Faster Octave and Matlab Code

Christian Himpe (christian.himpe@wwu.de)

WWU Münster
Institute for Computational and Applied Mathematics

23.10.2013

Overview

GNU Octave

What OCTAVE is:

- an **open-source** alternative (clone) to Matlab
  → `http://octave.org`
- reproducing some of the Matlab toolboxes (control, image, optim, multicore) → `http://octave.sourceforge.net`
- providing more (consistent) operators (++,+=,!=,*=,**,...)
  → `http://octave.org/doc/interpreter`

What OCTAVE is not:

- compatible with all commands (but most)
- as fast as Matlab (but close)
- by default providing a GUI (but there are if you insist)

Check if you are in OCTAVE: `exist('OCTAVE_VERSION')`

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- Copy-On-Write
- Java
- Linear Algebra

Acceleration:

- **Preallocate**
- Faster Allocation
- bsxfun
- Copy-On-Write
- Java
- Linear Algebra

```
tic;
for I=1:2000,
 for J=1:2000,
  x(I,J) = I + J;
 end;
end;
toc
```

Acceleration:

- **Preallocate**
- Faster Allocation
- bsxfun
- Copy-On-Write
- Java
- Linear Algebra

```
 tic;
x = zeros(2000);
for I=1:2000,
 for J=1:2000,
  x(I,J) = I + J;
 end;
end;
toc
```

28s vs 12s

Acceleration:

- Preallocate
- **Faster Allocation**
- bsxfun
- Copy-On-Write
- Java
- Linear Algebra

```
tic;
A = zeros(10000);
toc
```

Acceleration:

- Preallocate
- **Faster Allocation**
- bsxfun
- Copy-On-Write
- Java
- Linear Algebra

```
tic;
B(10000,10000) = 0;
toc
```

0.2s vs 0.00003s

Acceleration:

- Preallocate
- Faster Allocation
- **bsxfun**
- Copy-On-Write
- Java
- Linear Algebra

```
tic;
A = rand(2000);
for I=1:2000,
 A(I,:)=A(I,:)-mean(A);
end;
toc
```

Acceleration:

- Preallocate
- Faster Allocation
- **bsxfun**
- Copy-On-Write
- Java
- Linear Algebra

```
tic;
A = rand(2000);
bsxfun(@minus,A,mean(A));
toc
```

7.3s vs 0.1s

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- **Copy-On-Write**
- Java
- Linear Algebra

```
function y = f(m)
 m(1,1) = 1;
 y = sum(sum(m));
end

tic;
f(rand(8000))
toc
```

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- **Copy-On-Write**
- Java
- Linear Algebra

```
function y = f(m)
 y = sum(sum(m))
 y = y - m(1,1) + 1.0;
end

tic;
f(rand(8000))
toc
```

0.25s vs 0.05s

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- Copy-On-Write
- **Java**
- Linear Algebra

```
tic;
h = waitbar(0,'Wait!');
for I=1:2000,
 waitbar(I/2000,h);
end;
toc
```

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- Copy-On-Write
- **Java**
- Linear Algebra

```
tic;
fprintf('Wait!');
for I=1:2000,
 fprintf('|');
end;
fprintf('\n');
toc
```

1.5s vs 0.03s

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- Copy-On-Write
- Java
- **Linear Algebra**

```
tic;
A = rand(2000);
B = rand(2000);
trace(A*B),
toc
```

Acceleration:

- Preallocate
- Faster Allocation
- bsxfun
- Copy-On-Write
- Java
- **Linear Algebra**

```
tic;
A = rand(2000);
B = rand(2000);
sum(sum(A.*B')),
toc
```

2s vs 0.2s

Profiling:

- Reproducible Randomness
- Timing
- Better Timing
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- Memory Profiling

Profiling:

- **Reproducible Randomness**
- Timing
- Better Timing
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- Memory Profiling

```
rand('seed',x);
```

Profiling:

- **Reproducible Randomness**
- Timing
- Better Timing
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- Memory Profiling

```
randn('seed',x);
```

Profiling:

- Reproducible Randomness
- **Timing**
- Better Timing
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- Memory Profiling

```
tic;
  somecode();
toc
```

Profiling:

- Reproducible Randomness
- Timing
- **Better Timing**
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- Memory Profiling

```
T0 = cputime;
somecode();
cputime - T0
```

Profiling:

- Reproducible Randomness
- Timing
- Better Timing
- **Static Code Analysis**
- Code Complexity
- Runtime Profiling
- Memory Profiling

```
mlint('myfunc.m');
```

Profiling:

- Reproducible Randomness
- Timing
- Better Timing
- Static Code Analysis
- **Code Complexity**
- Runtime Profiling
- Memory Profiling

```
mlint('myfunc.m','-cyc')
```

Profiling:

- Reproducible Randomness
- Timing
- Better Timing
- Static Code Analysis
- Code Complexity
- **Runtime Profiling**
- Memory Profiling

```
profile on;
somecode();
profile off;
profreport;
```

Profiling:

- Reproducible Randomness
- Timing
- Better Timing
- Static Code Analysis
- Code Complexity
- Runtime Profiling
- **Memory Profiling**

```
profile -memory on;
somecode();
profile off;
profreport;
```

Miscellaneous:

- Try to vectorize **each** `for`-loop!
- Use current versions of OCTAVE and MATLAB!
- OCTAVE and MATLAB use **column-major** matrices!
- Avoid implicit type-casts!
- Prefer `shiftdim` and `permute` over `squeeze`!
- `arrayfun` can be slower than a `for`-loop!
- Do not use the `Jet`-colormap!
- Adopt a consistent coding style! i.e.:
  `note.sonots.com/Matlab/MatlabCodingStyle.html`

MEX Code:

MEX (Matlab EXecutable) allows to compile MATLAB scripts or enable calling C/C++ functions from MATLAB.

Question you should ask yourself before writing MEX:

- Do I know how to use a compiler?
  (optimization, alignment, machine-type flags etc.)
- Do I know the current C/C++ standards?
  (move semantics, containers, constexpr)
- Do I know what slows down low-level code?
  (aliasing, branching, [wrong] caching)
- Do I know Static Code Analysis and Profiling for C++?
  (cppcheck, valgrind etc.)

tl;dl

Remember:

1. Correctness
2. Performance
3. Documentation
4. Compatibility
5. Readability

Links:

- `http://blogs.mathworks.com/loren`
- `http://matlabtips.com`
- `http://wiki.octave.org`
- `http://matlab.wikia.com/wiki/FAQ`
- `http://gist.github.com/gramian/6027733`

Thanks!