From `printf` to a Proper Bug Finding Strategy

Debugging and Testing Basics

Mladen Ivkovic
Durham University
November 2023

The Joy Of Finding Bugs

```
 $ make
gcc main.c -o ex -g -Wall -Werror -pedantic -std=c11
19:56 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/coding/d
 $ ./ex
Segmentation fault (core dumped)
19:56 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/coding/d
 $
```

How to Approach Bugs

1) Don't panic.
2) Take a systematic approach. Find out:
    1) What is happening?
    2) Why is it happening?
    3) Where is it happening?

    This is what I want to talk about today

3) Design a unit test that reproduces the bug
4) Fix the bug
5) Document the fix (at least a comment in the code) – your future self and collaborators will thank you!

printf – The Ol' Reliable

```c
#include <stdio.h>

int main(void) {


  initialise();
  do_stuff();
  do_other_stuff();
  do_more_stuff();
  finish();

  return 0;
}
```

```
 $ make
gcc main.c -o ex -g -Wall -Werror -pedantic -std=c11
19:56 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/coding/d
 $ ./ex
Segmentation fault (core dumped)
19:56 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/coding/d
 $ 
```

printf – The Ol' Reliable

Bisect the program – find out how far it gets

```c
#include <stdio.h>

int main(void) {

  printf("Check 1\n");
  fflush(stdout);
  initialise();
  printf("Check 2\n");
  fflush(stdout);
  do_stuff();
  printf("Check 3\n");
  fflush(stdout);
  do_other_stuff();
  printf("Check 4\n");
  fflush(stdout);
  do_more_stuff();
  printf("Check 5\n");
  fflush(stdout);
  finish();

  return 0;
}
```

```
20:22 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/codi
 $ make
gcc main.c -o ex -g -Wall -Werror -pedantic -std=c11
20:22 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/codi
 $ ./ex
Check 1
Check 2
Check 3
Segmentation fault (core dumped)
```

WARNING: Don't forget to flush the `stdout` – the buffering may not display all the output!

Alternative: Set the `stdbuf` to unbuffered:

```
$ stdbuf -o0 ./your_program
```

`printf` – The Ol' Reliable

Rinse and repeat, until you find where the problem is

```
void do_other_stuff(void){

  do_first_other_stuff();
  do_second_other_stuff();
  do_third_other_stuff();
}
```

```
void do_other_stuff(void){

  printf("In do_other_stuff: Check 1\n");
  fflush(stdout);
  do_first_other_stuff();
  printf("In do_other_stuff: Check 2\n");
  fflush(stdout);
  do_second_other_stuff();
  printf("In do_other_stuff: Check 3\n");
  fflush(stdout);
  do_third_other_stuff();
}
```

```
20:22 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/codin
 $ make
gcc main.c -o ex -g -Wall -Werror -pedantic -std=c11
20:28 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/codin
 $ ./ex
Check 1
Check 2
Check 3
In do_other_stuff: Check 1
In do_other_stuff: Check 2
Segmentation fault (core dumped)
20:28 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/codin
 $
```

`printf` – The Ol' Reliable

- Bisection with printouts is good and well – but not very efficient.
  - Bisection usually means running your code over and over again until you zero in on where the program finds an error.
  - Meanwhile, the *cause* of the error may be somewhere else entirely.
  - Typically, you'd remove your checks again after you're done. So if a similar issue arises, you need to start from scratch.

Debugging Strategy

- Design your project with debugging in mind
  - Keep things clean and documented
  - Have an option to make your code talkative
  - Add tests, validations, assertions inside and outside of your code
    - Outside: Regular testing and verification.
    - Inside: On-the-fly validation, defensive programming
- Make your life easier with external tools

Let's look at these now

Talkative Code

- Supercharge The Ol’ Reliable `printf`
- Typically implemented as a runtime option determining how verbose you want your code to be.
- You can set several levels of verbosity!

Example

```c
enum verbosity_level {
  verbosity_level_quiet = 0,
  verbosity_level_verbose = 1,
  verbosity_level_trace = 2,
  verbosity_level_debug = 3
}


/**
 * Print out a message if the code is run with a high
 * enough verbosity level.
 * @param message Message to print out
 * @param level   threshold level of message to print.
 * @param verbose verbosity of the run.
 */
void log_message(char* message,
                 verbosity_level level,
                 int verbose){

  if (level <= verbose){
    printf("%s\n", message);
  }
}
```

```c
int main(void) {

  // Can be read in e.g. through cmdline arg
  // or parameter file.
  int verbose = 0;

  log_message("Starting up",
              verbosity_level_quiet, verbose);
  initialise();

  log_message("calling do_stuff",
              verbosity_level_debug, verbose);
  do_stuff();

  log_message("calling do_other_stuff",
              verbosity_level_debug, verbose);
  do_other_stuff();

  log_message("calling do_more_stuff",
              verbosity_level_debug, verbose);
  do_more_stuff();

  log_message("calling finish",
              verbosity_level_debug, verbose);
  finish();

  log_message("Done. Bye",
              verbosity_level_quiet, verbose);

  return 0;
}
```

Supercharging the Logs

- You might want to consider adding:
  - Default flushing to `stdout` if verbosity level is intended for debugging
  - Writing different verbosity level outputs to different log files
  - Applying additional filters to your logs
  - All sorts of bells and whistles, e.g.
    - Which file, function, line number is this message coming from?
    - What is the elapsed time since startup?
    - What thread/rank is this message from?

External Testing

- We differentiate between
  - Unit Testing
  - Functional Testing
  - Regression Testing

- Ideally, you automate these tests.
  - e.g. every time you update your main git branch, you run a full test suite. (→ continuous integration)

Unit Testing

- Tests for small units of your code
  - e.g. individual functions, objects, ...
- Test that it does what it’s supposed to, *and nothing else*
- Test corner cases, limits, how it deals with wrong data, input etc.

Unit Testing: How-To

- Split your code into logical units, and test that the units behave exactly how you want them to.

- Ideally, you have a unit test for each function you write.

- In practice, you ***need*** to have unit tests for your core routines.

Unit Testing – What to test?

- Main use case
  - Does the code do what I want it to do?
- Every corner case you *can* think of
  - Correct values, wrong values, unexpected values, ...
- Every corner case you *can't* think of
  - Try your luck here! Use randomly generated setups, values, ...
  - Advice: If you are using random numbers:
    - set a random seed to make your run reproducible
    - make your program write out the used random seed so you can reproduce the test.

Unit Testing - Example

```
/* Sorts a list of `nelems` floats stored in `array` in numerically
 * non-decreasing order. */
void float_sort(float array[], size_t nelems);
```

- What if `nelems == 0`?
- What if array has only one element?
- What if array has multiple identical elements?
- What if array is already sorted? Or reverse sorted?
- Does it work with an even number of elements? Odd number of elements?
- What if array contains `inf, -inf, NaN, FLT_MIN, FLT_MAX` ?

Functional Testing

- Similar to unit testing in the sense that it tests (parts of) the code
- (Parts of) code not restricted to individual units any more.
- Typically functional testing isn't concerned with the source code, but with *some external specification*.
  - E.g. "*this program must be able to read in data written in format X and create a plot.*"
- In short: Test specific functionalities of (parts of) your code.

Regression Testing

- Test the actual result/output of your program for correctness.

- In a nutshell: test that known correct results are still correct.

- It’s vital to have a suite of tests, experiments, benchmarks etc. to validate current state of your code.

Something Broke After I Changed It

- Strategy: Look into what you changed, and how that could have caused a problem.

- Advice: Make use of git (or any other version control)
  - If you aren’t using git already, ***you really really should be***

Internal Testing

- Tests performed on-the-fly while your program runs
- Some of the trickiest bugs to uncover are due to faulty assumptions.
  - Example:
    "*My code has been running correctly so far*."
  - Your code may end up in a faulty state without you noticing or it being detected.
    - → try to detect it!
    - Explicitly verify current state of the running program satisfies assumptions and restrictions.

Internal Testing

- Two branches of internal tests:
  - Validate the current state of your program
    - Is everything as it should be?
    - Did an unexpected situation occur?
  - Defensive programming
    - Anticipate what state could break things, and guard against them, or raise an error notifying yourself what is wrong

Validation Example

```c
#include <stdio.h>

int main(void) {


  initialise();
  do_stuff();
  do_other_stuff();
  do_more_stuff();
  finish();

  return 0;
}
```

```c
#include <stdio.h>

#define DEBUG_LEVEL 2

int main(void) {

  initialise();
#if DEBUG_LEVEL > 1
  check_initialisation_corrent();
#endif

  do_stuff();
#if DEBUG_LEVEL > 1
  check_state_after_do_stuff();
#endif

  // etc
  do_other_stuff();
  do_more_stuff();
  finish();

  return 0;
}
```

Validation Example

- Very often you write your validation tests while first developing a new feature.
  - ***Keep them!*** Don't throw them away.
  - If you're worried about performance, you can hide them behind a compile time macro such as
    ```
    #if DEBUG_LEVEL > 1
    ```
    or
    ```
    #ifdef DEBUG_CHECKS
    ```
- They might automatically uncover bugs for you.
- If they don't, they will provide you with a palette of checks telling you what the problem ***isn't.*** That is valuable information.

Defensive Programming

- Anticipate what state could break things, and then explicitly check that you aren’t in that state.

Defensive Programming - Example

```c
/**
 * @brief Returns the sound speed given density and pressure
 *
 * @param density The density \f$\rho\f$
 * @param P The pressure \f$P\f$
 * @param hydro_gamma Adiabatic index.
 */
float gas_soundspeed_from_pressure(float density,
                                   float P,
                                   float hydro_gamma) {

  return sqrtf(hydro_gamma * P / density);
}
```

$$c_s = \sqrt{\frac{\gamma P}{\rho}}$$

- What can go wrong?
  - `density == 0` → division by zero
  - Any of the variables < 0 → undefined operation
  - Any of the variables is `inf` or `NaN`

Summary So Far

- `printf` bisection does the trick, but isn't the best way of doing things.
- Design your code with debugging in mind:
  - Make it talkative, enable logging relevant information
  - Add tests and validations
    - External: Unit tests, functional tests, regression tests
    - Internal: Validation tests, defensive programming

External Debugging Tools

- Even with a good defence, bugs happen.
- Some tools to deal with them:
  - `gdb`
    - tracebacks, live inspection
  - `Valgrind, address sanitizer`
    - memory leaks, memory issues

- Live demo scripts (+ hands on exercise)
  - https://github.com/mladenivkovic/debugging-essentials-demo

gdb

- Compile your program with debug symbols `-g`, or `-ggdb`
  - `$ gdb /path/to/executable`
  - `$ gdb --args path/to/executable --arg1 --arg2`
- Useful commands:
  - `b (break)` set a break point
  - `l (list)` show source code in CLI
  - `p (print)` print variables
    - For pointers: `p *pointer` works too!
  - `bt (backtrace)` show stack trace
  - `c (continue)` continue run
  - `n (next)` execute next line
  - `frame` show current location in trace
  - `frame <frame nr>` change into <frame nr>

gdb **breakpoints**

- Set breakpoint: (***before*** you execute `run`)
  - `break <line number in main file>`
    - e.g. `break 12` to break on line 12
  - `break path/to/file.c:<line number>`
    - e.g. `break includes/my_includes.h:13`
  - `break file.c:function_name`
    - To break when a function is called

gdb **breakpoints**

- Navigation:
  - `l (list)` show source code
    - `l <line number>` show source code starting at line number
  - `bt (backtrace)` show stack trace
  - `c (continue)` continue run
  - `where` show current location in trace
  - `frame` show current location in trace
  - `frame <frame nr>` change into <frame nr>
- `info break` : show information on currently set breakpoints
- `del 1` : delete breakpoint 1.
  - You can find breakpoint numbers using `info break`

gdb **watchpoints**

`watch <variable>` : halt program execution when value of variable is changed.

- You can set watchpoints at startup (before executing `run`) *if the variable is in global scope*.
- Otherwise, you need to first set a breakpoint at a point where the variable is in scope.
- Example:

```
// you can set watchpoint at
// `sum` before `run`

int sum;
int main(void){
  // do stuff
}
```

```
// you need to set a breakpoint
// inside main() and then set a
// watchpoint
int main(void){
  int sum;
  // do stuff
}
```

gdb **reverse**

- It is possible to run a program in reverse with gdb, but the functionality is very restricted.
  - Many optimizations/vectorizations break it. So make sure to compile and run without any optimizations enabled.
  - Chances are it might still not work with modern instructions.
  - Recording significantly slows down execution time. May require a lot of memory.

- How-To:
  1) Set a breakpoint from which to record the program execution.
  2) Run the `record` command when this breakpoint is hit.
  3) Set a different breakpoint at the point you wish to examine.
  4) Once you hit the breakpoint, you can go back through the code using
     - `rs (reverse-step)` : go in reverse line by line
     - `rc (reverse-continue)` : go in reverse until last breakpoint

gdb + segfaults and other errors

- One of the most useful features of gdb is to catch signals which would usually kill your program and halt it before it aborts.
- This permits you to explore the current state of your program just before your crash.
- This doesn’t need any special activation. Just run your code with gdb!
  - But don’t forget to compile it with debug symbols (-g or -ggdb) to get meaningful traces

gdb + core dumps

- It's possible to enable core dumps:
  - when program encounters error, write down what is currently in memory instead of just quitting
  - Enabling core dumps on linux:
    - `$ ulimit -S -c unlimited`
    - Note: your sysadmins may have disabled this.
    - Default core dump location may vary.
      - On HPC systems, it's often set to `$workdir`.
      - On Ubuntu 21+, `/var/lib/apport/coredump/`
- gdb can read that back in and allow you to debug!
- `$ gdb -c core.XXXX path/to/executable`

gdb – **“value has been optimized out”**

- „value has been optimized out” workaround
  - Compile entire program without optimization
  - … or tell compiler not to optimize specific function you’re looking at:

GCC:

```
#pragma GCC push_options
#pragma GCC optimize ("O0")

  void your_function(){...}

#pragma GCC pop_options
```

Intel:

```
#pragma optimize( "", off )

  void your_function() {...}

#pragma optimize( "", on )

// or

#pragma intel optimization_level 0
    void your_funtction(){...}
```

Clang:

```
__attribute__((optnone))
    void your_function()
    {...}
```

FPEs – Floating Point Exceptions

- There are ways of telling the compiler to raise an error if an invalid arithmetic operation (FPE) has occurred
- This is unfortunately not standardised – it depends on the compiler and the hardware.
- FPE types:

FE_DIVBYZERO	Pole error: division by zero, or some other asymptotically infinite result (from finite arguments).
FE_INEXACT	Inexact: the result is not exact.
FE_INVALID	Domain error: At least one of the arguments is a value for which the function is not defined.
FE_OVERFLOW	Overflow range error: The result is too large in magnitude to be represented as a value of the return type.
FE_UNDERFLOW	Underflow range error: The result is too small in magnitude to be represented as a value of the return type.

FPEs – Floating Point Exceptions

```c
// `feenableexcept()` is a GNU extension, not
// standard C. We need to define _GNU_SOURCE

#define _GNU_SOURCE
#include <fenv.h>
// make sure to link with -lm as well

int main(void){

   // combine options using binary OR
  feenableexcept(   FE_DIVBYZERO |
                    FE_INEXACT | FE_INVALID |
                    FE_OVERFLOW | FE_UNDERFLOW);

   // rest of your code
}
```

FPEs – Floating Point Exceptions

- With classic intel compiler:
  - Compile with
    ```
    $ icc main.c -fp-trap=mode -g
    ```
    - (don't need to do all the `feenableexcept()` stuff)
    - Modes:

`[no]divzero`	Enables or disables the IEEE trap for division by zero.
`[no]inexact`	Enables or disables the IEEE trap for inexact result.
`[no]invalid`	Enables or disables the IEEE trap for invalid operation.
`[no]overflow`	Enables or disables the IEEE trap for overflow.
`[no]underflow`	Enables or disables the IEEE trap for underflow.
`[no]denormal`	Enables or disables the trap for denormal.
`all`	Enables all of the above traps.
`none`	Disables all of the above traps.
`common`	Sets the most commonly used IEEE traps: division by zero, invalid operation, and overflow.

gdb + MPI: Poor Man's Parallel Debugger

```
$ mpirun -n 4  xterm -e gdb -ex run \
     --args your_program --arg1 --arg2
```

- Launch 4 MPI ranks
- Launch a terminal (`xterm`) and execute subsequent command (`-e`)
- Launch gdb and immediately execute „`run`" command
- gdb flag: The executable (`your_program`) will need command line arguments (`--arg1, --arg2`)
- Launch your program with extra cmdline arguments

- `MPI_Abort()` may exit gracefully instead of raising/signalling an error.
  - Simplest workaround: Replace with `abort()` while debugging.

valgrind

Track down memory leaks

- `$ valgrind path/to/your/executable`
- `$ valgrind --leak-check=full path/to/executable`
  - For more details, traces, etc.

Address Sanitizer

- `valgrind` is somewhat old software which might slow down your code drastically, require too much memory, or not be up-to-date with newer instructions.

- Using the address sanitizer may be a better choice.

- That is activated during compilation:
  - GCC example:
    ```
    $ gcc main.c -o ex -fsanitize=address
    ```
  - Works the same for `clang` and `icx`.

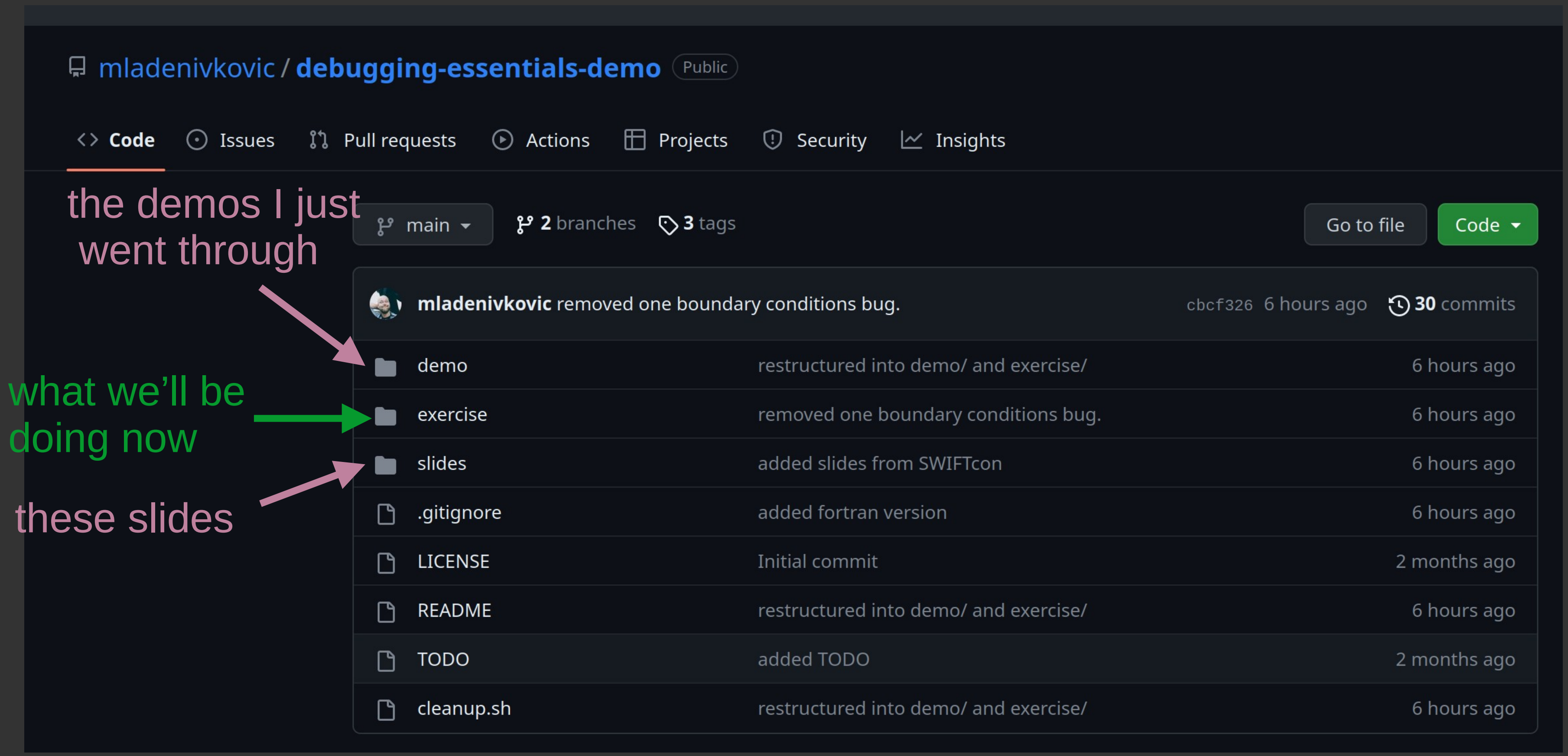
Additional Topics

Some additional tips and tricks with gdb, which aren't covered by the live demo.

- `display <variable>` :
  print variable every time it is touched throughout the run.
- `finish` :
  after a breakpoint, run function until the end, and halt there.
- `info locals` :
  print all local variables.
- `s (step)`:
  like `n (next)`, but execute a single step of the line, instead of the full line.
- `disable breakpoint/watchpoint` :
  disable breakpoint or watchpoint, but don't delete it. You can undo that with the `enable` command.
- `tbreak` :
  set temporary breakpoint. Is automatically deleted after being hit once.
- `tui enable` :
  start a fancy text user interface to look through program's source code. See https://sourceware.org/gdb/current/onlinedocs/gdb.html/TUI.html for more.

Hands-On Exercise

- Grab it from github:
  https://github.com/mladenivkovic/debugging-essentials-demo/



How the Game Is Played

- The `exercise/` directory contains a small toy code.
- I have planted bugs in that code.
- You go and find them!

  … after a short introduction that follows

Directory Contents

- exercise/src

  source files of toy code with planted bugs in C and in Fortran

- exercise/solution

  correct versions of the code.
  (You can compile and run this to see what the results should be.)

- exercise/theory

  TeX documentation on what code does and how it works.
  Everything you need to know about this code is on the first 2 pages!

- exercise/plot_solution.py

  python3 script which will plot the program's output for you

The Toy Code: `linear_advection`

- Solves the equation of linear advection in 1D:

$$\frac{\partial q}{\partial t} + a \cdot \frac{\partial q}{\partial x} = 0$$

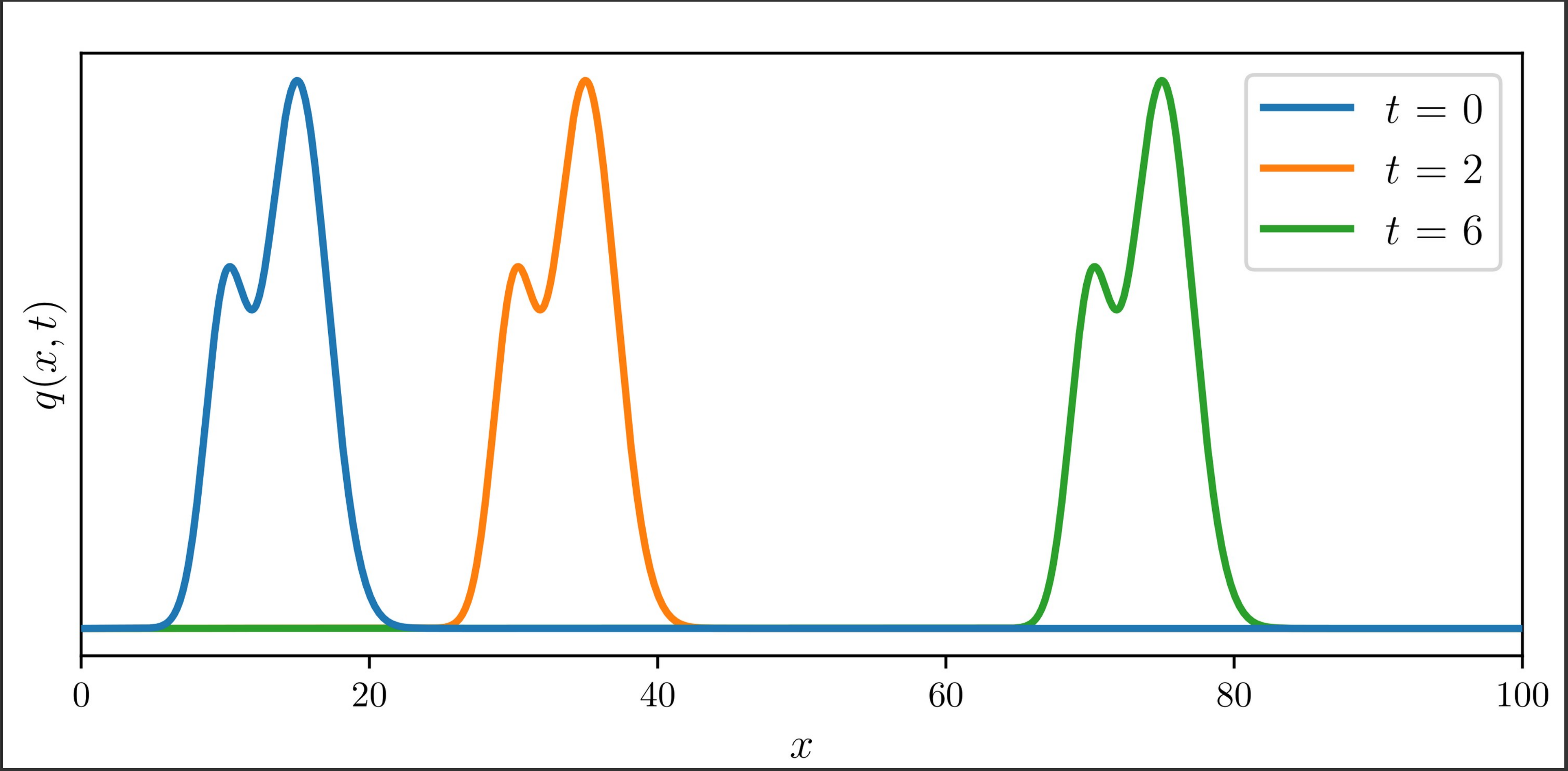
$$a = \text{ const } > 0.$$

- Using a very simple numerical method:

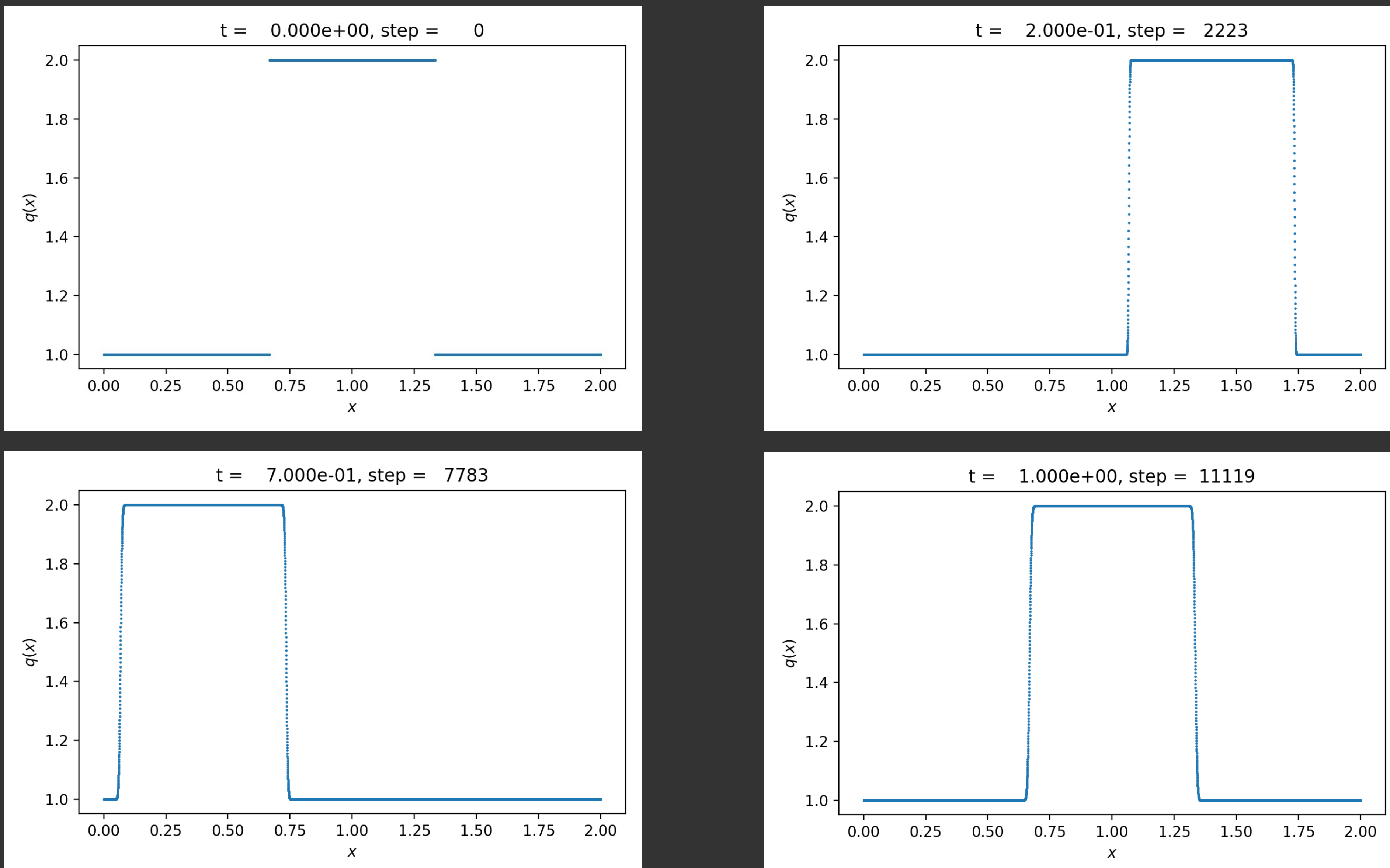
$$q_i^{n+1} = q_i^n + a\frac{\Delta t}{\Delta x}\left(q_{i-1}^n - q_i^n\right)$$

Analytical Solution

$$q(x,t) = q(x - a(t - t_0), t_0).$$

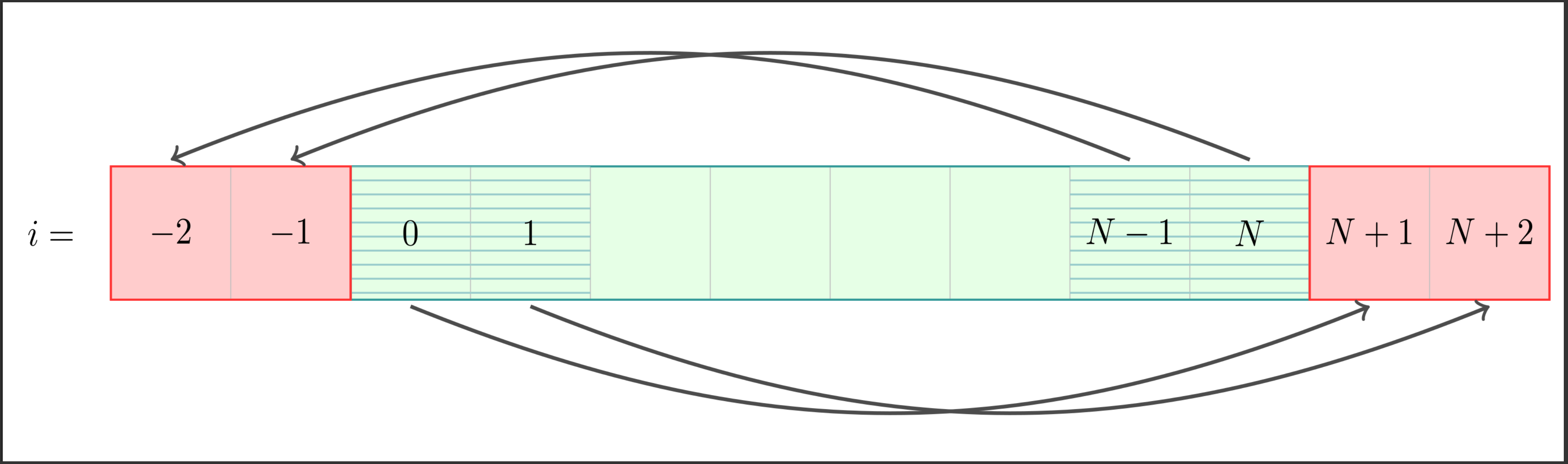


Numerical Solution



Periodic Boundary Conditions

- We add "ghost cells" along on the boundary
- Copy content of cells from boundary into ghost cells on other side



Base Algorithm of the Toy Code

- Initial Setup:
  - set $t_{current} = 0$
  - set up the initial states q_i^0 for each cell i.
- While $t_{current} < t_{end}$, solve the n-th time step:
  - Apply periodic boundary conditions
  - Find the maximal permissible time step Δt
  - For each cell i, find the updated states q_i^{n+1}
  - Update the current time: $t_{current} = t^{n+1} = t^n + \Delta t$

Determining the Time Step Size

- "CFL condition"

$$\Delta t_{max} = C_{cfl} \frac{\Delta x}{a}$$

- $C_{cfl} \in [0, 1)$: user set parameter
- Δx : cell width
- a: advection coefficient. Constant, > 0

Where to Start?

- Grab it from github:
  https://github.com/mladenivkovic/debugging-essentials-demo/
- Check out the theory document – everything you need to know about the code is in the first 2 pages
- Run and compile the code, see what happens
- Take a look at the `Makefile` – you might want to make some changes in there
- Try using new techniques to fix it!

Instructions for DINE

- The toy program should work on your own machines. You can however also work on DINE.
  - https://www.dur.ac.uk/icc/cosma/facilities/dine/
  - https://www.dur.ac.uk/icc/cosma/facilities/dine/notes/
- Required modules:
  - A compiler:
    - Intel Classic: `module load intel_comp/2018`
    - Intel oneAPI: `module load oneAPI/2022.3.0`
    - gnu: `module load gnu_comp/10.2.0`
    - LLVM: `module load llvm/13.0.0`
  - gdb: `module load gdb`
  - valgrind: `module load valgrind`
    - WARNING: needs gnu compiler: `module load gnu_comp/10.2.0`
  - Python: `module load python/3.9.1-C8`