

# To Rust or Not To Rust: JVO での経験


ザパート・クリストファー

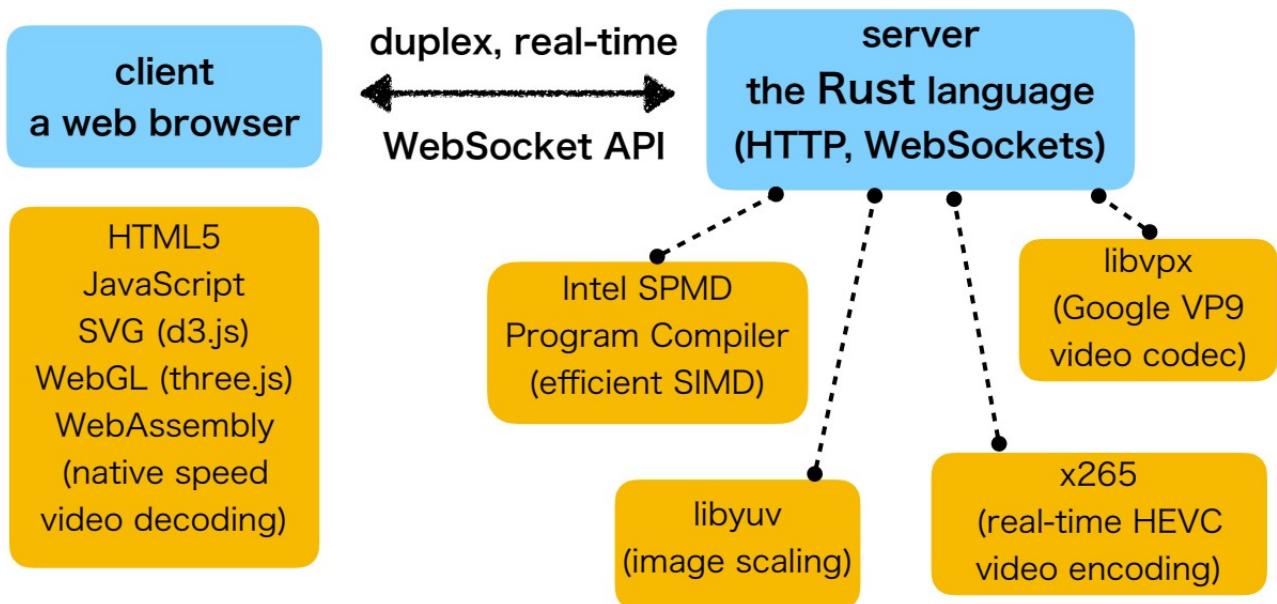
(特任専門職員、天文データセンター、国立天文台、三鷹、日本)

## 概要

At the Japanese Virtual Observatory we have experimented with the Rust programming language. Rust promises the following benefits: improved stability, memory safety and "fearless concurrency". The existing FITSWebQL server software has been ported from C/C++ to Rust. We have found improved performance in some places, mainly thanks to better CPU load balancing of the Rust rayon data parallelism library compared with C/C++ OpenMP. The use of an integrated HTTP/WebSockets actix-web networking library in Rust has made it possible to offer more responsive streaming downloads of partial FITS cut-outs. Apart from performance, the clear advantage of Rust over C/C++ is its superior stability and reliability in a 24-hour server environment. The Rust programming language itself also results in programs containing fewer bugs.

## 1. Introduction

The FITS WebQL server software in operation at the Japanese Virtual Observatory, NAOJ, enables users to view even over 100GB-large FITS files in a web browser running on a PC with a limited amount of RAM, without ever having to download the underlying FITS files. Users can interactively zoom-in to selected areas of interest with the corresponding frequency spectrum being calculated on the server in near real-time. After previewing FITS files users may choose to download interesting FITS files either in whole or to stream a partial region-of-interest (cut-out) from the JVO server to their own computers. The client itself (a browser) is a JavaScript application built on WebSockets, WebAssembly, HTML5, WebGL and SVG (shown in  1). In 2018 the current version 4 - completely re-written from scratch in the Rust programming language - had been released, featuring real-time streaming videos of individual frequency channels from FITS data cubes. The service can be accessed from the JVO Portal, found at <https://jvo.nao.ac.jp/portal/top-page.do> . The latest version 4 of the software (which includes the standalone personal desktop edition) is also freely available for download from the following GitHub repository: [https://github.com/jvo203/fits\\_web\\_ql](https://github.com/jvo203/fits_web_ql) .



☒ 1: A client-server architecture of FITSWebQLv4.

## 2. Why Rust?

The new version 4 initially started as a small feasibility study to find how easy it would be to re-implement the server part of FITSWebQLv3 in Rust. Another reason for doing a Rust re-write was the need to clean up the original C/C++ code as it has grown too complex to follow and too difficult to add new functionality. There are good reasons for switching from C/C++ to a new systems programming language such as Rust as it brings important benefits such as memory safety (*no memory leaks, dangling pointers*), thread safety (*no data races*), better (smoother) multi-threading (*"fearless concurrency"*) compared with OpenMP in C/C++ and a complete lack of segmentation faults (*no crashes*) due to inherent safety measures built into the Rust language. It is certainly possible to write C/C++ programs that are free of memory leaks and do not crash but from a programmer's standpoint Rust makes accomplishing these tasks much easier, all without sacrificing performance. Although the so-called standard *"safe"* Rust does not crash by itself in the face of programming errors (out-of-bounds array accesses etc.), when using the **"unsafe"** keyword to call external C/C++ libraries Rust cannot prevent bugs/data races present (potentially) in those external dependencies from hard-crashing the main Rust program. One needs to be very careful in the choice of external C/C++/Fortran libraries to call from Rust. However, this should not count as a disadvantage in comparison with C/C++ since those same buggy external libraries would cause the same segmentation faults if called from an equivalent C/C++/Fortran program. As an example, the C/C++ FITSWebQL versions 2 and 3 were launched on the server from within an infinite loop in a bash script that would quickly spawn a FITSWebQL process in case of a segmentation fault:

```
#!/bin/sh
while [ 1 ]; do
    DATE=$(date +"%Y%m%d%H%M%S")
    cd /home/chris/FITWebQL
    ./almawebql 1>logs/${DATE}.log 2>logs/${DATE}.err
done
```

This was especially useful during an early phase when not all rarely-triggered-but-critical bugs/errors had been found yet (i.e. mmap causes a hard segmentation fault upon bus errors, NFS mounts can go down due to network problems etc.). The Rust version 4 of FITSWebQL simply does not need to be called in such a way from within an infinite loop since the code is much more reliable.

Another important consideration in scientific computing is performance. Typically Rust programs run nearly as fast as C/C++ and certainly much faster than Java. Unlike Java, Rust does not suffer from any garbage collection freezes (like C/C++ there is no GC in Rust, memory is released as soon as variables go out of scope), and it takes advantage of the jemalloc memory allocator that is faster/more efficient compared with the default system memory allocator. Another excellent feature of Rust is the integrated **Cargo** package manager that helps to keep track of different versions of various external Rust package dependencies needed by a particular Rust program. It is extremely easy to downgrade external dependencies in case there are issues with newer versions. In contrast, working with traditional programming languages would involve manually downloading/compiling an external dependency, dealing with library paths, long Java classpaths etc. In Rust everything is handled automatically by Cargo.

### 3. C/C++ versus Rust feature comparison

C/C++	Rust
mutable by default (a <b>const</b> keyword is needed to prevent accidental data manipulation)	immutable by default (all variables are constants), an opt-in (let <b>mut</b> x = ...) is needed to enable subsequent writes
variables can be written to by another thread without any synchronisation	threads/functions take ownership of variables (only one owner at a time can write)
by default a lax compiler, beware of unexpected compiler bugs	the Rust compiler (borrow checker) is extremely strict; initially it may take a long time (mental gymnastics) to get a code to compile

the compiler does not catch any common memory bugs, a programmer needs to maintain a high state of alertness at all times, external memory-checking tools like <b>valgrind</b> are needed	the strict compiler helps prevent many common programming mistakes, dangling pointers etc., resulting in safer programs with fewer bugs
a fast auto-vectorized code either with the paid- for Intel C/C++/Fortran compiler or a free Intel SPMD Program Compiler <a href="https://ispc.github.io/">https://ispc.github.io/</a>	the default auto-vectorization can be hit-or-miss, easy integration with the Intel SPMD Program Compiler via a <b>ispc-rs</b> Rust crate (package) <a href="https://github.com/Twinklebear/ispc-rs">https://github.com/Twinklebear/ispc-rs</a>
error/exception handling an after-thought; it is easy to skip error checks during prototyping and then omit/forget to add proper error handling during production	forces a programmer to decide how to handle errors at every step, resulting in more reliable programs
excellent Parallel STL with C++17/20, easy parallelism with OpenMP	excellent data parallelism library <b>Rayon</b> <a href="https://github.com/rayon-rs/rayon">https://github.com/rayon-rs/rayon</a>
using OpenCL for <b>GPGPU</b> may be a bit cumbersome (a lot of low-level plumbing)	low-level OpenCL complexity is hidden from the end-user in an easy-to-follow Rust <b>ocl</b> crate (package): <a href="https://github.com/cogciprocate/ocl">https://github.com/cogciprocate/ocl</a>
WebAssembly with Emscripten: <a href="https://emscripten.org/">https://emscripten.org/</a>	native WebAssembly (Wasm) support: <a href="https://github.com/raphamorim/wasm-and-rust">https://github.com/raphamorim/wasm-and-rust</a>

#### 4. Final Remarks

C/C++ offers the best overall performance but it does so at the price of potential memory bugs and data races that can be difficult/time-consuming to debug, especially in a concurrent multi-threading environment. A C/C++ programmer always needs to maintain at work a state of high concentration to prevent introducing bugs. On the other hand, Rust offers stability and performance with fewer bugs to start with. Learning Rust can often teach one to write better, safer C/C++ and acquire good programming habits. Overall the experience with Rust at JVO has been positive although the Rust compiler can be painful to work with initially. Especially when learning Rust it is easy to “hit a wall” and get stuck, keep searching for a suitable solution for several days. On a positive note, Rust offers excellent documentation and tutorials. New users are encouraged to consult the “Rust Book” at <https://doc.rust-lang.org/stable/book/> .