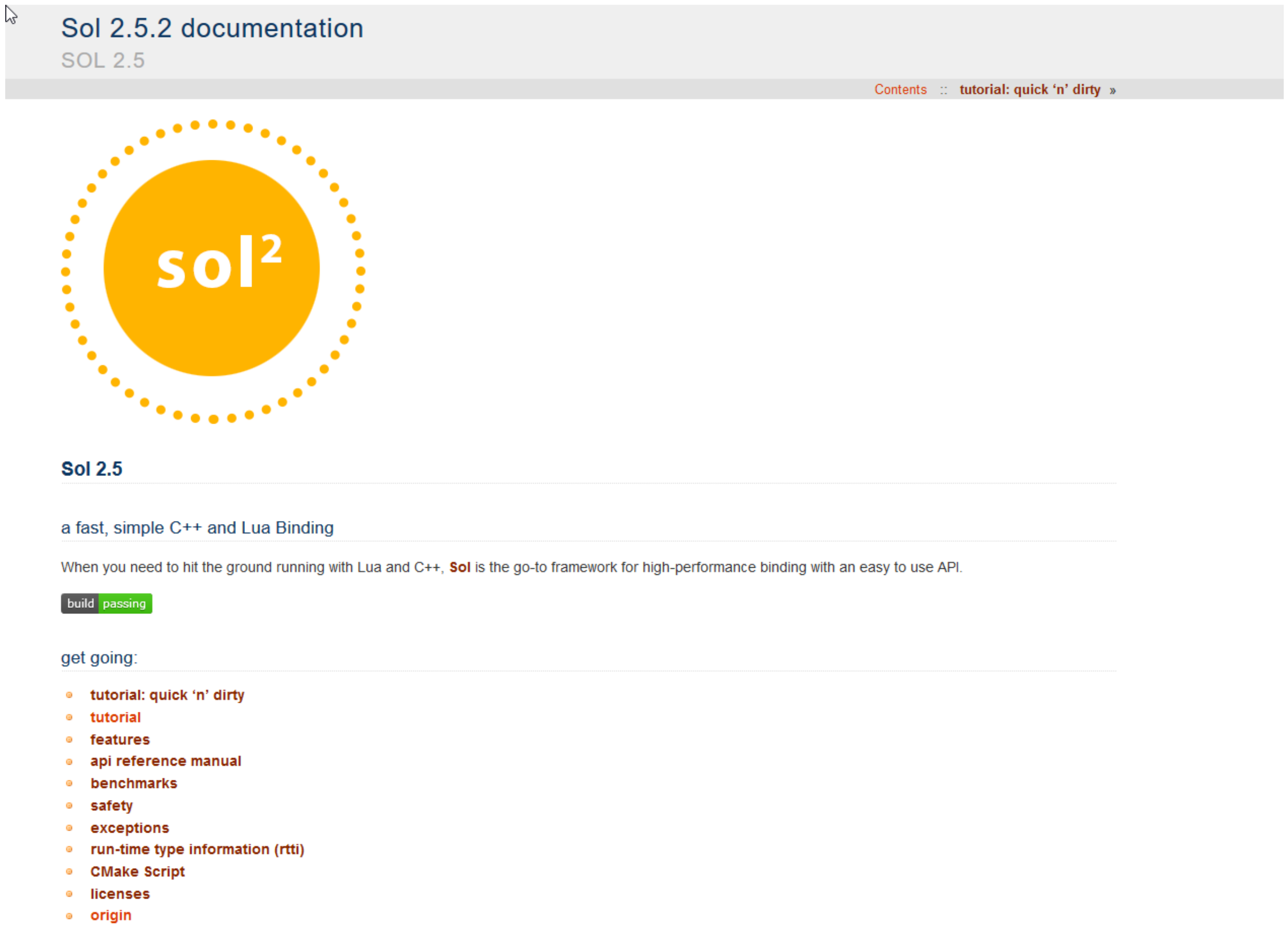
[3] P. Schuytema and M. Manyen, *Game Development With LUA (Game Development Series)*. Charles River Media, Inc., 2005.

[4] T. Niemüller, A. Ferrein, and G. Lakemeyer, *A Lua-Based Behavior Engine for Controlling the Humanoid Robot Nao*. Springer, 2010, pp. 240–251.

[5] A. Hiischi, "Traveling light, the lua way," *IEEE Software*, vol. 24, pp. 31–38, September 2007.

[6] P. C. U. of Rio de Janeiro in Brazil. Lua. [Online]. Available: https://www.lua.org/about.html

[7] Z. Ádám Mann, "Three public enemies: Cut, copy, paste," *IEEE Computer*, vol. 39, July 2006.


Sol 2.5.2 documentation

SOL 2.5

Contents :: tutorial: quick 'n' dirty »

Sol 2.5

a fast, simple C++ and Lua Binding

When you need to hit the ground running with Lua and C++, **Sol** is the go-to framework for high-performance binding with an easy to use API.

build passing

get going:

- tutorial: quick 'n' dirty
- tutorial
- features
- api reference manual
- benchmarks
- safety
- exceptions
- run-time type information (rtti)
- CMake Script
- licenses
- origin

Figure 2. THE DOCUMENTATION of Sol^2 hosted by ReadTheDocs: http://sol2.readthedocs.io/en/latest/.

[8] I. L. Taylor. Write gcc in c++. [Online]. Available: http://airs.com/ian/cxx-slides.pdf

[9] I. L. T. Linda Jacobson. Gcc's move to c++. LWN. [Online]. Available: https://lwn.net/Articles/542457/

[10] Nox. Binding lua to c++: Think twice before eating that glue. purplepwnystudios. [Online]. Available: http://purplepwny.com/blog/binding_lua_to_c_think_twice_before_eating_that_glue.html

[11] T. Mensch. Fun lua bindings. QuickCharge Games. [Online]. Available: http://realmensch.org/blog/fun-lua-bindings

[12] S. P. Conner. How close to c do you make your binding? lua-l-mailing-list. [Online]. Available: http://lua-users.org/lists/lua-l/2014-06/msg00435.html

[13] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 389–398.

[14] R. Software. luabind. [Online]. Available: http://www.rasterbar.com/products/luabind.html

[15] Rapptz. Sol. [Online]. Available: https://github.com/Rapptz/sol

[16] B. Banachewicz and A. Bellec. Lundi. [Online]. Available: https://github.com/lundiorg/lundi

[17] E. Martin and Ninja-Build. ninja. [Online]. Available: https://github.com/ninja-build/ninja

[18] M. Reddy, *API Design for C++*. Elsevier, 2011.

[19] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 506–507. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176622

[20] J. Ong. Selene. [Online]. Available: https://github.com/jeremyong/Selene

[21] D. Siaw. luacppinterface. [Online]. Available: https://github.com/davidsiaw/luacppinterface

[22] A. D. Andre and C. D. Wickens, "When users want what's not best for them," *Ergonomics in Design: The Quarterly of Human Factors Applications*, vol. 3, no. 4, pp. 10–14, 1995.

[23] Filip and R. P.-A. Lundgren, *CryENGINE Game Programming with C++, C# and Lua*. Packt Publishing Ltd., 2013.

[24] I. MacLarty. Amulet. [Online]. Available: https://github.com/ianmaclarty/amulet

[25] R. Ierusalimschy, *Programming in Lua*. Roberto Ierusalimschy, 2006.

[26] satoren. Kaguya. [Online]. Available: https://github.com/satoren/kaguya

[27] B. Dawes and D. Abrahams. Boost c++ libraries. boostorg. [Online]. Available: http://www.boost.org

[28] satoren. Lua binding benchmark. [Online]. Available: http://satoren.github.io/lua_binding_benchmark/

[29] OldFisher. Lua api++. [Online]. Available: https://github.com/OldFisher/lua-api-pp

[30] D. B. Mark Gossage. Swig: Simple wrapper and interface generator. [Online]. Available: http://www.swig.org

[31] P. Kreiger. luawrapper. [Online]. Available: https://github.com/ahupowerdns/luawrapper

[32] O. Krüger. luwra. [Online]. Available: https://github.com/vapourismo/luwra

[33] L. Devine. Oolua. [Online]. Available: https://oolua.org/

[34] S. K. Chiu. lua-intf. [Online]. Available: https://github.com/SteveKChiu/lua-intf

[35] xgene. slb3. [Online]. Available: https://github.com/xgene/slb

[36] S. P. Conner. Re: Modern lua binding library. lua-l-mailing-list. [Online]. Available: http://lua-users.org/lists/lua-l/2016-03/msg00160.html

[37] ThePhD. Sol2: Lua ¡-¿ c++ binding framework. reddit. [Online]. Available: https://www.reddit.com/r/cpp/comments/4a8gy7/sol2_lua_c_binding_framework/

[38] E. Holscher, C. Leifer, and B. Grace. Home - readthedocs. ReadTheDocs. [Online]. Available: https://readthedocs.org/

[39] P. Nash. Catch. [Online]. Available: https://github.com/philsquared/Catch

[40] R. M. Fernandes. nonius. [Online]. Available: https://nonius.io/