Ubiquity Generator (UG) Software Management Plan

This Software Management Plan was developed in the DFG funded project HPO-Navi (https://gepris.dfg.de/gepris/projekt/391087700). It was aligned along the Software Sustainability Institute recommendations for Software Management Plans (https://doi.org/10.5281/zenodo.2159713)

What software will you develop?

- This is the place where you can enter some information on the software you plan to develop. A sort abstract perhaps containing the name, if it is a new development or an update, the general purpose. If you are developing from scratch bear in mind that your inventive name might already be taken.

The name of the software is "Ubiquity Generator", in short UG. As far as we know it does not violate any existing trademarks held by others. There is UG (https://link.springer.com/article/10.1007/s007910050003), which is a flexible software toolbox for solving partial differential equations. The UG is "Unstructured Grid" and it does not create any confusion.

The software has been developed since 2010 by Yuji Shinano and is constantly being extended and improved. UG is a generic framework for parallelizing an external branch-and-bound based solver for optimization problems in a distributed or shared memory computing environment. It exploits powerful performance of state-of-the-art "base solvers", such as SCIP, CPLEX, etc. without the need for base solver parallelization.

The plan is to bring the software from a prototypical state to the release of a first major version in terms of organization and documentation of source code and documentation as well as the build and installation process.

Who are the intended users of your software?

- Here you can describe the intended target group of your software. Is it just a special kind of script for one user and one purpose unlikely to be reused by others, or will it serve a greater community? How are the expected skills for using the software in terms of compiling knowledge vs. starting an executable? What is the needed operating system, or other infrastructure like compilers or languages?

Initially, we expected to use it to parallelize branch-and-bound based solvers. However, it is extended to one that can be used any type of search based solvers and even others.

We want to integrate a cmake support so that compilation and installation from source code will be easy even to new users. It would help to have some knowledge about MPI (Message Passing Interface) and its programming style.

As source code can be potentially compiled on any system there are no limits. However the two main systems that are supported will be Linux and Mac.

How will you make your software available to your users?

- This section is about the various ways that you could choose to make your software publicly available. General ideas are along the lines of: Will you release your software at all? In which form (binaries, libraries packages, source code)? Where can someone get your software and are there any boundaries, like registration or paying a usage fee?

The software UG will be released as part of the SCIP Optimization Suite on scip.zib.de. We will release the source code since UG is intended to be an open source software. It is also included in some of the binary packages. The software is and will be available at ug.zib.de which links to scip.zib.de.

What licenses are attached to your software?

- Here you can and definitely should choose from the vast array of software licenses. If you do not choose a license then some generic intellectual property right will apply and nobody is allowed to do anythinmg with your software. Most common choices are MIT, APACHE, GPL.

LGPL

How will you support those who use your software?

- In this area you can explain your intended support or user contact like: How much effort will you have available to support your users, if any? And what level of support would that be? Are there established ways for user interaction like a ticket system?

In the long term there will be a mailing list for questions and support. The developers will receive and answer questions via the mailing list. The information about the mailing list will be published on the ug website and everybody on the mailinglist will be able to see the previously asked questions and answers. For the time being people are encouraged to write to the main developer directly.

How will your software contribute to research?

- This section is about the way that you think your software will help the overall effort of science. Possible questions include: Will it help your users, or yourself, to conduct research more easily? Produce results more rapidly? Produce results to a higher degree of accuracy or a finer level of detail? Exploit the power of super-computers? Will it help your users, or yourself, to further research by enabling research that cannot be done at present? What are the limitations of similar research software that already exists and how will your software be better?

It helps users to build a parallel solver, which can run on a large scale super-computer with minimal effort. It can parallelize an existing state-of-the-art sequential or multi-threaded solver relatively easily. Users can use a large scale super-computer easily with the latest algorithm implementation.

How will your software relate to other research objects?

- A general classification of your software will help other users assess if your software is right for them. Give them some hints in the direction of: Which published research objects relate to your research software? Are there published papers that describe the research that your software will enable? Will your software extend or depend upon other research software? Will you publish papers about the research you have done to which your software has contributed? Will you publish any data sets produced by your software?

UG extends SCIP, as an academically open research branch-and-bound solver. According to swMATH (an information service for mathematical software), UG is currently referenced in 23 research papers. There are public data sets that can be used by UG, some examples will be distributed with the source code.

How will you measure your software's contribution to research?

- This set of question deals with impact of your software and the ways you intend to measure it. If you have not thought about this aspect of scientific output, feel free to scip this area for now. Questions to ask yourself could be: What evidence do your funders and other stakeholders expect you to present to show that your software has contributed to research? Will you measure who has downloaded your software? Will you gather information on publications that describe research which your software has helped to enable, including those by other researchers? Do you intend to have a recommended reference or citation for your software?

The scientific impact is measured by mentions in research papers and contributions to public libraries of mathematical problems whose solution has been improved by UG. For general usage statistics we count the number of downloads from our website. We have a recommended reference or citation for UG which we encourage users to use.

Where will you deposit your software to guarantee its long-term availability?

- Most funders require the products of science to be FAIR (Fiandable, Accessible, Interoperable and Reusable). To this end ask yourself some of the following questions: Will you deposit releases of your software into a digital repository? What digital repositories are appropriate for you to use? Has your funder or publisher recommended or mandated a digital repository to use? Are the policies of the digital repository acceptable to you? Is the digital repository free or do you have to pay a fee? If there is a fee, is this a one-off payment and can you afford it? oes the digital repository give you a unique persistent digital identifier for your deposit?

We will publish the source code in the ZIB institutional repository OPUS. In the course of this project an open source software (https://git.zib.de/rkhorsan/gitram) was developed for harvesting gitlab information to automatically create and archive citable versions of software (and exemplary UG). In the ZIB we are in the position to have on the one hand OPUS for ongoing availability of published versions and on the other hand EWIG for long-term preservation of code or other digital objects.

Software assets used and produced

- You should state the environment your software needs to function. Possible guideline questions could be: What software will be used by your project? What software will be produced by your project? What are the dependencies / licenses for third party code, models, tools and libraries used?

Different base solvers will be interfaced by UG. Our project will produce a software framework that interfaces a base branch-and-bound solver that it will parallelize.

Intellectual Property and Governance

- This section is not only about you possibly using other people code but should give you some ideas how to make your code reusable by other people. Please bear in mind, that if there is no useful license is attached to your code, nobody would have the right to re-use it. Have you chosen an appropriate license for software developed by your project? Is your license clearly stated and acceptable to all partners? What are the licenses for third party code, models, tools and libraries used? Are there any issues that you are aware of to do with patents, copyright and other IP restrictions?

Up to version 0.9.1 of UG: The ZIB academic license (https://www.scipopt.org/academic.txt) guarantees freedom to share and change software for academic use. Commercial use is not permitted. The license is accepted by all known users.

From version 1.0.0 of UG: Gnu Lesser General Public License: The license allows developers and companies to use and integrate the UG framework into their own software. However, anyone who modifies UG is required to make it available under the same LGPL license.

Access, sharing, quality assurance and reuse

- This section addresses the different realms of research: The private domain, the collaborative domain and the permanent/public domain. E.g. have you identified suitable project infrastructure early, particularly a code repository (either in-house or public)? Will your project repository be public or private? Do you have a requirement for private storage? How you will manage releases (how often, how delivered, how will you decide when to release)? How will you ensure you deliver "correct" code (e.g. tests, frameworks, checklists, quality control)? How you will deliver readable code that can be understood by others (e.g. documentation, coding standards, code reviews, pair programming)? How will you make it easy to reference and cite the software produced by your project?

We have an in-house structure at ZIB: A gitlab repository server and a jenkins continuous integration server. We plan on integrating the unit-testing and performance testing of UG in jenkins, an automated continuous integration server. There will be merge request templates with checklists for code reviews and quality control. Code changes should be reviewed (for example in merge requests in gitlab). We will release UG and its source code as a part of the SCIP Optimization Suite, that is scheduled to be more or less regularly once a year. There will be a DOI issued for every version of the framework, this information and the versions of UG will be archived in the long-term preservation system of ZIB and therefore citeable and accessible.

The code is formatted and documented by the coding style that is defined in coding-style-guidelines.md in the UG repository.

Long-term preservation

- You should consider some requirements regarding the long term preservation of your code. E.g. where will you deposit software for long-term preservation or archival? Does your institutional repository allow deposit of software? Does your chosen repository have a clear preservation policy? How will you record specific and implicit dependencies (e.g. browsers, operating systems, SDKs) required by your software? Does your software require access to any public web services / infrastructure / databases that may change or disappear? Do you have a need to record and track versions of service interfaces and any use of open or proprietary standards that may change/become superseded by others?

The projects Opus and EWIG intend to harvest information from the gitlab repositories and websites to automate long-term preservation and archival of software. UG is supposed to be the exemplary project for this initiative. The OPUS repository can be used to deposit software. This functionality is developed within the research project HPO-NAVI. The interfaces to the base solvers may change in time so we will have to keep an eye on that.

Resourcing and responsibility

- When it comes to not only deposit your software after creation but to keep ist alive for an extended period of time you should consider some of the more management-like questions in this section. If you have developed just a specific kind of script for your project, you could probably skip this section. If you plan for longevity of your code, ask yourself some of the following questions. What software development model will you aim to use? How you will support your software (how much effort is available, what level of service will you offer, how will you interact)? Will this change over time? What effort is available to support the software (funded on your project, unfunded volunteers, temporary, students)? Whose responsibility is it for different roles (e.g. project manager, build manager, technical authority, change board, support requests)? How you will track who does and has done what (e.g. TODOs, issues, bugs and queries)? How do you ensure adequate knowledge exchange within the team to ensure that knowledge is not lost when people leave (e.g. documentation, pair programming, reviews)? How often will you review and revise the software management plan?

At the moment there is only one main developer for UG. For the test management and release the test manager of the department is responsible. Our source code is managed in gitlab hosted at zib, which also provides us with an issue-tracker and git versioning. We want to implement a practice of documenting the source code and reviewing changes as well as create a guide for new developers and new users. It is intended to review and revise this software management plan once a year.

Source code

Documentation

- General documentation related issues are: What type of documentation is available for the software? Is the purpose of the software stated in the documentation? Does the documentation describe how to a) test b) use c) build?

There will be a documentation generated by doxygen https://ug.zib.de/doc-1.0.0/html/ and also a website https://ug.zib.de. In this documentation the process of building the software is explained.

Testing

- You could drive the quality assurance even further by including a test-scheme. Guiding questions are: Do you test your software? What type of testing are you using? Do you use any testing methodology? Are the tests for the software automated? Are the tests available with the source code?

There will be a test suite that is coordinated by Scripts and Makefiles, also there will be some tests available via Ctest. We also have Continuous Integration and nightly testing on a Jenkins server at ZIB. These tests run automatically but can also be triggered by hand. The test infrastructure is included in the source code package.

Reproducibility

- This section is really optional in terms of software management plans which help your development work. It is more in the area of reproducible research and making it easier for other people to understand your way of producing this particular piece of software. Guiding questions in this direction could be: Do you use a version control system? What version control system do you use? Do you assign a version to each release of your software? Do you use semantic versioning? How to you define dependencies of your software and their version? Do you provide input and output examples?

As a source code management system we will use git with the gitlab instance hosted at zib. To every release there will be a version number. We use semantic versioning, as in given a version number MAJOR.MINOR.PATCH, we increment the

- MAJOR version when we make incompatible API changes,
- MINOR version when we add functionality in a backwards compatible manner, and
- PATCH version when we make backwards compatible bug fixes.

The dependencies UG is relying on are supposed to be mentioned in the documentation and implicitly listed by the CMake Build System. For testing purposes the documentation mentions some input and expected output examples.