Computer ‘Arithmetic’
Integer Arithmetic

- Fortran does this ok, just need to watch out for numbers (including intermediate results) too large/small to express. Operations then lead to wrapping.

- Sometimes 32 bit integers are not enough

  \[
  E = B \times D = 25000000 \quad \Leftarrow \text{Right}
  \]

  \[
  A = E \times C = 445948416 \quad \text{Wrong}
  \]

  \[
  \text{print*}, A/C = 89189 \quad \Leftarrow \text{But this is } E!
  \]

- A sanity test for sufficient integer range: see int_sanity.f90

  \[
  n_{\text{total}} = n \times n \times n
  \]

  if \( n_{\text{total}} /\div n \times \text{INT}(n,8) \times n \) call panic(…)
Floating point Arithmetic

- **Floating point** numbers are a finite subset of the rational numbers. They are bounded and have the same frequency per decade.

- In true arithmetic operations with rational numbers are also rational numbers.

- Beside the issue of boundedness, in FP arithmetic, an operation with FP numbers may only approximate an FP number - rounded or truncated.
Floating point Exceptions

• Beside yielding approximate results, floating point operations can yield:

• **Overflow**, typically output as Inf - beyond the range of the fp numbers

• **Underflow** - closer to zero than representable, usually reset to zero. Watch for division by an underflow!!!

• Not mathematically permissible: log(0), sqrt(-1.) Usually output as NaN (not a number)

See exceptions.f90
FP Arithmetic - more consequences

Neither associative nor distributive:
(A+B)+C may not be A+(B+C) (ditto for *)
(A+B)--B may not be A (ditto for * and / )
A+A+A may not be 3.0*A

They do not have a multiplicative inverse: Not all A have a B = 1.0/A, such that A*B = 1.0

• Not continuous (for any of +, --, *, or /):
B > 0.0 may not mean A+B > A
A > B and C > D may not mean A+C > B+D
A > 0.0 may not mean 0.5*A > 0.0

See var.f90, sum_order.f90

A poorly-designed algorithm can be vulnerable to the vagaries of floating point arithmetic. Worthwhile to use good third-party libraries.
FP Error Analysis

Let
\[ x_{fp} = x \times (1 + O(\varepsilon)) \]
\[ y_{fp} = y \times (1 + O(\varepsilon)) \]

Then
\[ x_{fp} \times y_{fp} \approx x \times y \times (1 + O(\varepsilon)) \]
\[ x_{fp} + y_{fp} \approx (x + y) \times (1 + O(\varepsilon)) \]

In addition, if \( x \) and \( y \) are of opposite signs, \( x + y \) can be considerably smaller than either \( x \) or \( y \), and the result is a larger relative error.
Three kinds of programming inefficiencies:

- a computation-bound program
- a memory-bound program
- an IO-bound program
Efficient Computation

• AVOID DIVISION! It takes multiple clock cycles. If you divide by the same number frequently, compute and save the reciprocal.

• Most CPUs can do an addition and multiplication simultaneously - write code that tends to pair them

\[ a(i) = a(i) + b(i) \times c(i) \]
Compiler Optimization

- Compilers have flags to perform optimization automatically, typically -O, -O2, -O3
- Compilers often have other flags to do faster but more approximate arithmetic.

Original loop:
```plaintext
do i = 1, ni
  a(i) = a(i) + b(i)*c(i)
enddo
```

Unrolled loop:
```plaintext
do i = 1, ni, 4
  a(i) = a(i) + b(i)*c(i)
  a(i+1) = a(i+1) + b(i+1)*c(i+1)
  a(i+2) = a(i+2) + b(i+2)*c(i+2)
  a(i+3) = a(i+3) + b(i+3)*c(i+3)
enddo
```

- Compiler optimization carries risk! Make sure your program gives (approximately) the same answer with and without optimization
Subroutine Bottlenecks

- Subroutine calls use lots of clock cycles which increase with the number of arguments. You want to maximize computation per subroutine call.
- **Inlining** is an effective way to optimize this. There are usually compiler tools to do this.
- Passing array subsections into subroutines may require a physical copy before any work is done.
Efficient Memory

- Memory efficiency is typically about managing cache use.

- Avoid long strides - they tend to force the program out of cache more frequently.

BAD:
DO i = 1, ni
  DO j = 1,nj
    a(i,j) = a(i,j) + b(i,j)*c(i,j)
  ENDDO
ENDDO

GOOD:
DO j = 1, nj
  DO i = 1,ni
    a(i,j) = a(i,j) + b(i,j)*c(i,j)
  ENDDO
ENDDO
IO Management

• IO bandwidth is much smaller than memory bandwidth.

• Access to local disks is faster than to remote filesystems.

• Unformatted IO avoids conversion of data and is faster than formatted IO.

• There are vendor specific techniques to overlap IO and computation - asynchronous IO.
Where to Optimize? Profiling

- Profiling directs the programmer where to focus optimization efforts - work on the piece of code that consumes most of the time (wall-clock).

- On Mac OSX, use Shark, no special compilation necessary.

- Other platforms may have proprietary profilers, often need to use special compile flags. Look for prof or gprof.

- Other performance monitors tell you other metrics like flop-rate, cache usage, etc.
You’ve written your program, it compiles. Now you run it and it either crashes or gives bad output. You typically need to find out two things - where the error happens, and the variable values at that point.

Types of errors:

- **Floating point exception** - produces a number that is not a valid floating point number. To stop at an exception you need to enable floating point trapping.
- **Segmentation fault (Sigsegv) or Bus error** - often involve a memory error like a bad subscript or argument, often cryptic.
Simple Debugging

• Judicious use of write statements can help pinpoint where the program crashed and provide variable values.

• Every time you add write statements you must recompile. Also, an executable with write statements is not quite the one that blew up. Still, useful if you have a hunch where the problem is.

• Sanity checking - good program practice that tells you things are going wrong before they blow up.

```cpp
if (speed > 0.0 .and. speed < 3.0e8) then
    continue
else
    call panic(’Speed error in my-function’)
endif
```
Many compilers have options to provide extra error checking both during the compilation stage and during run-time. These do incur overhead and should be turned off when debugging is done.

- **Array bounds checking**: detects if an array subscript is out of bounds.

- **Uninitialized variable initialization** - all variables initialized with an **FP exception**. Together with **FP trapping** this lets you detect if you are using a variable before it is assigned a value.

- **Actual option syntax** is compiler-specific.
A program that terminates abnormally often generates a core file. The core file is a snapshot of the program contents at the time of crash. It includes a traceback (which program statement you are at) and the values of each variable.

Some compilers generate the traceback apart from the core and write to standard output.

There are debugger utilities to examine the core file. To use these you need to generate symbol tables, -g, in the object (.o) files, -c. Usually best to turn off all optimization as well.
Command line debuggers

- There are a host of similar command line debuggers that depend on the compiler: dbx, gdb, pgdbg, idb.

- To examine a core file:
  - `gdb executable_file corefile`; then at prompt type `where`.

- Examine values of variables:
  - `print i; print x(3,3)`
To see what the program is like before the crash you can run it to a breakpoint: `stop line_number`

- `gdb executable_file`
- `stop line_number`
- `r(un)`

- **Execution commands:**
  - `r` (run from beginning)
  - `c` (continue)
  - `s` (step one line)
  - `n` (next line)

- **Examine source code:**
  - `l` (list 10 lines)
  - `l line_number`; `l filename.f90`
  - `u` (up one program level)
  - `d` (down one program level)
Graphical Debuggers

- Graphical interface debuggers greatly enhance debugging power. Examples are **Totalview** (widely available) and **DDT** (at NERSC).