Tools for correctness, part 1

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling (clang, make, …)
  - Architectures, portability (ABIs, …)
  - Code management (regex, git)
- Part 3: Correctness
  - Specifications
  - Documentation, testing ⟵ TODAY
  - Static & dynamic analysis, debugging
- Part 4: Performance
  - CPU pipelines, caches
  - Data structures
  - Parallel computation

Documentation

Documentation is GOOD

- Allows others to understand your code
- Allows you (in a few weeks) to understand your own code
- Helps make your thought process and assumptions explicit

Types of documentation

- Reference manuals
  - Complete, authoritative source of information

    (if the code does not do what the manual says, then the code is wrong)
  - Must use precise language (even at the cost of legibility)
  - Examples: “`man`” pages, ABI docs, C standard, IEEE-754 specifications
- Tutorials
  - Beginner-friendly. Emphasize getting things to work quickly

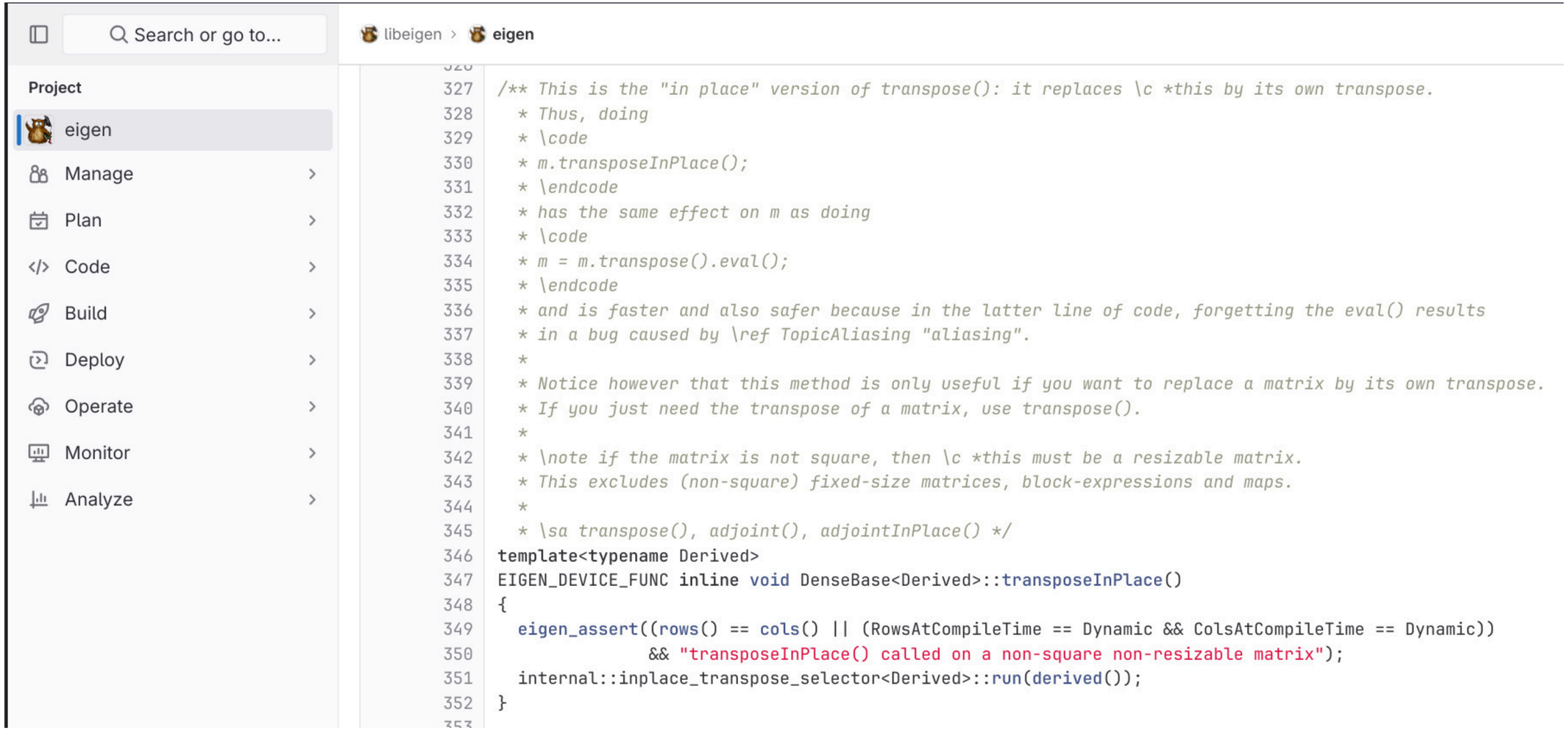
    (even at the cost of completeness)
  - Examples: various books (K&R C, Think Python) and intro material
- Questions and answers (Q&A)
  - Not exhaustive
  - Quick answers to frequently asked questions
  - Examples: Stack Overflow

Automated documentation

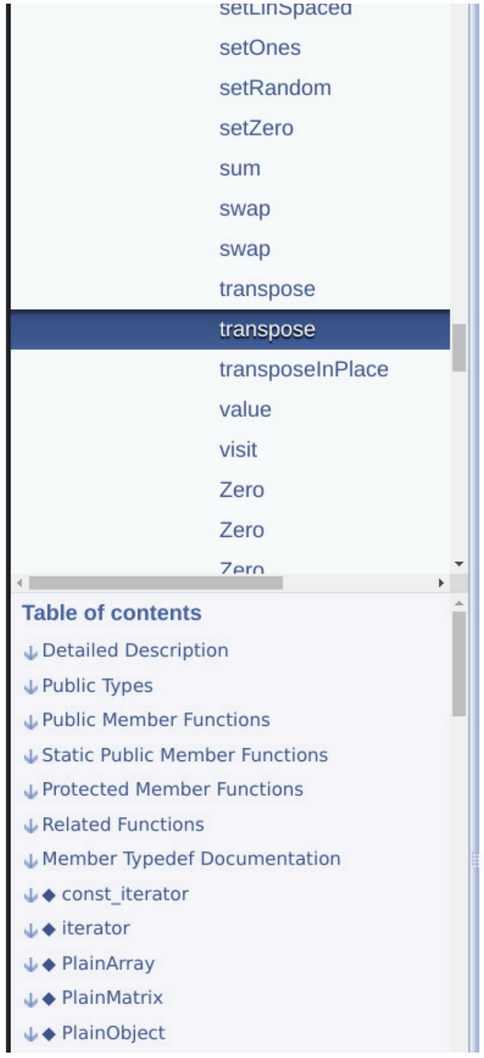
Automated documentation systems

- read and parse source code
- find functions (methods, classes, …)
- create a (PDF or webpage) document containing function signatures
- specially-formatted comments in the source code are copied into the documentation along with the corresponding function signatures

Doxygen




```
/** This is the "in place" version of transpose(): it replaces \c *this by its own transpose.
  * Thus, doing
  * \code
  * m.transposeInPlace();
  * \endcode
  * has the same effect on m as doing
  * \code
  * m = m.transpose().eval();
  * \endcode
  * and is faster and also safer because in the latter line of code, forgetting the eval() results
  * in a bug caused by \ref TopicAliasing "aliasing".
  *
  * Notice however that this method is only useful if you want to replace a matrix by its own transpose.
  * If you just need the transpose of a matrix, use transpose().
  *
  * \note if the matrix is not square, then \c *this must be a resizable matrix.
  * This excludes (non-square) fixed-size matrices, block-expressions and maps.
  *
  * \sa transpose(), adjoint(), adjointInPlace() */
template<typename Derived>
EIGEN_DEVICE_FUNC inline void DenseBase<Derived>::transposeInPlace()
{
  eigen_assert((rows() == cols() || (RowsAtCompileTime == Dynamic && ColsAtCompileTime == Dynamic))
               && "transposeInPlace() called on a non-square non-resizable matrix");
  internal::inplace_transpose_selector<Derived>::run(derived());
}
```



◆ transposeInPlace()

template<typename Derived >

void **Eigen::DenseBase**< Derived >::transposeInPlace

This is the "in place" version of **transpose()**: it replaces `*this` by its own transpose. Thus, doing

```
m.transposeInPlace();
```

has the same effect on m as doing

```
m = m.transpose().eval();
```

and is faster and also safer because in the latter line of code, forgetting the **eval()** results in a bug caused by **aliasing**.

Notice however that this method is only useful if you want to replace a matrix by its own transpose. If you just need the transpose of a matrix, use **transpose()**.

Note

if the matrix is not square, then `*this` must be a resizable matrix. This excludes (non-square) fixed-size matrices, block-expressions and maps.

See also

transpose(), adjoint(), adjointInPlace()

Python docstrings

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Automated documentation systems

- General:
  - doxygen
  - sphinx
- Python-specific:
  - pdoc
  - PyDoc
  - pydoctor

Assertions

- Assertions are used to document (and check) assumptions made in the code.

- An assertion failure
  - should correspond to a bug in your code,
  - in Python, raises AssertionError exception
  - in C, triggers an immediate crash (`abort()`) of your program.

```python
def gcd(a, b):
    if a < b:
        a, b = b, a

    while b != 0:
        assert a >= b       # <---- this should always be true
        a, b = b, a % b

    return a
```

```c
#include <assert.h>

int gcd(int a, int b)
{
    if (a < b) {
        int r = a;
        a = b;
        b = r;
    }

    while (b != 0) {
        assert(a >= b);      // <---- this should always be true

        int r = a % b;
        a = b;
        b = r;
    }

    return a;
}
```

Disabling assertions

In Python:

```
python -O script.py
```

In C:

```
clang -D NDEBUG -Wall -O3 -o main main.c
```

(equivalent to

```
#define NDEBUG
```

at the beginning of every file)

Error vs assertion failure

- an error happens when, for external reasons, your program cannot run
  - examples: out of memory, file cannot be read, network unreachable

- an assertion fails if a fundamental assumption in your code is violated
  - indicates a bug in your code

Testing

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a | b) == 1;
}
```

```c
void run_tests_0()
{
    assert(either_nonzero(5, 5) != 0);
    assert(either_nonzero(0, 5) != 0);
    printf("OK\n");
}
```

Test coverage

- line coverage:

  is every line of code covered by some test case?

- branch coverage:

  for every conditional branch, is there a test covering each of the two possibilities (taking the branch or not taking it)?

- path coverage:

  is there a test covering all possible execution paths?

```
gcc -Wall -O3 --coverage -c -o either_nonzero.o either_nonzero.c
gcc -Wall -O3 --coverage -o run main.c either_nonzero.o
```

```
./run_tests
```

```
OK
```

```
gcov either_nonzero.c
```

```
File 'either_nonzero.c'
Lines executed:100.00% of 4
Creating 'either_nonzero.c.gcov'

Lines executed:100.00% of 4
```

```
gcov -b either_nonzero.c
```

```
File 'either_nonzero.c'
Lines executed:100.00% of 4
Branches executed:100.00% of 4
Taken at least once:75.00% of 4
No calls
Creating 'either_nonzero.c.gcov'

Lines executed:100.00% of 4
```

```
function either_nonzero called 2 returned 100% blocks executed 100%
        2:    4:int either_nonzero(int a, int b)
        -:    5:{
        2:    6:    if (a != 0)
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
        -:    7:        a = 1;
        -:    8:
        2:    9:    if (b != 0)
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
        -:   10:        b = 1;
        -:   11:
        2:   12:    return (a | b) == 1;
        -:   13:}
```

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a | b) == 1;
}
```

```c
void run_tests_0()
{
    assert(either_nonzero(5, 5) != 0);
    assert(either_nonzero(0, 5) != 0);
    printf("OK\n");
}
```

Line coverage vs. branch coverage

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a | b) == 1;
}
```

```c
void run_tests_x()
{
    assert(either_nonzero(5, 5) != 0);
    printf("OK\n");
}
```

Line coverage: 100%

Branch coverage: 50%

Branch coverage vs. path coverage

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a | b) == 1;
}
```

```c
void run_tests_y()
{
    assert(either_nonzero(0, 0) == 0);
    assert(either_nonzero(0, 5) != 0);
    assert(either_nonzero(5, 0) != 0);
    printf("OK\n");
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 75%

How does it work?

```
gcc -Wall -O3 --coverage -c -o either_nonzero.o either_nonzero.c
```

```
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero(int a, int b)
{
    line_covered(6);
    if (a != 0) {                // line 6
        branch_covered(6, 1);
        line_covered(7);
        a = 1;                   // line 7
    } else {
        branch_covered(6, 0);
    }

    line_covered(9);
    if (b != 0) {                // line 9
        branch_covered(9, 1);
        line_covered(10);
        b = 1;                   // line 10
    } else {
        branch_covered(9, 0);
    }

    line_covered(12);
    return (a | b) == 1;         // line 12
}
```

Limitations of test coverage measures (1)

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero_WRONG_1(int a, int b)
{
    if (a != 0)
        a = 1;

    if (b != 0)
        b = 1;

    return (a + b) == 1;
}
```

```c
void run_tests_1()
{
    assert(either_nonzero_WRONG_1(0, 0) == 0);
    assert(either_nonzero_WRONG_1(0, 5) != 0);
    assert(either_nonzero_WRONG_1(5, 0) != 0);
    //assert(either_nonzero_WRONG_1(5, 5) != 0); // <-- this one fails
    printf("OK\n");
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 75%

Limitations of test coverage measures (2)

```c
/*
  This functions returns:
    0        if both of its arguments are zero
    nonzero  if one or both of its arguments are nonzero
*/
int either_nonzero_WRONG_2(int a, int b)
{
    return a + b;
}
```

```c
void run_tests_2()
{
    assert(either_nonzero_WRONG_2(0, 0) == 0);
    assert(either_nonzero_WRONG_2(0, 5) != 0);
    assert(either_nonzero_WRONG_2(5, 0) != 0);
    assert(either_nonzero_WRONG_2(5, 5) != 0);
    //assert(either_nonzero_WRONG_2(5, -5) != 0); // <-- this one fails
    printf("OK\n");
}
```

Line coverage: 100%

Branch coverage: 100%

Path coverage: 100%

Fuzzing

We need good tests

Assertions and tests are useful

but only if we have good test cases

and enough of them

$\Rightarrow$ How do we generate good tests?

On a basic level, a fuzzer proceeds as follows:

1. take a (mostly valid) example input file

2. run the tested program with that input file

3. check for crashes (e.g. segmentation fault, assertion failures)

4. modify the input file:
    - random modifications
    - truncations, duplications
    - mergers with other example input files

5. go back to 2

Advanced fuzzers

- use test coverage techniques

  to determine which input files are “interesting”,

  in that they cover previously-uncovered source code

- use static analysis techniques

  to determine input file modifications that could trigger specific code branches

AFL++

- open source project (https://aflplus.plus/)
- takes as an input a directory with many (mostly valid) example input files
- generates modified input files that (try to) yield crashes

```
afl-fuzz -i directory/with/example/inputs/ -o directory/for/crash/files/ -- ./executable @@
```