Embedded computing for scientific and industrial imaging applications

Lecture 8 - Computer arithmetic, optimization flags

Outline

- More about computer arithmetic
- Optimization and compiler flags

Floating point real numbers

Base 10 scientific notation:

$0.2345\text{e-}18 = 0.2345 \times 10^{-18} = 0.0000000000000000002345$

Mantissa: 0.2345, Exponent: −18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: −11011 means:

$$\begin{aligned} 0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\ &= 0.703125 \text{ (base 10)} \\ -11011 &= -1(2^{4}) + 1(2^{3}) + 0(2^{2}) + 1(2^{1}) + 1(2^{0}) \\ &= -27 \text{ (base 10)} \end{aligned}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

Floating point real numbers

C/C++:

float (kind=4): 4 bytes

This used to be standard single precision

double (kind=8): 8 bytes

This used to be called double precision real

MATLAB default datatype is 8 bytes.

8 bytes = 64 bits,

53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes).

We can store 52 binary bits of precision.

$2^{-52} \backsimeq 2.2 \times 10^{-16} \Longrightarrow$ roughly 15 digits of precision.

Floating point numbers

Since $2^{-52} \approx 2.2 \times 10^{-16}$ ⇒ roughly 15 decimal digits of precision.

We can hope to get at most 15 correct digits in computations.
For example:

```
    printf("pi = %.16f\n", M_PI);
    printf("100 * pi = %.16f\n", 1000.0 * M_PI);

    pi = 3.1415926535897931
    1000 * pi = 3141.5926535897929170
```

Note: storage and arithmetic is done in base 2 Converted to base 10 only when printed!

Absolute and relative error

Let $z\hat{}$ = exact answer to some problem,

z^* = computed answer using some algorithm.

Absolute error: $|z^* - z\hat{}|$

Relative error: $|z^* - z\hat{}| / |z\hat{}|$

If $|z\hat{}| \approx 1$ these are roughly the same.

But in general relative error is a better measure of how many correct digits in the answer:

Relative error $\approx 10^{-k} \Rightarrow \approx k$ correct digits.

Precision of floating point

If x a real number then $fl(x)$ represents the closest floating point number.

Unless overflow or underflow occurs, this generally has relative error

$$|(fl(x) - x) / x| \leq \varepsilon_m$$

where ε_m is Machine epsilon.

$\varepsilon_m \approx 10^{-k} \Rightarrow$ about k correct digits.

8-byte double precision: $\varepsilon^m \approx 2.22 \times 10^{-16}$.

Machine epsilon (for 8 byte reals)

```
double y = 1. + 3.e-16;
printf("y = %.16f\n", y); printf("y - 1.0 = %e\n", y - 1.0);
```

```
y = 1.0000000000000002
y - 1.0 = 2.220446e-16
```

Machine epsilon is the distance between 1.0 and the next largest number that can be represented:

$2^{-52} \approx 2.2204 \times 10^{-16}$

```
y = 1.0 + 1e-16;
printf("y = %.16f\n", y);
printf("%d\n", y == 1.0);
```

```
y - 1.0 = 2.220446e-16
y = 1.0000000000000000
1
```

Check `$CLASS_REPO/codes/08_codes`

Catastrophic cancellation of nearly equal numbers

We generally don’t need 16 digits in our solutions
But often need that many digits to get reliable results.

```
printf("pi = %.16f\n", M_PI);
y = M_PI * 1.e-10;
printf("y = %.16f\n", y);
```

```
pi = 3.1415926535897931
y = 3.141593e-10
```

```
double z = 1.0 + y;
printf("z = %.16f\n", z);
printf("z - 1 = %e\n", z - 1.0);
```

```
z = 1.0000000003141594
```

# 15 digits correct in z

```
z - 1 = 3.141594e-10
```

# only 6 or 7 digits right!

Check `$CLASS_REPO/codes/08_codes`

Sample compiler optimizations

Visual studio C/C++ Compiler options.

/O Options (Optimize Code)

https://msdn.microsoft.com/en-us/library/k1ack8f1.aspx

GNU gcc compiler options

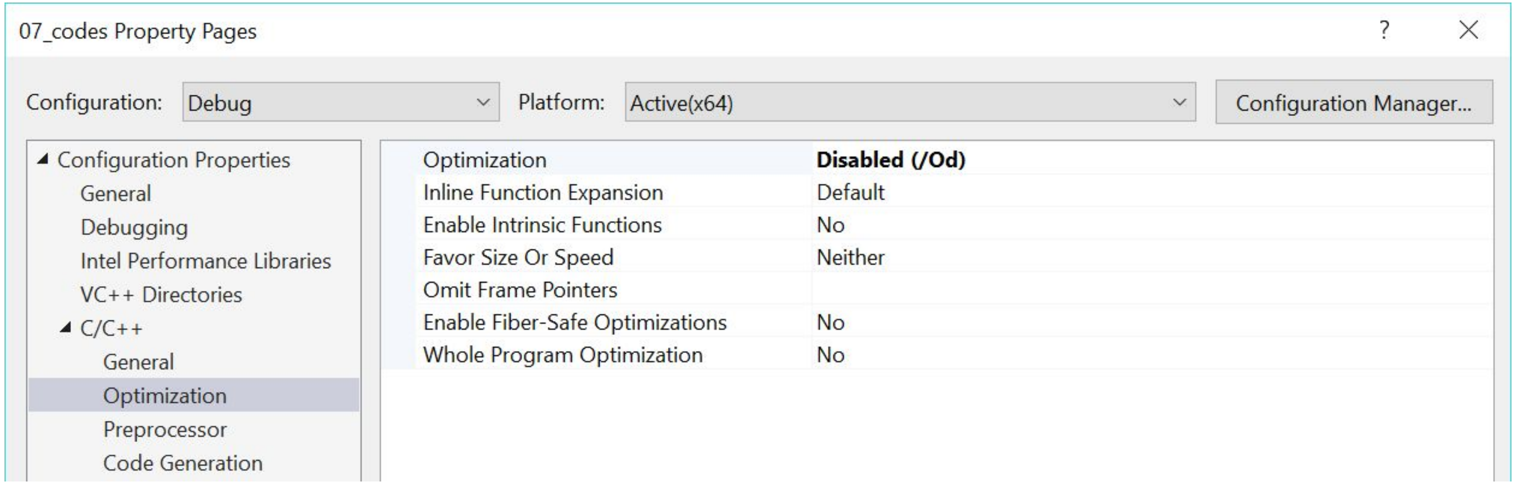
http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/ Optimize-Options.html

for a list of many gcc optimization flags.

/O Options (Optimize Code)

- /O2 optimizes code for maximum speed. (Release configuration)
- /Od disables optimization, speeding compilation and simplifying debugging. (Debug configuration)

You can find in "Project properties" => C/C++ => Optimization"



Optimization	**Disabled (/Od)**
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Favor Size Or Speed	Neither
Omit Frame Pointers	
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	No

Manual code optimization

Often it is necessary to rethink the algorithm in order to optimize code.

“Premature optimization is the root of all evil” (Don Knuth)

Once code is working, determine which parts of code need to be improved and spend effort on these sections.

Use tools such as Visual C++ PGO(profile-guided optimization), Intel® VTune™ or gprof to identify bottlenecks.