Correctness and Debugging Workshop
Debugging Principles
or
The Essentials of gdb

Mladen Ivkovic
Durham University
November 2025

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
 $ 
```

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/de
 $ ./ex
Segmentation fault (core dumped)
19:56 [mivkov@mivkov-ThinkPad-X1-Carbon-6th] - ~/coding/de
 $
```

`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)
```

`printf` – The Ol' Reliable

- Rinse and repeat, until you find where the problem is
- **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

- 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.

External Debugging Tools

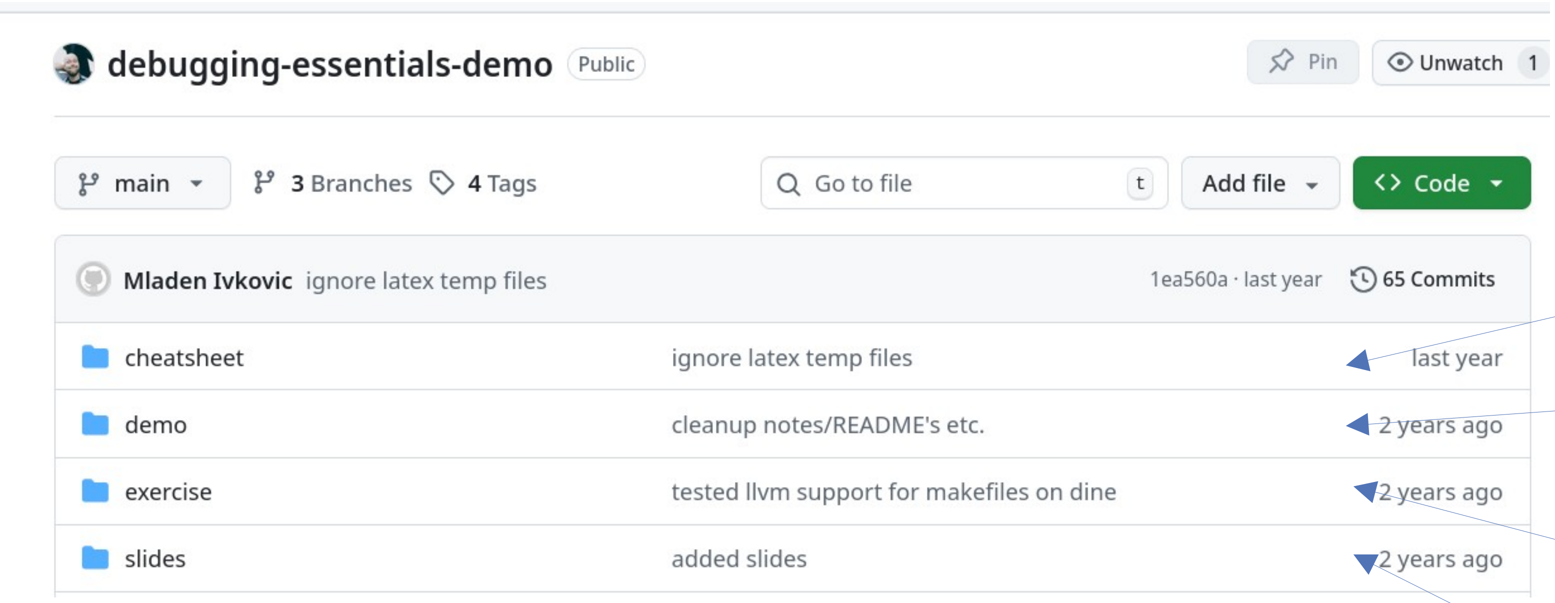
- There are tools which can help you to deal with them:
  - e.g. `gdb`, `DDT`, `lldb`, `mdb`, et al. for stack traces, live inspection, monitoring
  - e.g. `valgrind`, `address sanitizer` for memory leaks, memory issues
  - e.g. `MUST` for MPI issues
  - and many more

Today's Schedule: gdb

- Live demo how to use gdb
- Hands-on exercise
- **github.com/mladenivkovic/debugging-essentials-demo**

```
$ git clone git@github.com:mladenivkovic/debugging-essentials-demo.git
```

github.com/mladenivkovic/debugging-essentials-demo



gdb cheatsheet

Examples used in live-demo

Hands-on exercises

These (and older) slides

gdb – Basic Usage

1. Compilation:

   Compile your program with **debug symbols:**
   add `-g` or `-ggdb` flag

2. Running: run gdb executable

```
$ gdb /path/to/your/executable
$ gdb --args path/to/your/executable --arg1 --arg2
```

gdb – Basic Commands

`r (run)`	run the program
`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 (e.g. after hitting breakpoint, halting...)
`n (next)`	execute next line
`frame` or `where`	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 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>

gdb – More on Breakpoints

`info break` : show information on currently set breakpoints

`del 1` : delete breakpoint 1.

  You can find breakpoint numbers using `info break`

`break <location> if <condition>`

  Set a **conditional breakpoint** at location (e.g. line number, function) and break only if condition is satisfied, e.g.
  `break main.c:12 if i == 12 && j < 23`

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.

gdb – Watchpoints Scope Example

Variable `sum` in global scope

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


int sum;

int main(void){

  // do stuff
}
```

Variable `sum` **not** in global scope

```
// you need to set a
// breakpoint inside main()
// and then set a
// watchpoint for `sum`

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.

gdb – reverse: 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 second `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

When running code, it is possible to enable core dumps:

when program encounters error, write down what is currently in memory instead of just quitting

Typically writes `core.XXXX` files, where `XXXX` are some numbers.

`gdb` can read that back in and allow you to debug!

```
$ gdb -c core.XXXX path/to/your/executable
```

gdb – Core Dumps: How to Enable and Where to Find

Enabling core dumps on linux:

```
$ ulimit -S -c unlimited
```

Note: *your sysadmins may have disabled core dumps*.

Default core dump location may vary:

- On HPC systems, it's often set to `$workdir`
- Ubuntu 21+: `/var/lib/apport/coredump/`
- Manjaro: `/var/lib/systemd/coredump/`

gdb – Core Dumps With `coredumpctl`

```
$ coredumpctl info
```

Show info of last core dump.
To find out where it's stored, look for line starting with "`Storage`"

```
$ coredumpctl debug
```

Launch the debugger on the last available core dump

```
$ coredumpctl debug -debugger=/path/to/gdb
```

Specify which debugger to use

```
$ coredumpctl dump --output core.out
$ gdb -c core.out /path/to/your/executable
```

Some operating systems will compress their core dumps, making them unreadable by `gdb`. These two lines uncompress the last core dump into `core.out` and read them in using `gdb`

gdb – “value has been optimized out”

„value has been optimized out” workarounds:

- Compile entire program without optimisation: -O0 compiler flag
  (You may be able to recompile only a handful of files you’re interested in using that flag, leaving the rest optimised)
- 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, alternatively -----

#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.

FPEs – Types of Floating Point Exceptions

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 in C

```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
}
```

`gdb + MPI`: Poor Man's Parallel Debugger

- Running `gdb` with `MPI`: Simultaneously launch one `gdb` instance per `MPI rank` in a terminal emulator (`xterm`)

```
$ 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*

`gdb + MPI`: Poor Man’s Parallel Debugger

- **Warning**: depending on the implementaiton, `MPI_Abort()` may exit gracefully instead of raising/signalling an error, killing the `gdb` session instead of letting it catch and handle the abort.
- Simplest workaround: Replace `MPI_Abort()` with `abort()` while debugging.

gdb – Additional Topics, Tricks etc

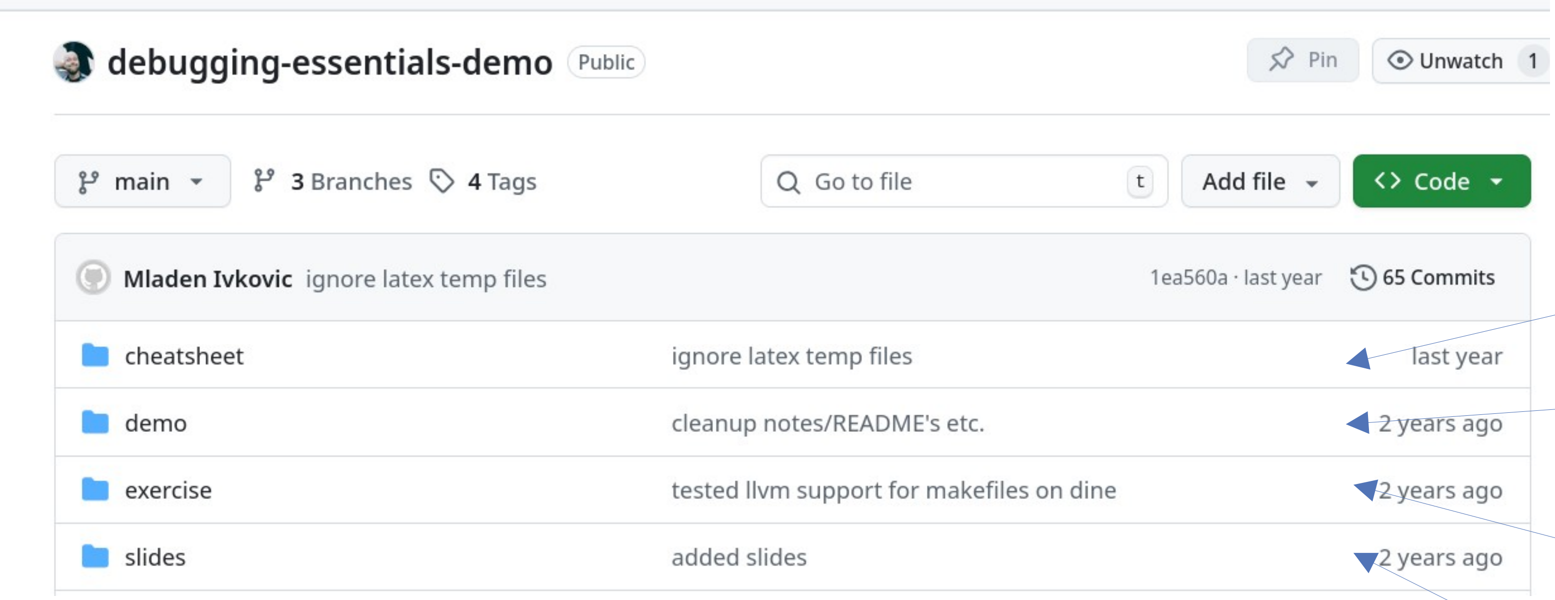
- `display <variable>`
  print variable every time it is touched throughout the run.
- `finish`
  after a breakpoint, run function until the end, and halt there.
- `info locals`
  print all local variables.
- `s (step)`
  like `n (next)`, but execute a single step of the line, instead of the full line.
- `disable <breakpoint-nr>/<watchpoint-nr>`:
  disable `breakpoint` or `watchpoint` by number (use `info break` or `info watch` to find number), but don't delete it.
  You can undo that with the `enable` command.
- `tbreak`
  set temporary breakpoint: will be automatically deleted after being hit once.

`gdb` – Additional Topics, Tricks etc

- `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.
- `make`: run `make` from within `gdb`
- shell : open a shell (`bash/zsh/csh` etc, your default) inside `gdb`
- python : open a `python` interpreter within gdb
- `save breakpoints <filename>`:
  Save `breakpoints` of current session into `<filename>`
- `source <filename>`:
  Load (e.g. saved `breakpoints`) from `<filename>`

Hands-On Exercise

```
$ git clone git@github.com:mladenivkovic/debugging-essentials-demo.git
```



gdb cheatsheet

Examples used in live-demo

Hands-on exercises

These (and older) slides

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:

$$\frac{\partial q}{\partial t} + a \cdot \frac{\partial q}{\partial x} = 0 \qquad\qquad a = \text{const} > 0.$$

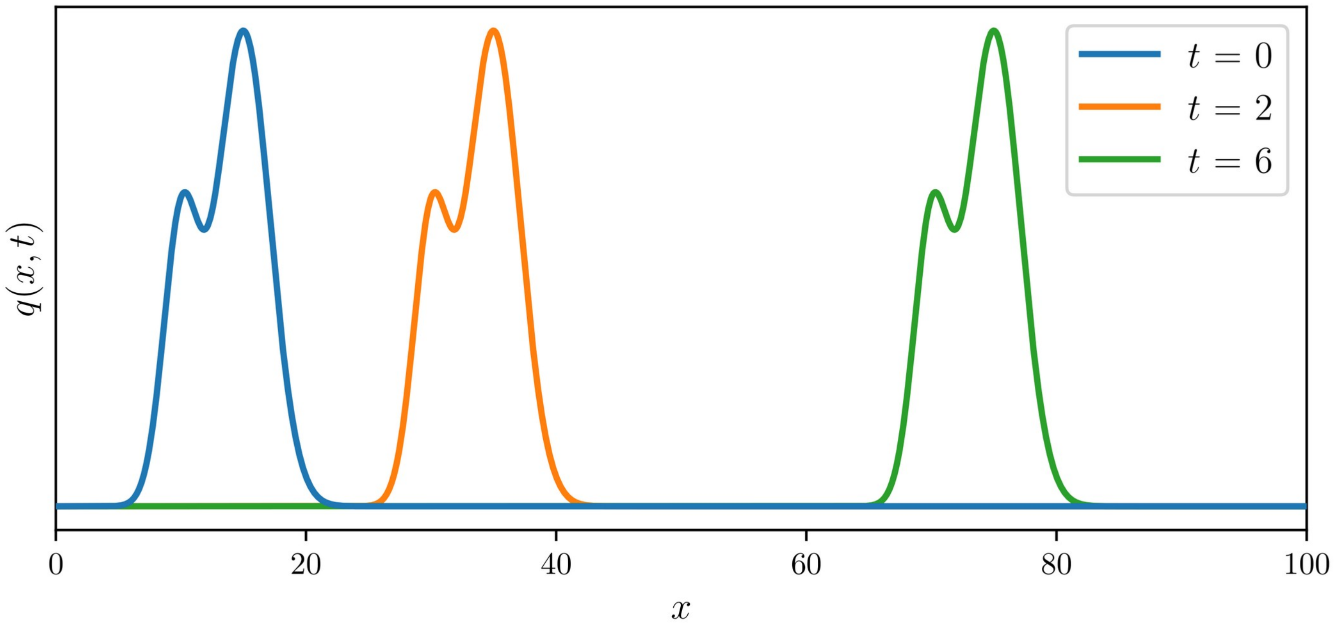
- Using a very simple numerical method:

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

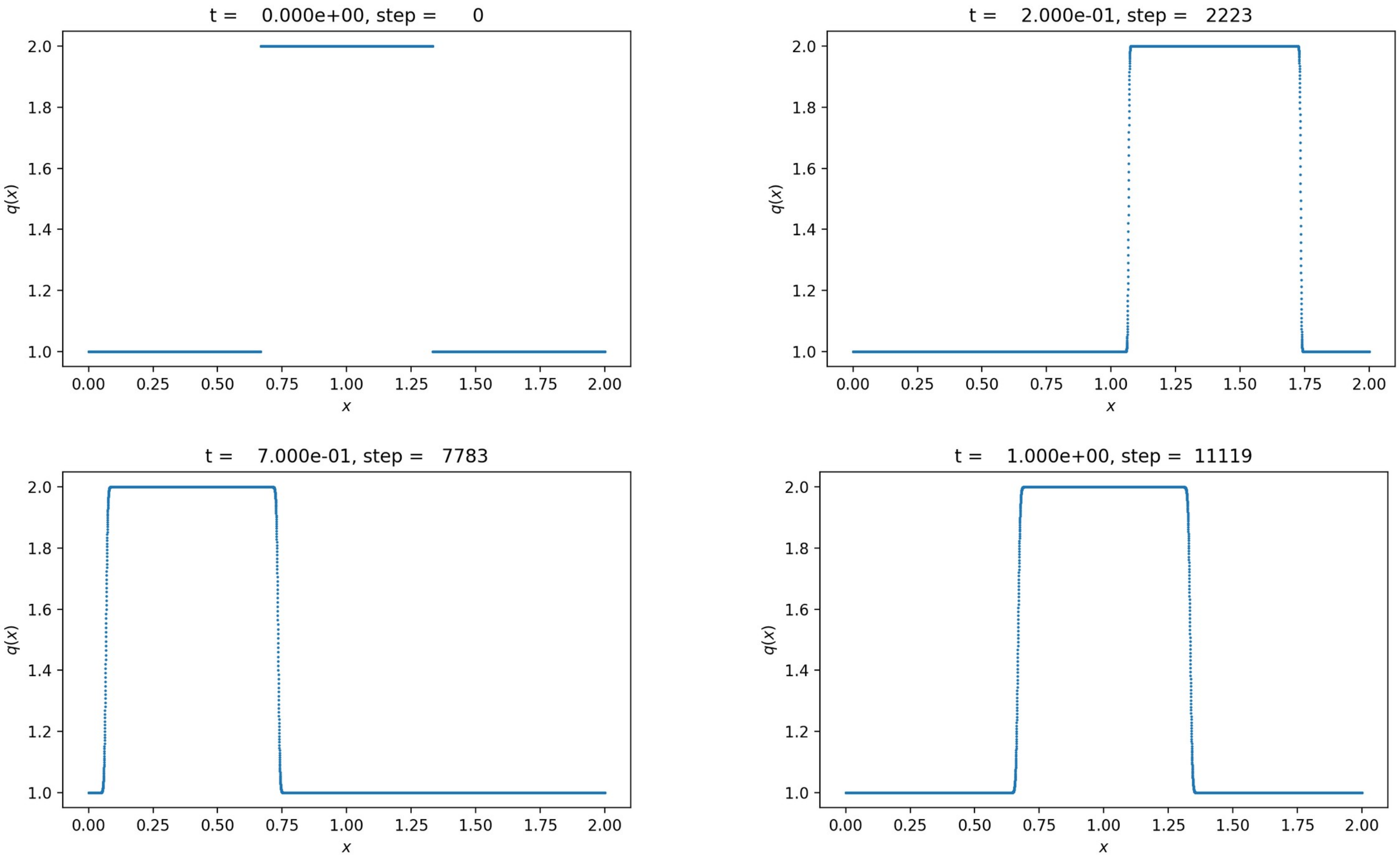
Analytical Solution

- Original solution at *$t = t_0$* moved (advected) by *$a(t - t_0)$*

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

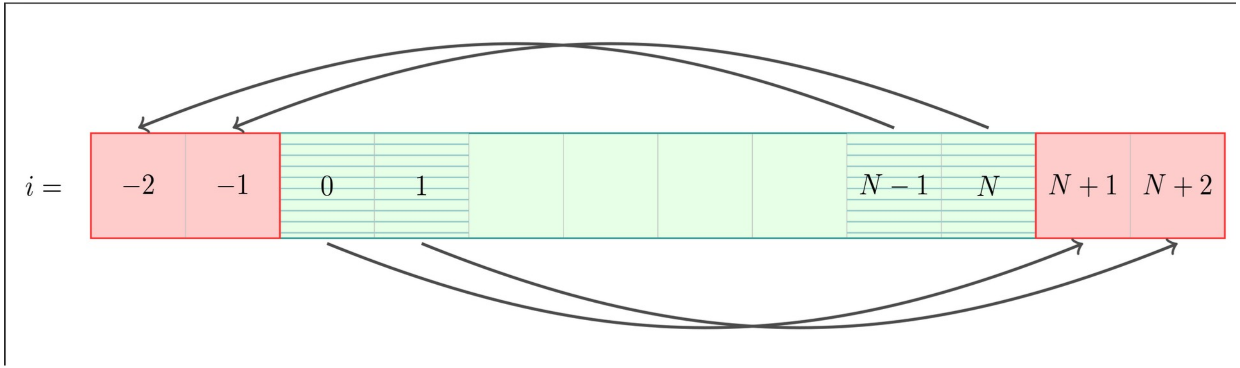


Numerical Solution



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, positive, non-zero.

Where to Start?

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

  ```
  $ git clone git@github.com:mladenivkovic/debugging-essentials-demo.git
  ```
- Check out the theory document – everything you need to know about the code is in the first 2 pages
- Compile and run the code, see what happens
- Take a look at the `Makefile` – you might want to make some changes in there
- Try using `gdb` to fix it!

Final Slide

Backup Slides

`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`.
- ***Note**: for more complex compilations* `-fsanitize=address` *must also be passed to the linker!*