Improving software quality in bioinformatics through teamwork

Katalin Ferenc[1], Ieva Rauluseviciute[1], Ladislav Hovan[1], Vipin Kumar[1], Mariekie Kuijjer[1], and Anthony Mathelier[1,2,✉]

[1] Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway
[2] Department of Medical Genetics, Institute of Clinical Medicine, University of Oslo and Oslo University Hospital, Oslo, Norway

✉ Correspondence: Anthony Mathelier <anthony.mathelier@ncmm.uio.no>

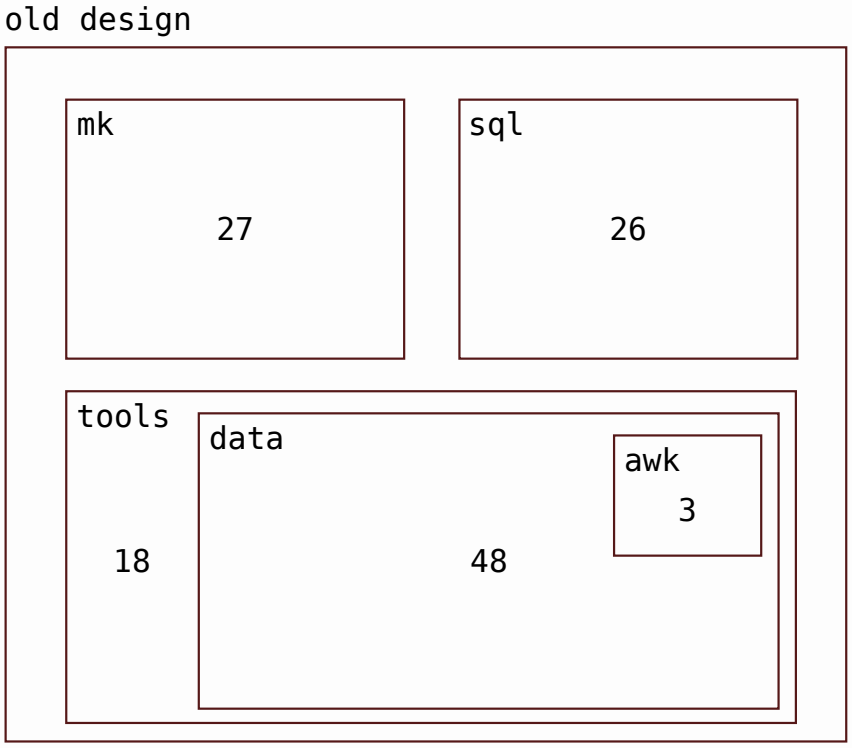
SUPPLEMENTARY FILE

SUPPLEMENTARY METHODS

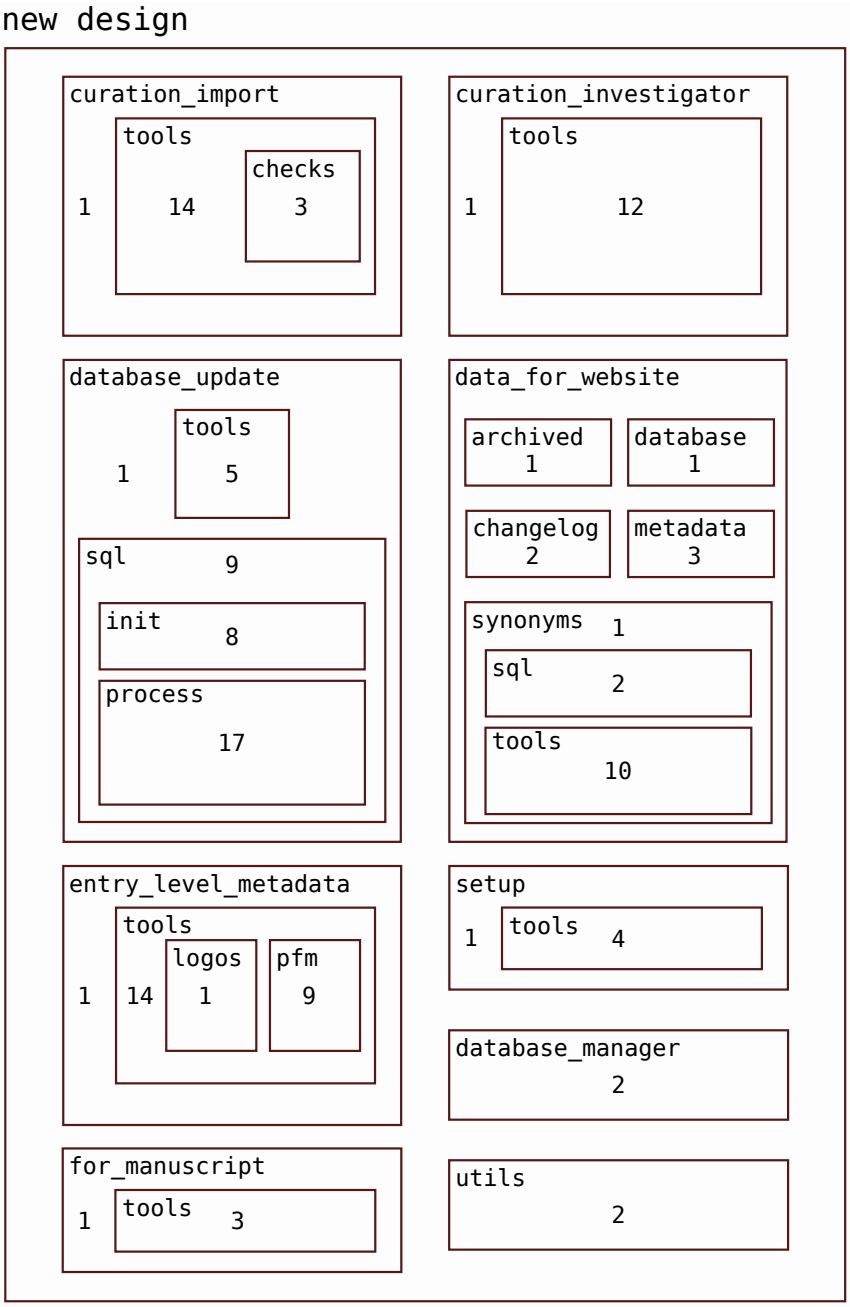
Following the standard methods of literature review, here we list the phrases and platforms of search. The literature search was performed in multiple iterations using Google (to include grey literature), PubMed and Google Scholar based on phrases “guidelines for bioinformatics software”, “rules for biologists learning bioinformatics”, “scientific software development”, “software engineering bioinformatics” and “bioinformatics software recommendations” throughout 2023. Additionally, relevant articles were selected based on the snowball effect from the references of the initial publications.

SUPPLEMENTARY FIGURES

Modularization



Supplementary Figure 3: **Improving the modularization of a large codebase: previous.** In the previous design the files were arranged by on their type. The numbers denote the number of files in each directory represented by the rectangle. mk: makefile



Supplementary Figure 4: **Improving the modularization of a large codebase: current.** In the current design the files are arranged by their function. The numbers denote the number of files in each directory represented by the rectangle. The number of files are different due to added features and changes beyond the organization. pfm: position frequency matrix

Testing

This is an example of testing, represented by a subset of test used by the SPONGE package. The unit tests check the correctness of individual functions. Some of the tests shown test the plogp function, which calculates the value of p * log2(p) while treating the zero case correctly. Also tested is the calculation of the information content for individual motifs in the calculate_ic function. The integration tests check that the entire workflow produces the expected output, effectively checking that the components work well together. In this case, the full functionality of SPONGE with the default parameters is checked.

Selected content of `tests/test_sponge.py` is shown below.

```
import pytest

### Unit tests ###

# Helper functions
import sponge.helper_functions as helper_f

# parametrize allows testing multiple inputs without code
duplication
@pytest.mark.parametrize("input, expected_output", [
    (0, 0),
    (0.5, -0.5),
    (1, 0),
])
def test_plogp(input, expected_output):
    assert helper_f.plogp(input) == expected_output
```

```
import pytest
from sponge.test_motifs import *

def test_calculate_ic_no_info(no_info_motif):
    assert helper_f.calculate_ic(no_info_motif) == 0



def test_calculate_ic_all_the_same(all_A_motif):
    # Length of the test motif is 6, so expected value is 2 * 6 =
```

```
12
    assert helper_f.calculate_ic(all_A_motif) == 12


def test_calculate_ic_SOX2(SOX2_motif):
    assert (helper_f.calculate_ic(SOX2_motif) ==
        pytest.approx(12.95, abs=0.01))

### Integration tests ###
import os
import pytest

from sponge.sponge import Sponge

# The test is marked as slow because the download of the bigbed
file takes
# a lot of time and the filtering is also time consuming unless
parallelised
@pytest.mark.slow
def test_full_default_workflow(tmp_path):
    # Tests the full SPONGE workflow with default values

    ppi_output = os.path.join(tmp_path, 'ppi_prior.tsv')
    motif_output = os.path.join(tmp_path, 'motif_prior.tsv')

    sponge_obj = Sponge(
        run_default=True,
        temp_folder=tmp_path,
        ppi_outfile=ppi_output,
        motif_outfile=motif_output,
    )

    assert os.path.exists(ppi_output)
    assert os.path.exists(motif_output)
```

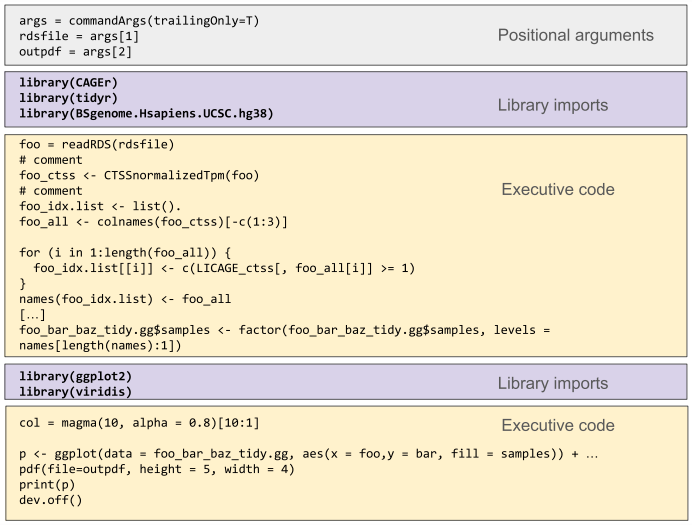
The motifs used by the tests are defined in a separate file and accessible as pytest fixtures.

```
import pytest
import pandas as pd
from Bio.motifs.jaspar import Motif
```

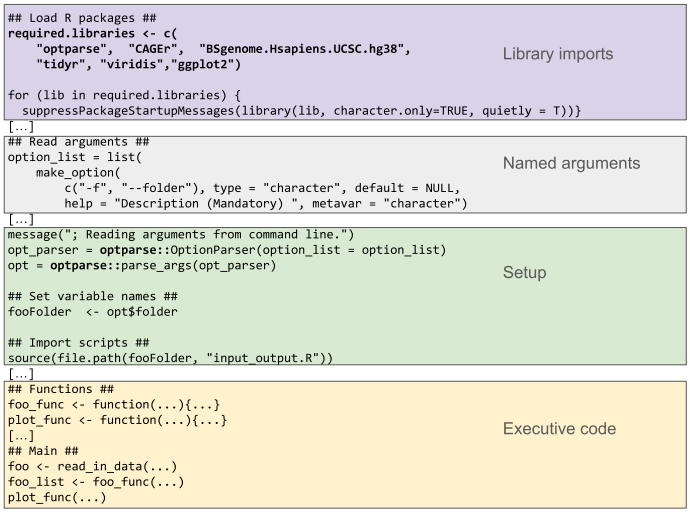
```
from pyjaspar import jaspardb

# A motif without any information
@pytest.fixture
def no_info_motif():
    no_info_row = [0.25] * 4
    no_info_counts = [no_info_row] * 6
    no_info_pwm = pd.DataFrame(no_info_counts, columns=['A', 'C',
'G', 'T'])
    no_info_motif = Motif(matrix_id='XXX', name='XXX',
counts=no_info_pwm)

    yield no_info_motif


# A motif with perfect information
@pytest.fixture
def all_A_motif():
    all_A_row = [1] + [0] * 3
    all_A_counts = [all_A_row] * 6
    all_A_pwm = pd.DataFrame(all_A_counts, columns=['A', 'C',
'G', 'T'])
    all_A_motif = Motif(matrix_id='XXX', name='XXX',
counts=all_A_pwm)

    yield all_A_motif


# A real motif for SOX2
@pytest.fixture
def SOX2_motif():
    jdb_obj = jaspardb(release='JASPAR2024')
    SOX2_motif = jdb_obj.fetch_motif_by_id('MA0143.1')

    yield SOX2_motif
```

Dependency management

There are two angles of dependency management we give example to here. First, we share a previous and current version of a code where the placing of the package imports are improved. This code also can be seen as an example for modularization with the rearrangement of the linear script to setup and functions. Furthermore, we improved the documentation and usability with using named arguments instead of positional ones.



Supplementary Figure 5: **An example for dependency management within the code: before.**



Supplementary Figure 6: **An example for dependency management within the code: after.**

Second, we share an example of documenting the requirements where the responsibility of installing the software is moved from the user to the developer. README-based solution: the user is required to install the dependencies, version and source might be given but compatibility following updates is not ensured.

```
## Installation


- R (version >= 3.6.1)
- CAGEr (version >= 2.6.1) (for installation follow the
instructions here [https://bioconductor.org/packages/release/bioc/
vignettes/CAGEr/inst/doc/CAGEexp.html#normalization])
- BSgenome.Hsapiens.UCSC.hg38
- tidyr
- viridis
- ggplot2
```

Container-based solution: the user can either use the publicly available container that includes a snapshot of all necessary requirements, or build their own environment.

```
## Installation


Container available at https://hub.docker.com/r/cbgr/cager261
For details, refer to requirements.R
```

The content of `requirements.R` is shown below.

```
## Container folder structure
.libPaths( c( "/opt/software" , .libPaths() ) )
## CRAN packages:
packages_cran = c(
  "optparse",         ## Read in data
  "tidyr",                ## Data formatting and manipulation
  "ggplot2",           ## Plotting
  [...])
message(
"; Installing these R packages from CRAN repository:",
packages_cran)
install.packages(
packages_cran, repos="https://cran.uib.no/", lib="/opt/software")
```

```
## Bioconductor packages:
packages_bioconductor <- c(
"BSgenome.Hsapiens.UCSC.hg38")

message(
"; Installing these R Bioconductor packages: ",
packages_bioconductor)
BiocManager::install(packages_bioconductor, lib="/opt/software")
```

The content of the `Dockerfile` is shown below.

```
# Docker install R 4.3, Bioconductor 3.17
FROM bioconductor/bioconductor_docker:3.17

# Set up folder structure
WORKDIR /opt/software

# Install CAGEr 2.6.1
RUN R -e 'BiocManager::install("CAGEr")'

# Install other R dependencies
COPY requirements.R /opt/software/requirements.R
RUN Rscript requirements.R
ENV R_LIBS=${R_LIBS}:/opt/software
```

SUPPLEMENTARY TABLES

Supplementary Table 1: **Software quality attributes and their description** TODO

Supplementary Table 2: **Examples of software quality meeting topics** This table contains examples of the topics of past software quality meetings. It has been organise to follow the same categories as **Table 1**.

Category	Title	Description
Software development 101	To be identified	To be filled

Category	Title	Description
Advanced software development	Design patterns	To be filled
Software development process	Code review	To be filled
Testing and validation	Why testing?	To be filled
Reproducibility	Dependency management	To be filled
Documentation	On Pages and Reports	To be filled
Community effort	To be identified / we never covered it probably	To be filled