Computational Economics: Practical Tools and Techniques
Part 1: Overview of scientific computing

Victor V. Zhorin

Computation Institute/BFI

Nov 13, 2013

BECKER FRIEDMAN INSTITUTE
FOR RESEARCH IN ECONOMICS
THE UNIVERSITY OF CHICAGO

Historical perspectives on computers and economics

- Odhner Arithmometer (1874): 9 digit precision for operands, 13 digits for results
- EDSAC (UK, 1949-1958):17 bits (one word) or 35 bits (two words) long operands and results, ˜220 multiplications per second, 12kW, weight 6t (ENIAC, US, 1946-1955: 10 bits operands and results, ˜400 multiplications per second, 160 kW, 27 t) - the first article by Hendrick Houthakker published in econometrics based on results from a digital calculator having a stored memory program. "Some Calculations on Electricity Consumption in Great Britain", Journal of Royal Statistical Society, 1951.
- Orcutt's portable regression analyzer (1945), 2 digit accuracy, 33 items in storage on punch cards, "A Study of the Autoregressive Nature of the Time Series Used for Tinbergen's Model of the Economic System of the United States, 1919-1932", Journal of Royal Statistical Society, 1948.

Barriers and costs

- "Ninja gap" - performance gap between naively written C/C++ code that is parallelism unaware and best-optimized code on modern multi-/many-core processors
- There is an average Ninja gap of 24X (up to 53X) for a recent 6-core Intel Core i7 X980 Westmere CPU, and that this gap if left unaddressed will inevitably increase (Nadathur Satish at al, Computer Architecture (ISCA), 2012)
- Computational complexity and numerical algorithms complexity: complementary or substitute?
- Massively parallel processors favor **simpler** numerical algorithms: larger block-independent data frames with optimal function calls versus sophisticated (expensive) serial code

Round-off (trivial) errors (logistic functions based models)

MATLAB (standard double data type)
There is no quadruple precision available in standard MATLAB distributions

```
1e15*sum([-5, 1e-15, 5])
ans = 0.8882

1e15*sum([-15, 1e-15,15])
ans = 1.7764

eps(-15)+eps(1e-15)+eps(15)
ans = 3.5527e-15
```

"Big Data"?

,
2012: Really Big Objects Coming to R: matrices were limited 46,000 by 46,000 before.
Symbolic computing in MATLAB and Mathematica
ADVANPIX Multiprecision Computing Toolbox for MATLAB - highly optimized for quadruple precision

```
tic; [U,S,V] = svd(Y); toc;        % MATLAB Symbolic Math Toolbox
Elapsed time is 59.357723 seconds.
norm(Y-U*S*V',1); 3.061145275981987186191661e-31
tic; [U,S,V] = svd(X); toc;        % Multiprecision Computing Toolbox
Elapsed time is 0.668525 seconds.    % ~80 times faster
norm(X-U*S*V',1);  5.718086004885703247723198511534649e-31
```

Complex Step Derivative: First Derivative Approximations (Euler equation)

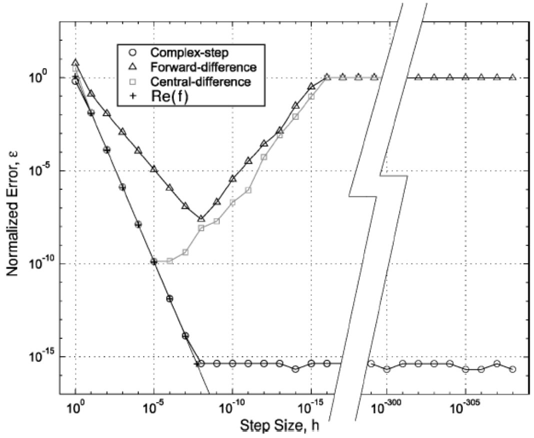
"It is disappointing, certainly, that **no simple finite-difference formula** gives accuracy comparable to the machine accuracy or even the lower accuracy to which f[unction] is evaluated" –William Press, Numerical Recipes in C++

- $f'(x) = \frac{f(x+h)-f(x)}{h} + \mathcal{O}(h)$, truncation error is $\mathcal{O}(h)$
- Truncation error vs. subtractive cancellation (loss of significance)

```
%MATLAB
x=2; h=10^-1; ((x+h)^3-x^3)/h
ans =   12.6100;
x=2; h=10^-15; ((x+h)^3-x^3)/h
ans =    10.6581;
```

- Taylor series for real function extended into the complex plane $f(x+\imath h) = f(x) + \imath h \frac{\partial f}{\partial x} - ...$
- $$\frac{\partial f}{\partial x} \approx \frac{\mathrm{Im}[f(x+\imath h)]}{h}$$
- Error is $O(h^2)$

Complex Step Derivative: Asymptotics



From: J. R. R. A. Martins, P. Sturdza and J. J. Alonso, "The Complex-Step Derivative Approximation". ACM Transactions on Mathematical Software, Vol. 29, No. 3, September 2003, Pages 245-262.

Complex Step Derivative: Implementation in MATLAB using OOP

```
% Example
%tic;h=@(x) 2*pi*x.^3 + 5*x.^2 + 3*x;d=CSD;nx=100000;x=linspace(-100,100,nx);tt=
      zeros(1,nx);tt=d.gradient(h,x);toc; Elapsed time is 14.981273 seconds.
classdef CSD < handle
    properties
        def_step = 1e-9;
    end
    methods
         function grad = gradient(this, h, x0, step)
            num_var= length( x0 ) ;
            grad = zeros(num_var, 1 ) ;
            if nargin < 4
                dx = this.def_step ;
            else
                dx = step ;
            end
            x_i = x0 + 1i * dx ;
            for i_x = 1 : num_var
                x1 = x0;
                x1( i_x ) = x_i(i_x) ;
                [ fval ] = h( x1(i_x) ) ;
                grad(i_x ) = imag( fval / dx ) ;
            end
        end
    end
end
```

Complex Step Derivative: Implementation in Fortran

```fortran
implicit none
interface
function simple_function(x)
implicit none
complex(8) :: simple_function
complex(8), intent(in)  :: x
end function simple_function
end interface
real(8), parameter :: def_step = 1e-9
real(8), allocatable :: x_real(:), dfdx(:)
real(8) xmin, xmax
complex(8), allocatable :: x_complex(:)
integer nx,i
nx = 100000
xmin = -100
xmax = 100
allocate(x_real(nx), dfdx(nx), x_complex(nx))
do i=1,nx
x_real(i) = xmin + (i-1)*(xmax-xmin)/(nx-1)
x_complex(i) = dcmplx(x_real(i), def_step)
dfdx(i) = dimag(simple_function(x_complex(i))) / def_step
end do
```

Complex Step Derivative: Implementation in Fortran

```fortran
implicit none
interface
function simple_function(x)
implicit none
complex(8) :: simple_function
complex(8), intent(in)  :: x
end function simple_function
end interface
real(8), parameter :: def_step = 1e-9
real(8), allocatable :: x_real(:), dfdx(:)
real(8) xmin, xmax
complex(8), allocatable :: x_complex(:)
integer nx,i
nx = 100000
xmin = -100
xmax = 100
allocate(x_real(nx), dfdx(nx), x_complex(nx))
do i=1,nx
x_real(i) = xmin + (i-1)*(xmax-xmin)/(nx-1)
x_complex(i) = dcmplx(x_real(i), def_step)
dfdx(i) = dimag(simple_function(x_complex(i))) / def_step
end do
```

time ./csd $\rightarrow$0.004s

Speed gain: 3750 times over MATLAB OOP implementation, further gains are possible

pure function-based MATLAB 0.01s

Using Python as glue: Fortran/C to Python

```fortran
function simple_function(x)

implicit none
complex(8) :: simple_function
complex(8), intent(in) :: x

! Declare local constant Pi
complex(8), parameter :: Pi = 3.1415927

simple_function = 2*Pi*x**3 + 5*x**2 + 3*x

end function simple_function
```

```
 f2py -c -m ff simple_function.f90
 python
Python 2.7.5 |Anaconda 1.7.0 (x86_64)
>>> import ff
>>>  ff.simple_function(1)
(14.283185482025146+0j)
```

Scientific Computing versus Computer Science

- One of the other problems we face, with respect to software, is the fact that we haven't changed our programming model in the last thirty years. FORTRAN and C are the programming paradigms that we used, and we still use, and they have no mechanism for doing parallel processing, so all the processing comes about through functional calls that are made to do the message passing. We really need to think about new languages that can effectively exploit parallel processing.
- The chips in our laptops are capable of high performance; for instance, **if we do the right thing with respect to the memory hierarchy we can see really significant speedup over doing it in a naive way**.

An interview with Jack J. Dongarra, SIAM, 2004.

Benchmarks for serial code performance: naive (inefficient) Python

```python
#!/usr/bin/env python
import time

def linspace( start, end, count ):

    delta = (end-start) / float(count)
    return [start,] + \
    map( lambda x:delta*x + start, range( 1, count ) ) + [end, ]

if __name__ == "__main__":
    t = time.clock()
    nx = 10000000

    xmin, ymin = float(0), float(0)
    xmax, ymax = float(1), float(1)

    x = linspace(xmin, xmax, nx)
    y = linspace(ymin, ymax, nx)
    vec_mult=linspace(0,0,nx); # initialize vector

    for i in range(1,nx) :
        vec_mult[i] = x[i]*y[i]
    print sum(vec_mult)/nx
    print time.clock() - t
```

```
0.333333283333 12.832045s
```

Benchmarks for serial code performance: MATLAB versus Python

(Python Enthought Canopy distribution or Anaconda distribution)

The NumPy array: a structure for efficient numerical computation:

A convenient way of describing blocks of computer memory, so that the numbers represented may be easily manipulated - a view over the memory, **strided** memory model

```
#!/usr/bin/env python
import numpy
import time

t = time.clock()
nx = 10000000

xmin, ymin = float(0), float(0)
xmax, ymax = float(1), float(1)
dx = float(xmax-xmin)/(nx-1)
dy = float(ymax-ymin)/(nx-1)
x = numpy.arange(xmin, xmax, dx)
y = numpy.arange(ymin, ymax, dy)
print numpy.dot(x,y)/nx
print time.clock() - t
```

0.33333325 0.113677s **Speed gain: 100 times over naive implementation, further gains are possible**

```
%MATLAB
tic; nx = 10000000;x=linspace(0,1,nx);y=linspace(0,1,nx);disp(sum(x.*y)/nx);toc;
```

0.333333349999975 Elapsed time is 0.560156 seconds.

Math libraries

- Basic Linear Algebra Subprograms - BLAS/LAPACK
- Intel MKL: industry-standard, interfaces to C and Fortran; "Intel MKL is indispensable for any high-performance computer user on x86 platforms." Jack Dongarra
- OpenBLAS - multi-threaded high performance implementation for multi-core CPUs
- MacOS X: Accelerate framework
- MATLAB with Intel MKL $\Rightarrow$ 5-6 times faster than open source Fortran/BLAS
- The GNU Scientific Library (GSL) - a C replacement for numerical procedures written in Fortran (Netlib), NO high performance BLAS
- EIGEN templates and Armadillo C++ linear algebra library, the syntax (API) deliberately similar to MATLAB
- IMSL FORTRAN/C Numerical Library

Non-linear Optimization with derivatives: Components, Solvers

- Gradients and Hessians are critical for Newton-based NL solvers
- Solution update method: Sequential Quadratic Programming (SQP), Interior-Point (IP)
- Global optimum: trust region, line search
- Penalty function, tolerance, feasibility
- SNOPT

  line-search SQP; null-space CG option
  l_1 exact penalty function
- IPOPT - open source in COIN-OR

  line-search filter algorithm
- KNITRO

  trust-region Newton, interior with CG option or direct
  l_1 exact penalty function
  Active Set - for medium size problems with good initial guess

Non-linear Optimization with derivatives: Tools

AMPL (GAMS) Modeling and Programming Languages

- use automatic differentiation (AD vs. CSD vs. numeric differentiation)
- natural for optimization problems
- mathematical programming
- access to solvers
- portable code
- NEOS server
  (`http://www.neos-server.org/neos/solvers/index.html`) :

  input in GAMS or AMPL
  dozens of solvers available, including KNITRO, SNOPT, MOSEK, IPOPT

Non-linear Optimization with derivatives: Example

Structural model estimation with GMM, GMM (initial) $= 29$
"Constrained optimization approaches to estimation of structural models", C.-L. Su & K. Judd, Econometrica, 2012

```
KNITRO  6.0.0
Number of variables:                    2339
Number of constraints:                  2300
    linear equalities:                    44     nonlinear equalities:
         2256
Number of nonzeros in Jacobian:       133696
Number of nonzeros in Hessian:         60879
...
25 iterations; 50 function evaluations
GMM = 3.72555
 Total program time (secs)          =        12.79959 (12.774 CPU time)
```

```
 Ipopt 3.10.3 (NEOS)
Number of Iterations....: 94 Number of objective function evaluations = 153
GMM = 3.72555
 Total CPU secs in IPOPT (w/o function evaluations)   =      61.193
 Total CPU secs in NLP function evaluations           =     110.733
```

SNOPT 7.2-10 : failed. 26104 iterations, CONOPT 3.15C: failed; objective 1418728.502