Advanced Programming in Quantitative Economics

Introduction, structure, and advanced programming techniques

Charles S. Bos, Henning Bunzel

VU University Amsterdam
Tinbergen Institute

c.s.bos@vu.nl

15 – 19 August 2011, Aarhus, Denmark

Outline

Day 2 - Afternoon

13.00L Background of computations

- Floating point numbers and rounding errors
- Compilers and CPUs
- Computing environment at Aarhus University

14.30P Tutorial

- Simulate data duration model
- Apply concepts of the day
- Think of rounding errors

16.00 End

Precision

Not all numbers are made equal...
Example: What is $1/3 + 1/3 + 1/3 + ...$?

Listing 1: precision/onethird.ox

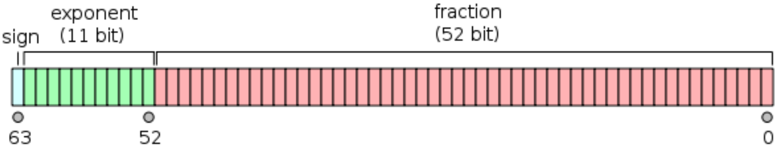
```
main()
{
  decl i, j, dD, dSum;

  dD= 1/3;
  dSum= 0.0;
  for (i= 0; i < 10; ++i)
    for (j= 0; j < 3; ++j)
      {
        print (dSum~i~(dSum-i));
        dSum+= dD;  // Successively add a third
      }
}
```

See outcome: It starts going wrong after 16 digits...

Representation

- Integers are represented exactly using 4 bytes/32 bits, in range `[INT_MIN, INT_MAX]= [-2147483648, 2147483647]`
- Doubles are represented in 64 bits. This gives a total of $2^{64} \approx 1.84467 \times 10^{19}$ different numbers that can be represented.



Double floating point format (Graph source: Wikipedia)

Split double in

- Sign (one bit)
- Exponent (11 bits)
- Fraction or mantissa (52 bits)

Double representation

$$x = \begin{cases} (-1)^{\text{sign}} \times 2^{1-1023} \times 0.\text{mantissa} & \text{if exponent=0x.000} \\ (-1)^{\text{sign}} \times \infty & \text{if exponent=0x.7ff} \\ (-1)^{\text{sign}} \times 2^{\text{exponent}-1023} \times 1.\text{mantissa} & \text{else} \end{cases}$$

Note: Base-2 arithmetic

Sign	Expon	Mantissa	Result
0	0x.3ff	0000 0000 0000	$-1^0 \times 2^{(1023-1023)} \times 1.0$ $= 0$
0	0x.3ff	0000 0000 0001	$-1^0 \times 2^{(1023-1023)} \times 1.000000000000000222$ $= 1.000000000000000222$
0	0x.400	0000 0000 0000	$-1^0 \times 2^{(1024-1023)} \times 1.0$ $= 2$
0	0x.400	0000 0000 0001	$-1^0 \times 2^{(1024-1023)} \times 1.000000000000000222$ $= 2.000000000000000444$

Bit weird...

Consequence: Addition

Let's work in Base-10 arithmetic, assuming 4 significant digits:

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	3	0.5670	0.5670×10^3	567

What is the sum?

Consequence: Addition

Let's work in Base-10 arithmetic, assuming 4 significant digits:

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	3	0.5670	0.5670×10^3	567

What is the sum?

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	4	0.0567	0.0567×10^4	567
$+$	4	0.1801	0.1801×10^4	1801

Shift to same exponent, add mantissas, perfect

Consequence: Addition II

Let's use dissimilar numbers:

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	1	0.5670	0.5670×10^1	5.67

What is the sum?

Consequence: Addition II

Let's use dissimilar numbers:

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	1	0.5670	0.5670×10^1	5.67

What is the sum?

Sign	Exponent	Mantissa	Result	x
$+$	4	0.1234	0.1234×10^4	1234
$+$	4	$0.0005\cancel{67}$	0.05×10^4	5
$+$	4	0.1239	0.1239×10^4	1239

Shift to same exponent, add mantissas, loose precision...

Further consequence:

Add numbers of similar size together, preferably!

In Ox/C/Java/Matlab/Octave/Gauss: 16 digits ($\approx$ 52 bits) available instead of 4 here

Consequence: Addition III

Check what happens in practice:

Listing 2: precision/accuracy.ox

```
main()
{
  decl dA, dB, dC;

  dA= 0.123456 * 10^0;
  dB= 0.471132 * 10^15;
  dC= -dB;

  println ("a: ", dA, ", b: ", dB, ", c: ", dC);
  println ("a + b + c: ", dA+dB+dC);
  println ("a + (b + c): ", dA+(dB+dC));
  println ("(a + b) + c: ", (dA+dB)+dC);
}
```

Consequence: Addition III

Check what happens in practice:

Listing 3: precision/accuracy.ox

```
main()
{
  decl dA, dB, dC;

  dA= 0.123456 * 10^0;
  dB= 0.471132 * 10^15;
  dC= -dB;

  println ("a: ", dA, ", b: ", dB, ", c: ", dC);
  println ("a + b + c: ", dA+dB+dC);
  println ("a + (b + c): ", dA+(dB+dC));
  println ("(a + b) + c: ", (dA+dB)+dC);
}
```

results in

```
Ox Professional version 6.00 (Linux_64/MT) (C) J.A. Doornik, 1994-2009
a: 0.123456, b: 4.71132e+14, c: -4.71132e+14
a + b + c: 0.125
a + (b + c): 0.123456
(a + b) + c: 0.125
```

Other hints

- Adding/subtracting tends to be better than multiplying
- Hence, log-likelihood $\sum \log \mathcal{L}_i$ is better than likelihood $\prod \mathcal{L}_i$
- Use true integers when possible
- Simplify your equations, minimize number of operations
- Don't do $x = \exp(\log(z))$ if you can escape it

Other hints

- Adding/subtracting tends to be better than multiplying
- Hence, log-likelihood $\sum \log \mathcal{L}_i$ is better than likelihood $\prod \mathcal{L}_i$
- Use true integers when possible
- Simplify your equations, minimize number of operations
- Don't do $x = \exp(\log(z))$ if you can escape it

(Now forget this list... use your brains, just remember that a computer is not exact...)

On architecture, algorithms, languages, and machines

Why do we program (repeat)?

> *To get to the results we need, in a fashion that is controllable, where we are free to implement the newest and greatest, and where we can be 'reasonably' sure of the answers*

What is important here?

1. To get *correct* code ($\approx$ maintainable, clear, adjustable)
2. To get *efficient* code ($\approx$ quick?)

Correct code: See rest of course

Efficiency II

Efficient code: Depends on

1. system (processor, memory size/structure, drives, network, operating system)
2. language
3. algorithm
4. more...?

Use an example:

What is the sum of all returns on the SP500 stock index, over 20 years?

$$S = \sum_{t=1}^{N} r_t$$

Efficiency: System

What do you prefer:

1. Old Apple II (1Mhz, 64KB)
2. Older Power Mac (2.5GHz, PowerPC processor)
3. This laptop (C2Duo 1.4Ghz, 1GB)
4. Home machine (I7 quadcore 2.66Ghz, 6GB)
5. Work machine (Nehalem dual quadcore 2.66Ghz, 12GB)
6. SARA Lisa supercomputer (536 dual quadcores, loads of memory)

Difference?

Efficiency: System II

Difference:

1. Apple does not even have the memory to store the data. Read from tape?
2. Mac: PowerPC had low-level possibilities to sum vectors of numbers quickly.
3. C2D laptop: Not bad, fraction of a second
4. Core I7: Better memory management in CPU, quicker
5. 2xI7: Can one use 8 cores for the summation?
6. SARA: Can one use 536 x 2 x 4 cores for summing 1000 numbers?

In general: Recent $\equiv$ better... More memory $\equiv$ better... More cores/CPUs $\not\equiv$ better *necessarily*

CPU

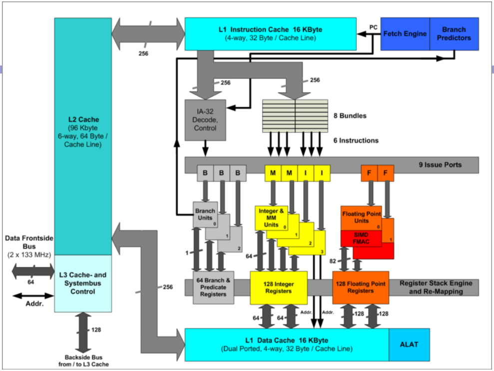


Figure: Itanium CPU architecture (Source: Wikipedia)

- Hard drive → memory → CPU
- Efficiency differs per CPU architecture
- Does your data fit in memory/cache? Or should you go back to hard drive to reload it
- Communication between parts of system: Speed matters
- ...

Multi-core?

$$S = \sum_{t=1}^{N} r_t = \sum_{t=1}^{N/4} r_t + \sum_{t=N/4+1}^{N/2} r_t + \sum_{t=N/2+1}^{3N/4} r_t + \sum_{t=3N/4}^{N} r_t$$
$$= S_1 + S_2 + S_3 + S_4$$

Use 4 cores for summation?

1. Decide on how to split the problem
2. Send each core the right amount of data
3. Wait for the answers to come back
4. Sum $S = S_1 + S_2 + S_3 + S_4$

Drawback: Overhead in communication between cores/nodes, not always faster...
(GPU, supercomputer, MPI, ...)

Algorithm

```
int i, iN;       // C code
double dS;

iN= ...;
dS= 0.0;
for (i= 0; i < iN; ++i)
  dS+= vR[0][i];
```

```
C FORTRAN CODE
       INTEGER i, iN
       REAL dS

       iN= ...
       dS= 0.0
       DO 10 i= 1, iN
         dS= dS + vR(0,i)
  10   CONTINUE
```

```
decl i, dS, iN; // Ox code

iN= sizerc(vR);
dS= 0;
for (i= 0; i < iN; ++i)
  dS+= vR[i];
```

```
decl dS;
// Ox code, using function

dS= sumc(vR);
```

- Fortran or C would be quickest, as compiled code can be executed directly by CPU
- Ox code is similar to C, but far slower: Each command has to be ‘translated’ to executable code, so even a loop takes time...
- Ox version using `sumc` is virtually as fast as Fortran or C

Algorithm II

Think...

Where do returns come from? First difference of log prices

$$p_t = \log P_t$$
$$r_t = p_t - p_{t-1}$$

What does this imply for the sum of the returns?

Algorithm II

Think...

Where do returns come from? First difference of log prices

$$p_t = \log P_t$$
$$r_t = p_t - p_{t-1}$$

What does this imply for the sum of the returns?

$$\begin{aligned} S = \sum r_t &= \sum_t (p_t - p_{t-1}) \\ &= (p_1 - p_0) + (p_2 - p_1) + \cdots + (p_N - p_{N-1}) = p_N - p_0 \end{aligned}$$

- Quicker algorithm
- This does more than quicker computer, quicker language...
- Think/measure where your program is slow: Work on bottlenecks first

Algorithm III

Good algorithms: Where to find?

- BLAS library (C/Fortran)
- LAPack library (C/Fortran)
- Within higher-level languages (Matlab/Octave/Ox/Gauss)

Examples: Regression, matrix inversion, random number generator, optimization... Easy to code little robustly, hard to get right...

Low-level 'algorithms': Operators

Not all operators are equal:

Speed	**Operation**	**Example**
Fast	Integer addition and subtraction	`I+J; I-J`
	Floating point addition and subtraction	`A+B; A-B`
	Int multiply, fl. point multiply/divide	`I*J, A*B, A/B`
⋮	Integer divide	`I/J`
	Exponentiation to a positive int. constant	`A^2`
	Exponentiation to a positive int. variable	`A^I`
Slow	Exponentiation to a floating point variable	`A^B`

What is the difference?

```
for (dI= 0.0; dI < iN; ++dI)
  dosomething();
```

```
for (i= 0; i < iN; ++i)
  dosomething();
```

Operators, do's and don'ts

Looking *only* at memory use and speed of operators:

Don't	**Do**
`A/2.0`	`0.5*A`
`A^2`	`A * A`
`B (E + F) - C*(E+F)`	`(B - C) * (E + F)`
`A^0.5`	`sqrt(A)`
`B/C/D/E`	`B/(C*D*E)`
`T1 = B + C, T2 = D*E, A = T1 + T2`	`A= B+C+D*E`
`T1 = X + Y, A= T1 + log(T1)`	`A= (X + Y) + log(X + Y)`

Note: Think also of clarity of your program...

Loops and execution: Gather

Loops take most time (even empty loops take time!)
Preferably use one loop, gather statements together

```
// Don't
dS= dQ= 0;
for (i= 0; i < iN; ++i)
  dS+= vR1[i];
for (i= 0; i < iN; ++i)
  dQ+= vR2[i];
```

```
// Do
dS= dQ= 0;
for (i= 0; i < iN; ++i)
  {
    dS+= vR1[i];
    dQ+= vR2[i];
  }
```

Loops and execution: If-then-else

Conditional constructs are hard for CPU, take if-then-else outside of loop if possible

```
// Don't
dS= 0;
for (i= 0; i < iN; ++i)
  if (iDay == 4)
    dS+= vR1[i];
  else
    dS+= vR2[i];
```

```
// Do
dS= 0;
if (iDay == 4)
  for (i= 0; i < iN; ++i)
    dS+= vR1[i];
else
  for (i= 0; i < iN; ++i)
    dS+= vR2[i];
```

If-then-else

Test for most common condition first

```
// Don't
if (iDay == 4)
  dS= sumr(vR1);
else
  dS= sumr(vR2);
```

```
// Do
if (iDay != 4)
  dS= sumr(vR2);
else
  dS= sumr(vR1);
```

Memory and speed

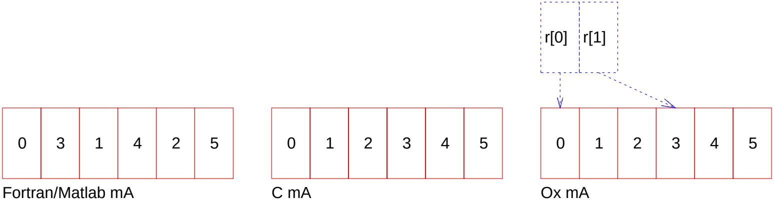
Should you care about memory?

Memory and speed

Should you care about memory? Yes and no... Make some speed difference.

Matrices $A = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$ is stored:

- Column-wise: Fortran, Matlab
- Row-wise: C
- Row-wise (with row pointers): Ox



Function calls

When you call a function:

- Local variables declare their memory for every function call. Can be expensive, for many/large variables. For speed: Use globals...
- It takes time: Putting small procedures in-line is quicker
- the passing of parameters also takes time. Passing pointers to variables, or using globals, is again faster.
- In general, using memory takes time (in C: new). Think about the constructs you need.
- Pointers are faster than copying data

but:

Forget this for now...

Structure!

Your time is more valuable than computer time...

Concentrate on *structured, readable* code.
Afterwards, with *bug-free* code

- profile your code to see which routine takes most time
- think of putting mostly used data constructs global
- aforementioned tricks can shave 5-20% off execution time. Better algorithm can improve infinitely more...
- move from laptop to recent desktop, think of using MPI, split program in smaller tasks, etc.