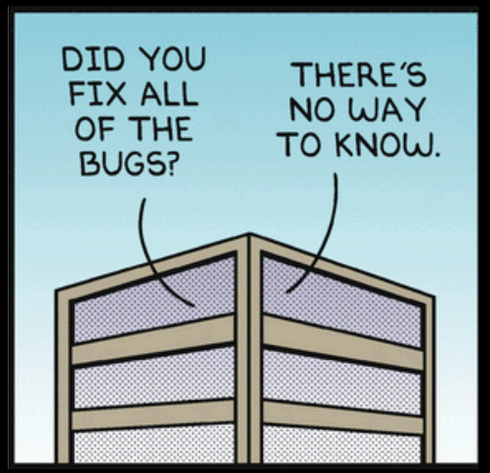
Best practices in scientific computing



Contents

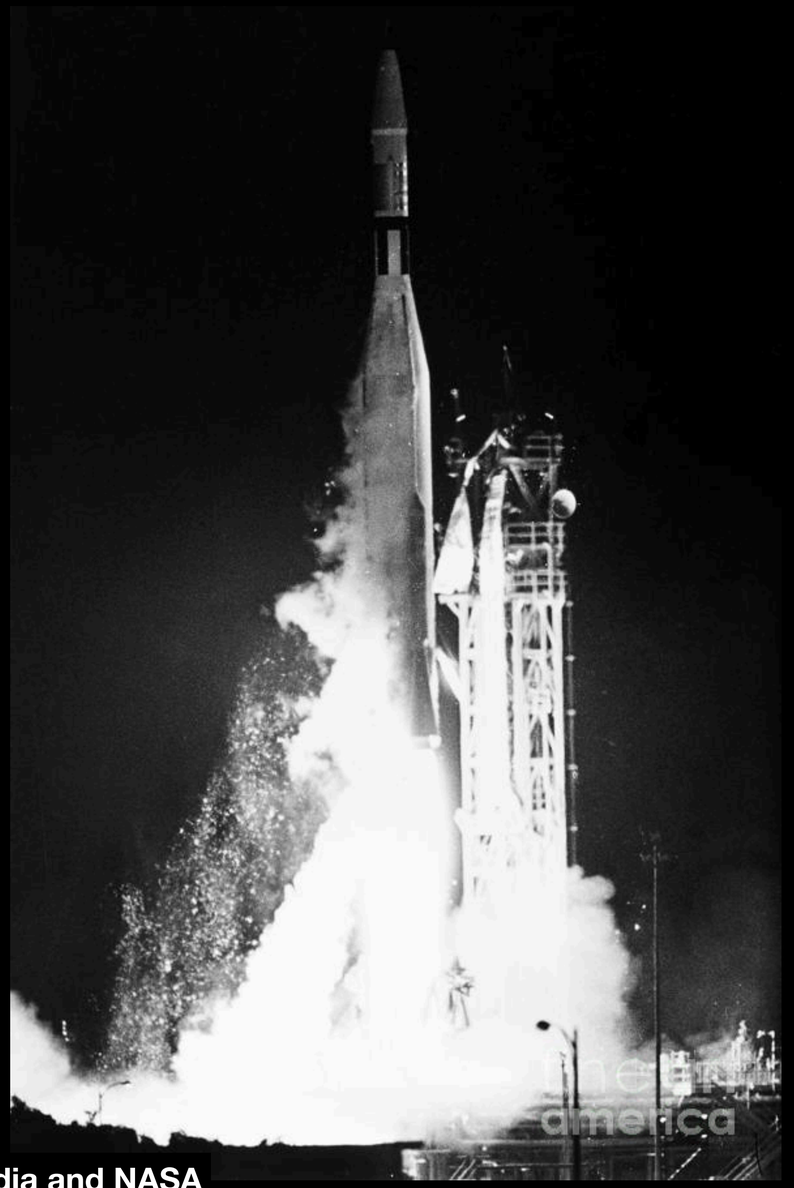
1. How bad software can be dangerous
2. How to manage your coding
3. Few suggestions about coding
4. A short howto for makefiles

Software failures: Mariner 1 spacecraft

Mariner 1 was the first spacecraft of the American Mariner program, designed for a planetary fly close to Venus ($18.5 million in 1962).
It was launched aboard an Atlas-Agena rocket on July 22, 1962.

Shortly after takeoff the rocket responded improperly to commands from the guidance systems on the ground, setting the stage for an apparent software-related guidance system failure.

With the craft effectively uncontrolled, a range safety officer ordered its destructive abort 294.5 seconds after launch.



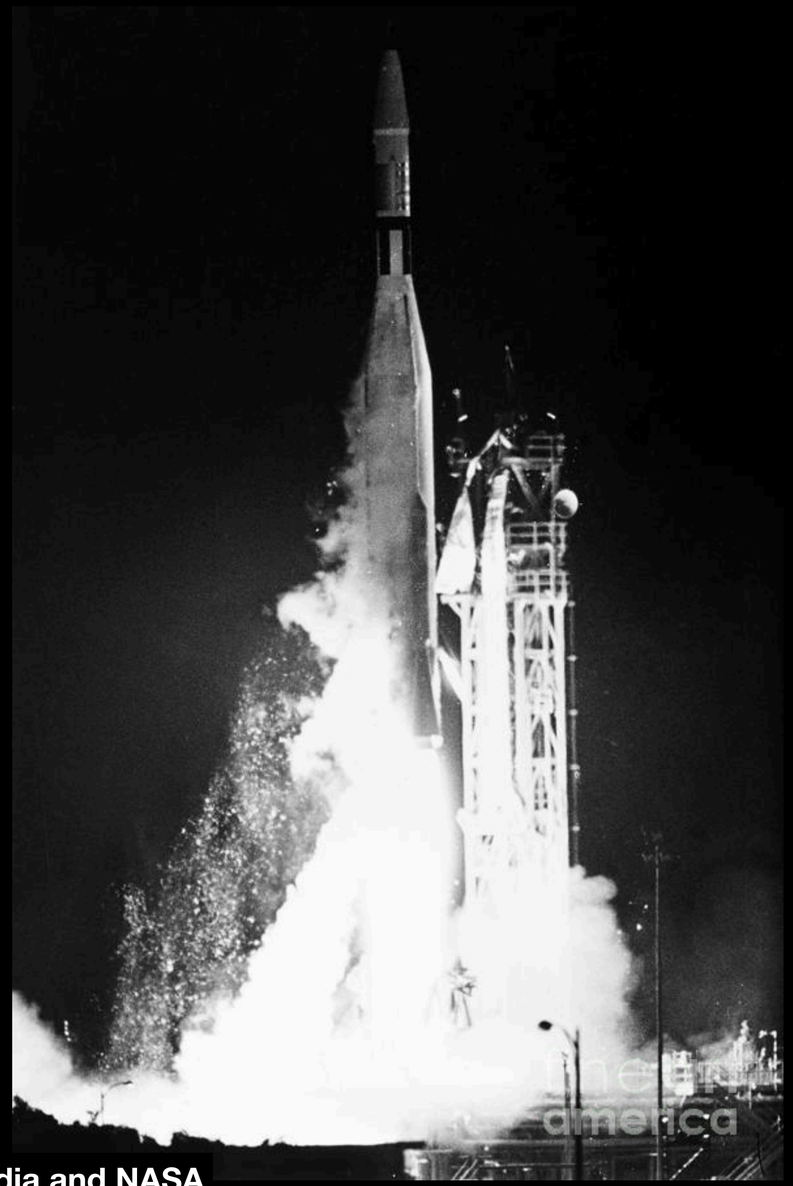
Software failures: Mariner 1 spacecraft

The most detailed and consistent account was that the error was in hand transcription of a mathematical symbol in the program specification for the guidance system, in particular a missing overbar

$$\partial_t \bar{R}_n$$

The coder missed the superscript bar which was meant "*the nth smoothed value of the time derivative of a radius R*".

Since the smoothing function indicated by the bar was left out of the specification for the program, the implementation treated normal minor variations of velocity as if they were serious, causing spurious corrections that sent the rocket off course.



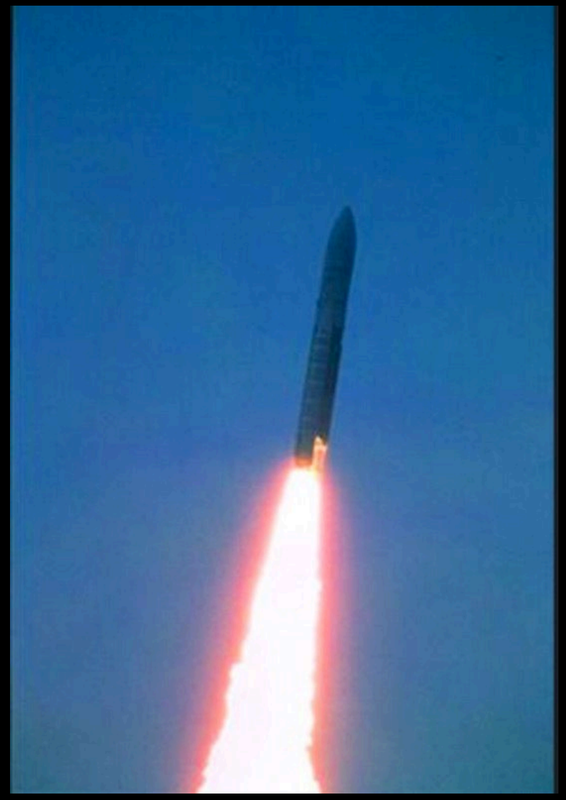
© Wikipedia and NASA

Software failures: Ariane 5 spacecraft

The morning of the 4th of June 1996 was partially cloudy at Kourou in Guyana as the European Space Agency (ESA) prepared for the first launch of the French-built Ariane 5 rocket. The rocket lifted off at 09:34.

Just 37 seconds later, the rocket veered on its side and began to break up. The range safety mechanism identified the impending catastrophe and initiated explosive charges that blew up the rocket to prevent further damages and possible casualties.

An investigation by the ESA determined that the accident was caused by a software 'bug.



© Ben-Ari, SIGCSE Bulletin 33 (2001) 58

Software failures: Ariane 5 spacecraft

1. The **internal navigation system** (**INS**) attempted to convert a 64-bit number to a 16- bit number without checking that 16 bits was sufficient to hold the value. This caused a runtime error.

2. The runtime error caused the **INS** software to stop execution.

3. The **INS** hardware (of both computers) sent a report of the error to the main computer.

4. The main computer erroneously interpreted and commanded the **nozzle** to fully deflect to one side.

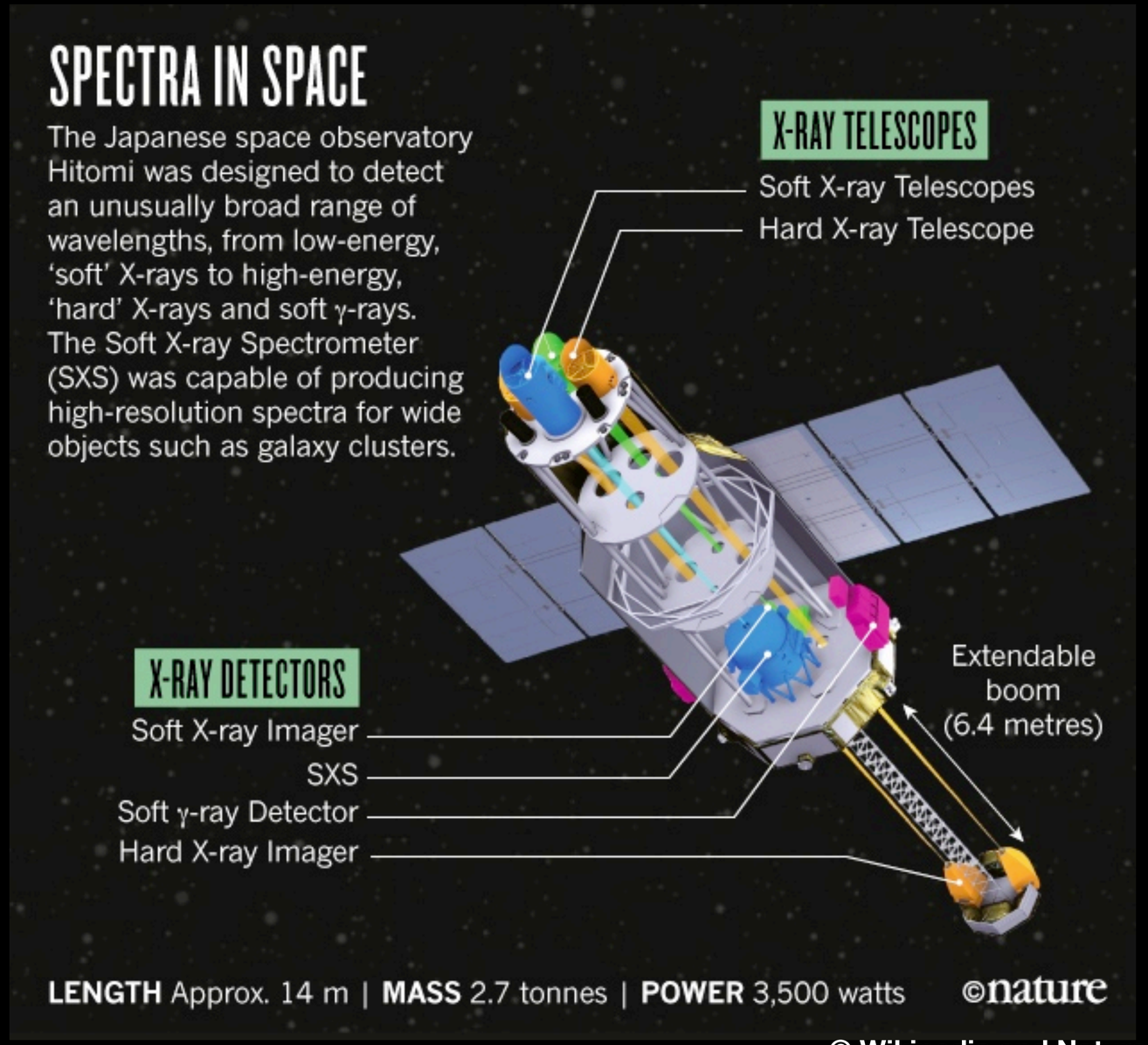
5. The rocket turned at a sharp angle and was subjected to forces that it was not designed to withstand. It began to break up and was destroyed.

© Ben-Ari, SIGCSE Bulletin 33 (2001) 58

Software failures: Hitomi (ひとみ) spacecraft

The space observatory was designed to investigate the hard X-ray band above 10 keV.

The spacecraft was launched on 17 February 2016 and contact was lost on 26 March 2016, due to multiple incidents with the attitude control system leading to an uncontrolled spin rate and breakup of structurally weak elements.



© Wikipedia and Nature

On 28 April, the Japan Aerospace Exploration Agency (JAXA) declared the satellite, on which it had spent 286 millions of US$, lost.
At least ten pieces — including both solar-array paddles that had provided electrical power — broke off the satellite's main body.

At 3:01 a.m. on 26 March, the spacecraft began a preprogrammed manoeuvre.
Hitomi relied on a set of gyroscopes to calculate its orientation in space.
But those gyroscopes were reporting, erroneously, that the spacecraft was rotating at a rate of about 20 degrees each hour.
Tiny motors known as reaction wheels began to turn to counteract the supposed rotation. Hitomi started to spun faster and faster.

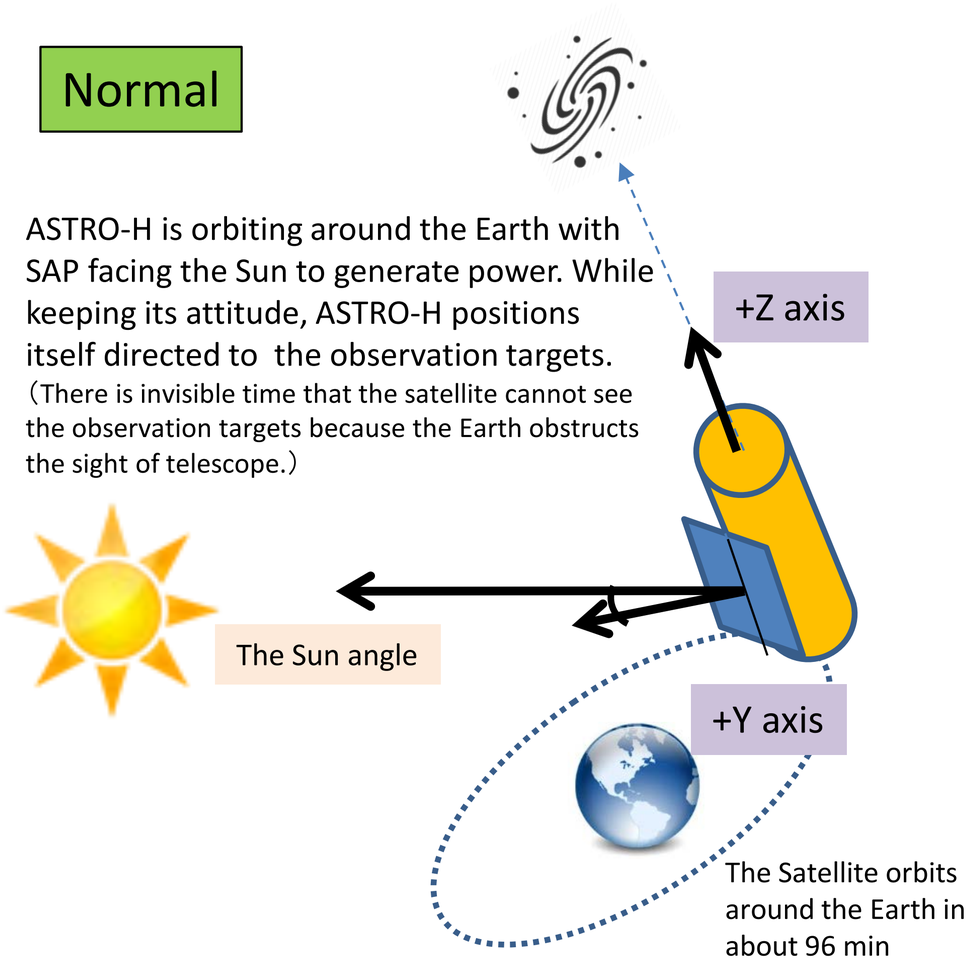
The spacecraft then automatically switched into a safe mode and, at about 4:10 a.m., **fired thrusters to try to stop the rotation**.
But because the wrong command had been uploaded, the firing caused the spacecraft to accelerate further.

Design and software errors

Schematic of ASTRO-H behavior under attitude anomaly

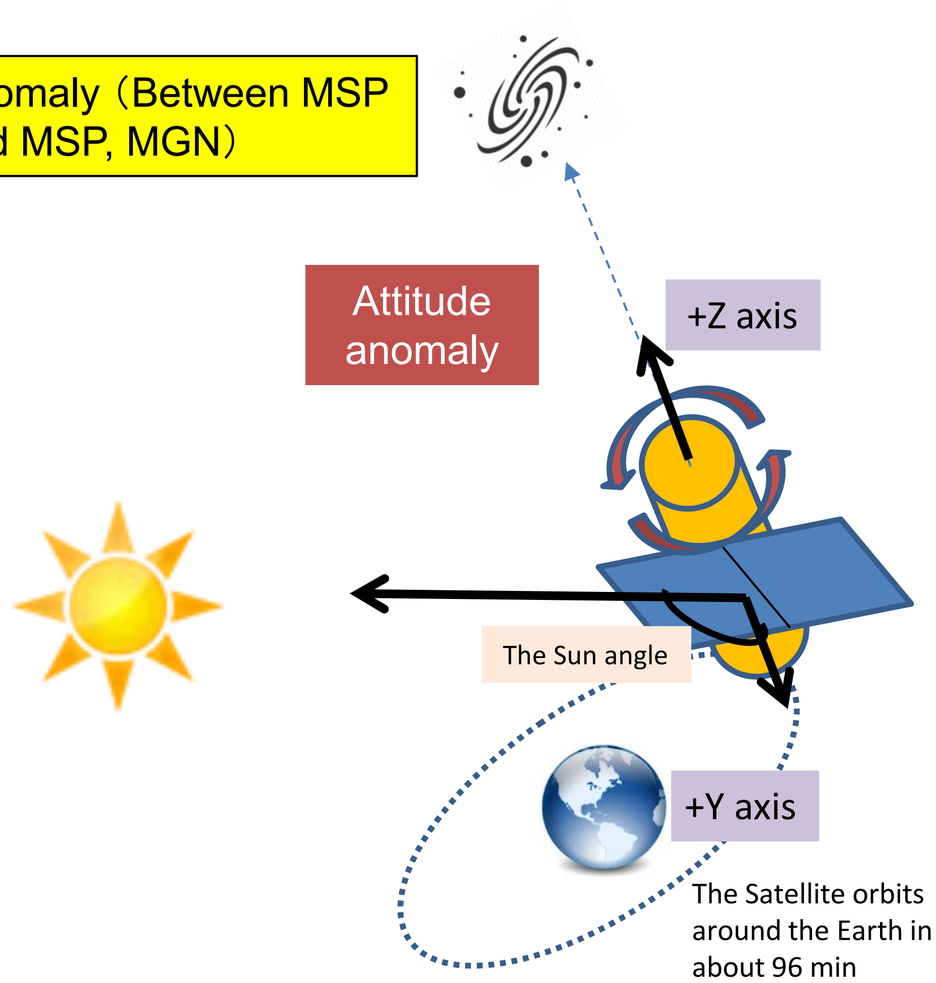
Normal

ASTRO-H is orbiting around the Earth with SAP facing the Sun to generate power. While keeping its attitude, ASTRO-H positions itself directed to the observation targets.
（There is invisible time that the satellite cannot see the observation targets because the Earth obstructs the sight of telescope.）



The nominal angle between satellite +Y axis and the Sun angle is within ±30 degrees.

Anomaly（Between MSP and MSP, MGN）

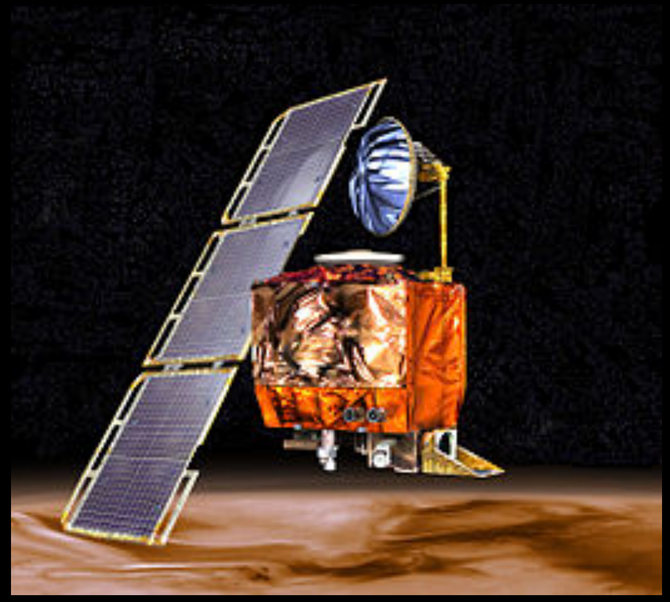


The IRU estimated error value continues, and ASTRO-H began to rotate along Z axis about 21.7 degree/hour slowly. The Sun angle at a time of the last telemetry reception at MGN was about 123 degrees.

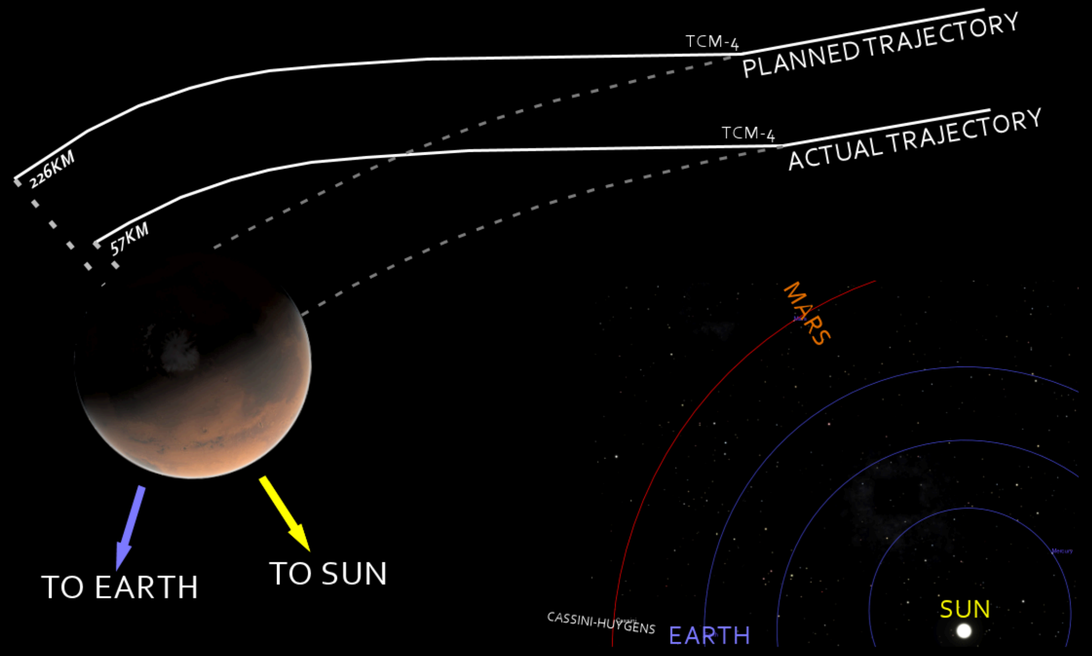
© http://global.jaxa.jp/projects/sat/astro_h/files/topics_20160415.pdf

Software failures: Mars Climate Orbiter

The ***Mars Climate Orbiter*** was launched by NASA on December 11, 1998 to study the Martian atmosphere.



On September 23, 1999, communication with the spacecraft was lost as the spacecraft went into orbital insertion, due to ground-based computer software which produced output in **non-SI units** of pound-force seconds (lbf·s) instead of the **SI units** of newton-seconds (N·s).



The spacecraft encountered Mars on a trajectory that brought it too close to the planet, and it was either destroyed in the atmosphere or re-entered heliocentric space after leaving Mars' atmosphere.

Software failures: Therac 25

The Therac-25 was a radiation therapy machine, also known as a linear accelerator that was used to treat cancer patients.

The machine was used to eliminate cancerous tissues and remove excess tumours found in cancer patients.

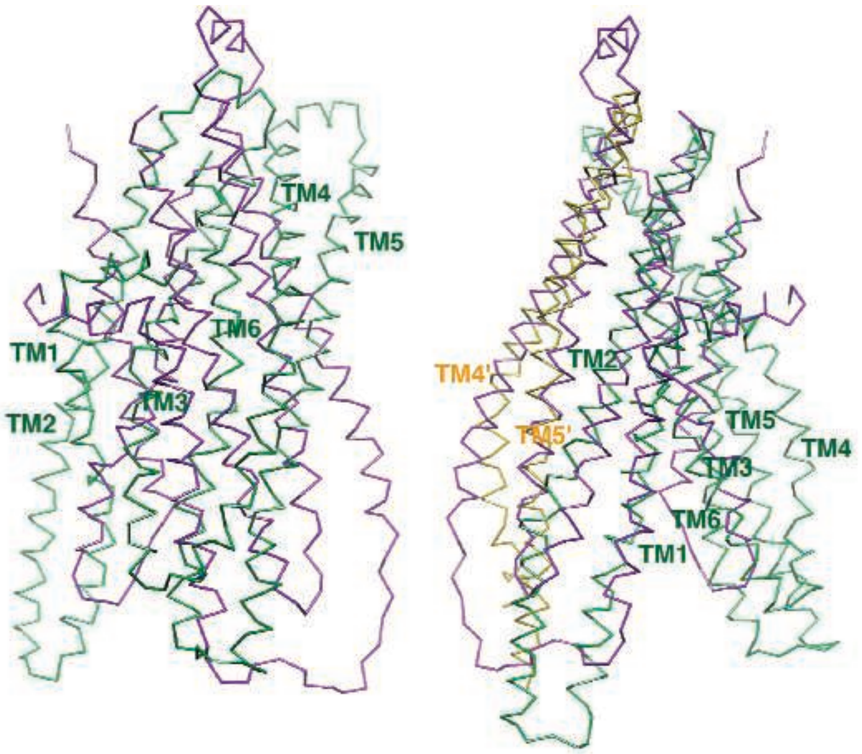
Six accidents happened leading to three deaths and to long-term effects in three other patients. An overdose diagnosis is of more than 1000 rads whereas a regular dose is only about 200 rads.

Following the fourth and fifth accidents, it was found that the company lacked documentation for system errors. In these accidents, it was described that a '***Malfunction 54***' error had appeared on the terminal screen, but it was mostly unknown what the errors meant. The deficiency of information led to operators ignoring the error messages and further carried on the procedure of the machine.

Software failures: ABC transporter

SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Software Problem Leads to Five Retractions



Flipping fiasco. The structures of MsbA (purple) and Sav1866 (green) overlap little (*left*) until MsbA is inverted (*right*).

Geoffrey Chang wrote in 2001 a Science paper, which described the structure of a protein called MsbA, isolated from the bacterium Escherichia coli.

Later, in September, Swiss researchers published a paper in Nature that cast serious doubt on the protein structure of Chang.

Chang discovered that a **homemade data-analysis program had flipped two columns of data**, inverting the electron-density map from which his team had derived the final protein structure. Unfortunately, his group had used the program to analyze data for other proteins.

As a result, Chang and his colleagues retract three Science papers and report that two papers in other journals also contain erroneous structures.

How to manage your coding: tips and tricks

OPEN ACCESS Freely available online

PLOS BIOLOGY

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Guy[6¤], Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11], Ethan P. White[12], Paul Wilson[13]

PLOS COMPUTATIONAL BIOLOGY

PERSPECTIVE

Good enough practices in scientific computing

Greg Wilson[1☯]*, Jennifer Bryan[2☯], Karen Cranston[3☯], Justin Kitzes[4☯], Lex Nederbragt[5☯], Tracy K. Teal[6☯]

1 Software Carpentry Foundation, Austin, Texas, United States of America, **2** RStudio and Department of Statistics, University of British Columbia, Vancouver, British Columbia, Canada, **3** Department of Biology, Duke University, Durham, North Carolina, United States of America, **4** Energy and Resources Group, University of California, Berkeley, Berkeley, California, United States of America, **5** Centre for Ecological and Evolutionary Synthesis, University of Oslo, Oslo, Norway, **6** Data Carpentry, Davis, California, United States of America

1. Write programs for people, not computers.

(a) A program should not require its readers to hold more than a handful of facts in memory at once.

(b) Make names consistent, distinctive, and meaningful.

(c) Make code style and formatting consistent.

Scientists writing software need to write code that both executes correctly and can be easily read and understood by other programmers (especially the author's future self).

(a) The primary way to accomplish this is to break programs up into easily understood functions, each of which conducts a single, easily understood, task.

1. Write programs for people, not computers.

(a) A program should not require its readers to hold more than a handful of facts in memory at once.

(b) Make names consistent, distinctive, and meaningful.

(c) Make code style and formatting consistent.

Scientists writing software need to write code that both executes correctly and can be easily read and understood by other programmers (especially the author's future self).

(b) Avoid non-descriptive names, like `a` and `foo`, or names that are very similar, like `results` and `results2`

1. Write programs for people, not computers.

(a) A program should not require its readers to hold more than a handful of facts in memory at once.

(b) Make names consistent, distinctive, and meaningful.

(c) Make code style and formatting consistent.

Scientists writing software need to write code that both executes correctly and can be easily read and understood by other programmers (especially the author's future self).

(c) If different parts of a scientific paper used different formatting and capitalization, it would make that paper more difficult to read. Homogeneity and Consistency!

2. Let the computer do the work.

(a) Make the computer repeat tasks.

(b) Save recent commands in a file for re-use.

(c) Use a build tool to automate workflows.

Scientists should make the computer repeat tasks and save recent commands in a file for re-use.

"do this again" saves time and reduces errors

A file containing commands for an interactive system is often called a script, though there is real no difference between this and a program.

Use Make (**http://www.gnu.org/software/make**)

3. Make incremental changes.

(a) Work in small steps with frequent feedback and course correction.

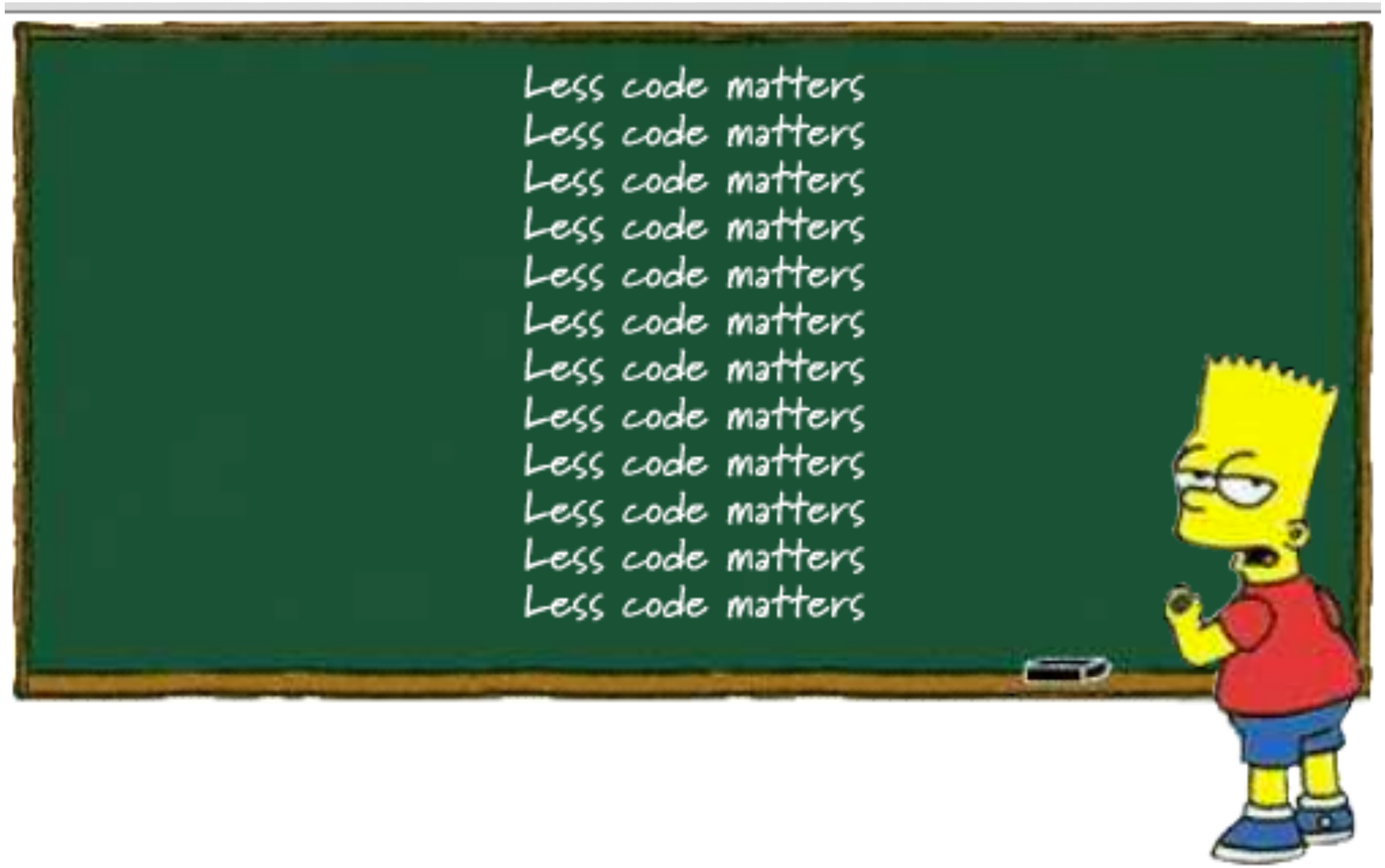
(b) Use a version control system.

(c) Put everything that has been created manually in version control.

Two of the biggest challenges scientists and other programmers face when working with code and data are keeping track of changes, and collaborating on a program or dataset: **Version Control System**

A VCS stores snapshots of a project's files in a repository (or a set of repositories). Programmers can modify their working copy of the project at will, then commit changes to the repository when they are satisfied with the results to share them with colleagues. Some of them are open source and freely available, including Git (**http://git-scm.com**), Subversion (**http://subversion. apache.org**), and Mercurial (**http://mercurial.selenic.com**). Many free hosting services are available as well: GitHub (**https:// github.com**), BitBucket (**https://bitbucket.org**), SourceForge (**http://sourceforge.net**), and Google Code (**http://code.google. com**)

The **DRY** Principle



4. Don't repeat yourself (or others).

(a) Every piece of data must have a single authoritative representation in the system.

(b) Modularize code rather than copying and pasting.

(c) Re-use code instead of rewriting it.

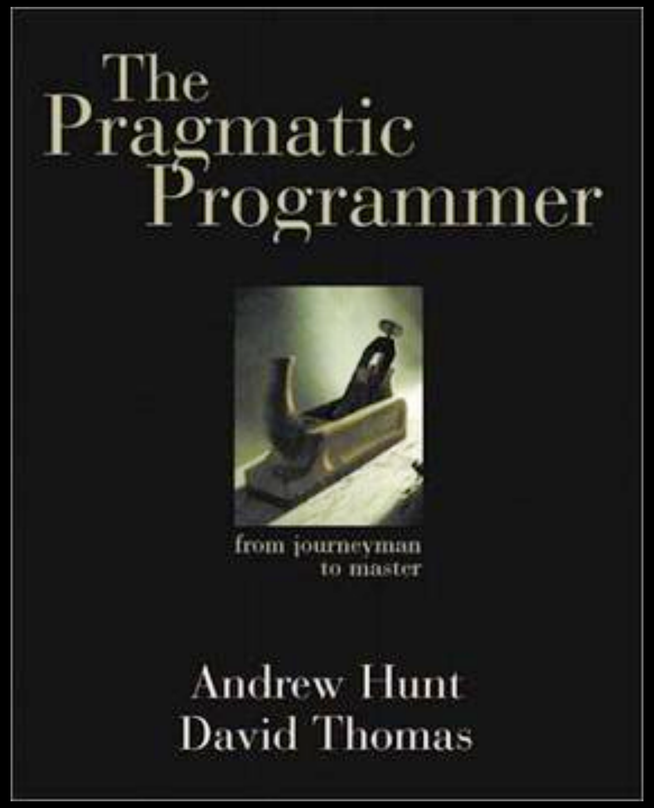
Follow the **DRY** Principle (don't repeat yourself)

At small scales, **modularize code** rather than copying and pasting. Avoiding "code clones" has been shown to reduce error rates: when a change is made or a bug is fixed, that change or fix takes effect everywhere.

At larger scales, it is vital that scientific programmers re-use code instead of rewriting it. Open source software is freely available on the web. It is typically better to **find an established library or package** that solves a problem than to attempt to write one's own routines.

The **DRY** Principle

…Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.



5. Plan for mistakes.

(a) Add assertions to programs to check their operation.
(b) Use an off-the-shelf unit testing library.
(c) Turn bugs into test cases.
(d) Use a symbolic debugger.

The first line of defense is **defensive programming**. Experienced programmers add assertions to programs to check their operation because experience has taught them that everyone (including their future self) makes mistakes. **An assertion is simply a statement that something holds true at a particular point in a program**.

5. Plan for mistakes.

(a) Add assertions to programs to check their operation.
(b) Use an off-the-shelf unit testing library.
(c) Turn bugs into test cases.
(d) Use a symbolic debugger.

The second layer of defense is **automated testing**. Automated tests can check to make sure that a single unit of code is returning correct results (unit tests), that pieces of code work correctly when combined (integration tests), and that the behavior of a program doesn't change when the details are modified (regression tests).

Debuggers are usually more productive than adding and removing print statements or scrolling through hundreds of lines of log output.

6. Optimize software only after it works correctly.

(a) Use a profiler to identify bottlenecks.
(b) Write code in the highest-level language possible.

Research has confirmed that most programmers write roughly the same number of lines of code per unit time regardless of the language they use. Since faster, lower level, languages require more lines of code to accomplish the same task, scientists are most productive when they write code in the highest-level language possible, and shift to low- level languages like C and Fortran only when they are sure the performance boost is needed.

Try to learn more than one language

7. Document design and purpose, not mechanics.

(a) Document interfaces and reasons, not implementations.

(b) Refactor code in preference to explaining how it works.

(c) Embed the documentation for a piece of software in that software.

Good documentation helps people understand code. This makes the code more reusable and lowers maintenance costs. However, inline documentation that recapitulates code is not so useful. **The scientific programmers should document interfaces and reasons, not implementations**.

The best way to create and maintain reference documentation is to embed the documentation for a piece of software in that software. Doing this increases the probability that when programmers change the code, they will update the documentation at the same time (**Doxygen**)

About documenting the code

One of the traits of good code is that it is self-documenting. In other words, the code itself (the variable names, method names, clear logic, etc.) clearly explains to the developer what is going on. In contrast, bad code is littered with **unnecessary comments**. Something like the following

```
/**
 * Add and return the sum of
 * num1 and num 2
 */

function sum(num1, num2) {
    return num1 + num2;
}
```

8. Collaborate.

(a) Use pre-merge code reviews.

(b) Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.

(c) Use an issue tracking tool.

A large body of research has shown that code reviews are the most cost-effective way of finding bugs in code. They are also a good way to spread knowledge and good practices around a team.

Why don’t you try the philosophy

***readable*, *reusable* and *testable*?**

readable, *reusable*, and *testable*

2. Software

   a Place a brief explanatory comment at the start of every program.

   b Decompose programs into functions.

   c Be ruthless about eliminating duplication.

   d Always search for well-maintained software libraries that do what you need.

   e Test libraries before relying on them.

   f Give functions and variables meaningful names.

   g Make dependencies and requirements explicit.

   h Do not comment and uncomment sections of code to control a program's behavior.

   i Provide a simple example or test data set.

   j Submit code to a reputable DOI-issuing repository.

- **Place a brief explanatory comment at the start of every program (2a)**, no matter how short it is. That comment should include at least 1 example of how the program is used; remember, a good example is worth a thousand words. Where possible, the comment should

- **Decompose programs into functions (2b)** that are no more than 1 page (about 60 lines) long. A function is a reusable section of software that can be treated as a black box by the rest of the program. The syntax for creating functions depends on programming language, but generally, you name the function, list its input parameters, and describe what information it produces. Functions should take no more than 5 or 6 input parameters and should not reference outside information.
  The key motivation here is to fit the program into the most limited memory of all: ours. Human short-term memory is famously incapable of holding more than about 7 items at once [15]. If we are to understand what our software is doing, we must break it into chunks that obey this limit, then create programs by combining these chunks. Putting code into functions also makes it easier to test and troubleshoot when things go wrong.

- **Be ruthless about eliminating duplication (2c)**. Write and reuse functions instead of copying and pasting code, and use data structures like lists instead of creating many closely related variables, e.g., create `score = (1, 2, 3)` rather than `score1`, `score2`, and `score3`. Also, look for well-maintained libraries that already do what you're trying to do. All programming languages have libraries that you can import and use in your code. This is code that people have already written and made available for distribution that has a particular function. For instance, there are libraries for statistics, modeling, mapping, and many more. Many languages catalog the libraries in a centralized source, for instance, R has CRAN, Python has PyPI, and so on. Thus, **always search for well-maintained software libraries that do what you need (2d)** before writing new code yourself, but **test libraries before relying on them (2e)**.

- **Give functions and variables meaningful names (2f)**, both to document their purpose and to make the program easier to read. As a rule of thumb, the greater the scope of a variable, the more informative its name should be; while it's acceptable to call the counter variable in a loop `i` or `j`, things that are reused often, such as the major data structures in a program, should *not* have 1-letter names. Remember to follow each language's conventions for names, such as `net_charge` for Python and `NetCharge` for Java.

- **Make dependencies and requirements explicit (2g)**. This is usually done on a per-project rather than per-program basis, i.e., by adding a file called something like `requirements.txt` to the root directory of the project or by adding a "Getting Started" section to the `README` file.
- **Do not comment and uncomment sections of code to control a program's behavior (2h)**, since this is error prone and makes it difficult or impossible to automate analyses. Instead, put if/else statements in the program to control what it does.
- **Provide a simple example or test data set (2i)** that users (including yourself) can run to determine whether the program is working and whether it gives a known correct output for a simple known input. Such a "build-and-smoke test" is particularly helpful when supposedly innocent changes are being made to the program or when it has to run on several different machines, e.g., the developer's laptop and the department's cluster.
- **Submit code to a reputable DOI-issuing repository (2j)** upon submission of paper, just as you do with data. Your software is as much a product of your research as your papers and should be as easy for people to credit. DOIs for software are provided by Figshare and Zenodo. Zenodo integrates directly with GitHub.

4. Project organization

   a Put each project in its own directory, which is named after the project.

   b Put text documents associated with the project in the `doc` directory.

   c Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory.

   d Put project source code in the `src` directory.

   e Put external scripts or compiled programs in the `bin` directory.

- **Ensure that raw data are backed up in more than one location (1b)**. If external hard drives are used, store them off-site of the original location. Universities often have their own data-storage solutions, so it is worthwhile to consult with your local Information Technology (IT) group or library. Alternatively, cloud computing resources, like Amazon Simple Storage Service (Amazon S3), Google Cloud Storage, or Microsoft Azure are reasonably priced and reliable. For large data sets, for which storage and transfer can be expensive and time-consuming, you may need to use incremental backup or specialized storage systems, and people in your local IT group or library can often provide advice and assistance on options at your university or organization as well.

- **Create the data you wish to see in the world (1c)**. Create the data set you wish you had received. The goal here is to improve machine and human readability, but not to do vigorous data filtering or add external information. Machine readability allows automatic processing using computer programs, which is important when others want to reuse your data. Specific examples of nondestructive transformations that we recommend at the beginning of analysis include the following:
  File formats: Convert data from closed, proprietary formats to open, nonproprietary formats that ensure machine readability across time and computing setups [9]. Good options include CSV for tabular data, JSON, YAML, or XML for nontabular data such as graphs (the node-and-arc kind), and HDF5 for certain kinds of structured data.
  Variable names: Replace inscrutable variable names and artificial data codes with self-explaining alternatives, e.g., rename variables called `name1` and `name2` to `personal_name` and `family_name`, recode the treatment variable from `1` vs. `2` to `untreated` vs. `treated`, and replace artificial codes for missing data, such as "`-99`," with `NA`, a code used in most programming languages to indicate that data are "Not Available" [10].

- **Create analysis-friendly data (1d)**. Analysis can be much easier if you are working with so-called "tidy" data [5]. Two key principles are as follows:
  Make each column a variable: Don’t cram 2 variables into one; e.g., "male_treated" should be split into separate variables for sex and treatment status. Store units in their own variable or in metadata, e.g., "3.4" instead of "3.4kg".
  Make each row an observation: Data often come in a wide format, because that facilitated data entry or human inspection. Imagine 1 row per field site and then columns for measurements made at each of several time points. Be prepared to gather such columns into a variable of measurements, plus a new variable for time point. Fig 1 presents an example of such a transformation.

- **Record all the steps used to process data (1e)**. Data manipulation is as integral to your analysis as statistical modeling and inference. If you do not document this step thoroughly, it is impossible for you or anyone else to repeat the analysis.
  The best way to do this is to write scripts for *every* stage of data processing. This might feel frustratingly slow, but you will get faster with practice. The immediate payoff will be the ease with which you can redo data preparation when new data arrive. You can also reuse data preparation steps in the future for related projects. For very large data sets, data preparation may also include writing and saving scripts to obtain the data or subsets of the data from remote storage.
  Some data-cleaning tools, such as OpenRefine, provide a graphical user interface but also automatically keep track of each step in the process. When tools like these or scripting are not feasible, it's important to clearly document every manual action (what menu was used, what column was copied and pasted, what link was clicked, etc.). Often, you can at least capture *what* action was taken, if not the complete *why*. For example, choosing a region of interest in an image is inherently interactive, but you can save the region chosen as a set of boundary coordinates.

…Last year's global fuss over the release of climate-science e-mails from the University of East Anglia (UEA) in Norwich, UK, highlighted the issue, and led the official inquiry to call for scientists to publish code. My efforts pre-date the UEA incident and grew from work in 2008 based on software used by NASA to report global temperatures. Released on its website in 2007, the NASA code was messy and proved difficult for critics to run on their own computers. Most did not seem to try very hard, and nonsense was written about fraud and conspiracy. **Openness improved both the code used by the scientists and the ability of the public to engage with their work**. This is to be expected. Other scientific methods improve through peer review. The open-source movement has led to rapid improvements within the software industry. But science source code, not exposed to scrutiny, cannot benefit in this way.

WORLD VIEW *A personal take on events*

Publish your computer code: it is good enough

Freely provided working code — whatever its quality — improves programming and enables others to engage with your research, says **Nick Barnes**.

- **It is not common practice!**

  It should be

- **People will demand for support**

  Not publishing can draw allegations of fraud. Which is worse? Nobody is entitled to demand technical support for freely provided code

- **Avoid abandonware**

POLICYFORUM

COMPUTATIONAL SCIENCE

Troubling Trends in Scientific Software Use

Lucas N. Joppa, [1*] Greg McInerny, [12] Richard Harper, [1] Lara Salido, [3] Kenji Takeda, [1] Kenton O'Hara, [1] David Gavaghan, [2] Stephen Emmott[1]

"Many of these scientists rely on the fact that the software has appeared in a peer-reviewed article, recommendations, and personal opinion, as their reason for adopting software. ***This is scientifically misplaced, as the software code used to conduct the science is not formally peer-reviewed****. This is especially important when a disconnect occurs between equations and algorithms published in peer-reviewed literature and how those are actually implemented in software reportedly used in those papers…"*

Science code manifesto

http://sciencecodemanifesto.org/

Code

All source code written specifically to process data for a published paper must bee available to the reviewers and readers of the paper

©

The copyright ownership and license of any released source code must be clearly stated

Science code manifesto

http://sciencecodemanifesto.org/

Citation

Researchers who use or adapt science source code in their research must credit the code's creators in resulting publications

Curation

Source code must remain available, linked to related materials, for the useful lifetime of the publication

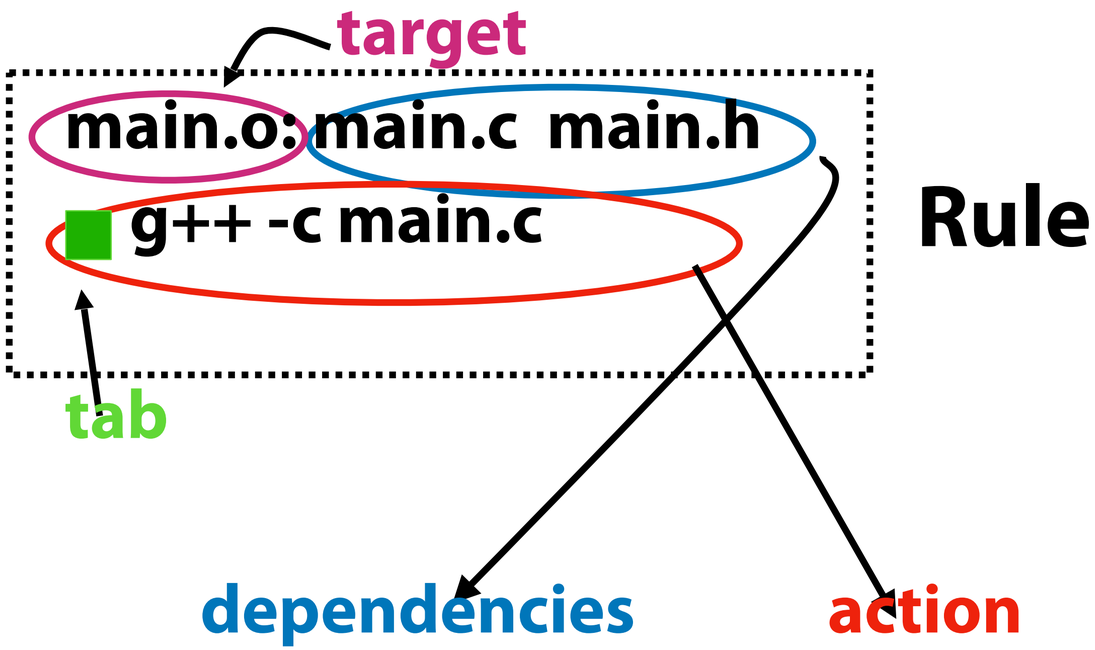
How to write a Makefile -1

1. **Makefile** is designed to allow programmers to efficiently compile large complex programs
2. Placing the commands to compile a program in a Unix script will cause ALL modules to be compiled every time. The “make” utility compile only the modules that have changed and the modules that depend upon them.
3. In Unix, when you type the command “make” the operating system looks for a file called either “makefile” or “Makefile”.
4. This file contains a series of directives that tell the “make” utility how to compile your program and in what order.
5. Each file will be associated with a list of other files by which it is dependent. This is called a **dependency line**.

How to write a Makefile -2

- A **comment** is indicated by the character "#". All text that appears after it will be ignored by the make utility until the end of line is detected. Comments can start anywhere.

- **Rules** tell make when and how to make a file. The format is as follows: a rule must have a dependency line and may have an action or shell line after it. The action line is executed if the dependency line is out of date.



How to write a Makefile -3

Let's suppose we have the following files

- main.c: the main program file
- lib.c: library file
- lib.h: header file

```
% gcc -c main.c -o main.o
% gcc -c lib.c -o lib.o
% gcc lib.o main.o -o exec
```

Useful flags

- -g: debugging
- -Wall: all warnings
- -Werror: treat every warnings as errors
- -O2, -O3: optimization

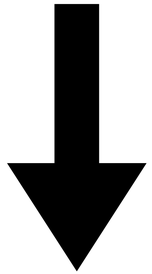
Examples

```
% gcc -g -Wall -Werror -c main.c -o main.o
% gcc -g -Wall -Werror -c lib.c -o lib.o
% gcc -g -Wall -Werror lib.o main.o -o binary
```

How to write a Makefile -4

target: dependency1 dependency2 ...
    unix command (start line with TAB)
    unix command

```
% gcc lib.o main.o -o binary
```



```
binary: lib.o main.o
    gcc lib.o main.o -o binary
```

How to write a Makefile -5

- Macro is a shorthand or alias used in the makefile
- Inside the file, to expand a macro, you have to place the string inside of `$(  )`. The whole thing is expanded during execution of the make utility.

Examples of macros:

- `HOME = /home/finelli`
- `CPP = $(HOME)/cpp`

How to write a Makefile -6

```
binary: lib.o main.o
    gcc -g -Wall lib.o main.o -o binary

lib.o: lib.c
    gcc -g -Wall -c lib.c -o lib.o

main.o: main.c
    gcc -g -Wall -c main.c -o main.o

clean:
    rm *.o binary
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
```

```
$(OUTPUT): lib.o main.o
    $(CC) $(CFLAGS) lib.o main.o -o binary

lib.o: lib.c
    $(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
    $(CC) $(CFLAGS) -c main.c -o main.o

clean:
    rm *.o $(OUTPUT)
```

How to write a Makefile -7

```
$(OUTPUT): $(OBJFILES)
    $(CC) $(CFLAGS) $(OBJFILES) -o binary

lib.o: lib.c
    $(CC) $(CFLAGS) -c lib.c -o lib.o

main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o

clean:
    rm *.o $(OUTPUT)
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o
```

How to write a Makefile -8

- **Inference rules** are a method of generalizing the build process. The symbol “%” is used.
- All .obj files have dependencies of all %.c files of the same name.

```
%.obj : %.c
        $(CC) $(FLAGS) -c $(.SOURCE)
```

How to write a Makefile -9

- Simple conditional statements can be included in a makefile. Usual syntax is:

```
ifeq (data1,data2)
  body of if
else
  body of else
endif
```

Note that assigning a value to a variable within the makefile overrides any value passed from the command line.

The GNU debugger -1

Standard compilation

```
gcc [flags] <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors main.f90 -o run
```

If you want to enable debugging, you must add a -g option

```
gcc [other flags] -g <source files> -o <output file>
```

For example:

```
gcc -Wall -Werror -ansi -pedantic-errors -g main.f90 -o
run
```

The GNU debugger -2

Launch “gdb” and you’ll get a prompt that looks like this:

```
(gdb)
```

To specify a program to debug, you’ll have to load it:

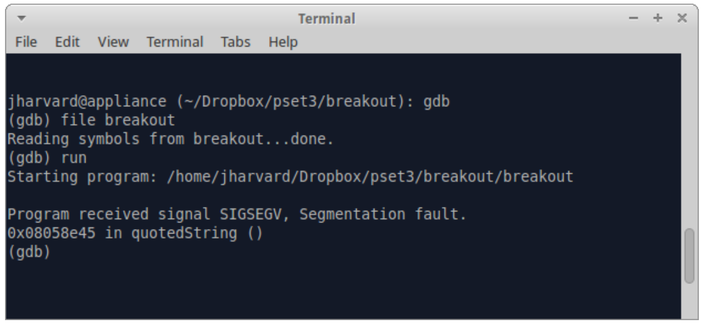
```
(gdb) file main.f90
```

Here, main.f90 is the program you want to load, and “file” is the command to load it. To run the program, just use:

```
(gdb) run
```

If it has no serious problems (i.e. the normal program didn’t get a segmentation fault, etc.), the program should run fine here too.

If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:



Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break."

This sets a breakpoint at a specified file-line pair:

```
(gdb) break main.f90:110
```

This sets a breakpoint at line 110, of main.f90. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

You can also tell gdb to break at a particular function. Suppose you have a function `my_func`:

```
int my_func(int a, char *b);
```

You can break anytime this function is called:

```
(gdb) break my func
```

Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same break command as before:

```
(gdb) break main.f90:110 if (condition)
```

The GNU debugger -4

You can proceed onto the next breakpoint by typing "continue"

```
(gdb) continue
```

You can single-step (execute just the next line of code) by typing "step."

This gives you really fine-grained control over how the program proceeds.

```
(gdb) step
```

Similar to "step," the "next" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.

```
(gdb) next
```

The GNU debugger -5

The print command prints the value of the variable specified, and print/x prints the value in hexadecimal:

```
(gdb) print my var
(gdb) print/x my var
```

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

```
(gdb) watch my var
```

Now, whenever my var's value is modified, the program will interrupt and print out the old and new values.

Use help to learn more commands!!