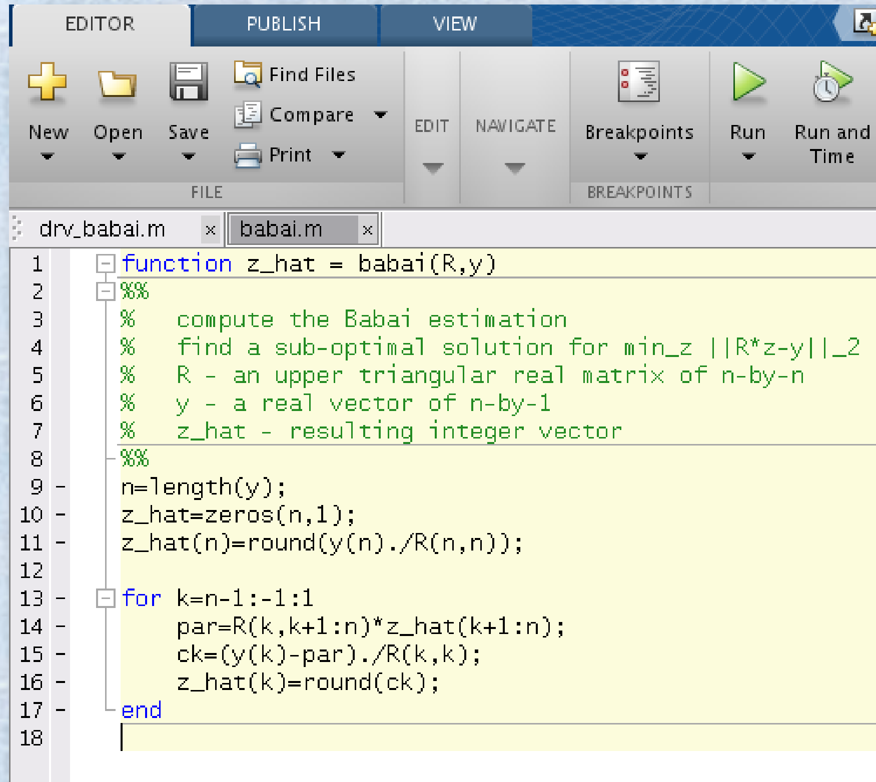
Mc2For: a compiler to transform MATLAB to Fortran 95

Presenter: Xu Li
Supervisor: Laurie Hendren
School of Computer Science
McGill University
xu.li2@mail.mcgill.ca

MATLAB Everywhere!

- Dynamic features, which is ideal for fast prototyping;
- Availability of many high-level array operations and;
- Access to a rich set of built-in functions.
- A quite big user community:
  - students, engineers and even scientists;



(Babai nearest plane algorithm)

Why NOT MATLAB?

- When problem size grows bigger, like
  - function be called a large number of times in one second;
  - large-sized input arrays.

Why NOT MATLAB?

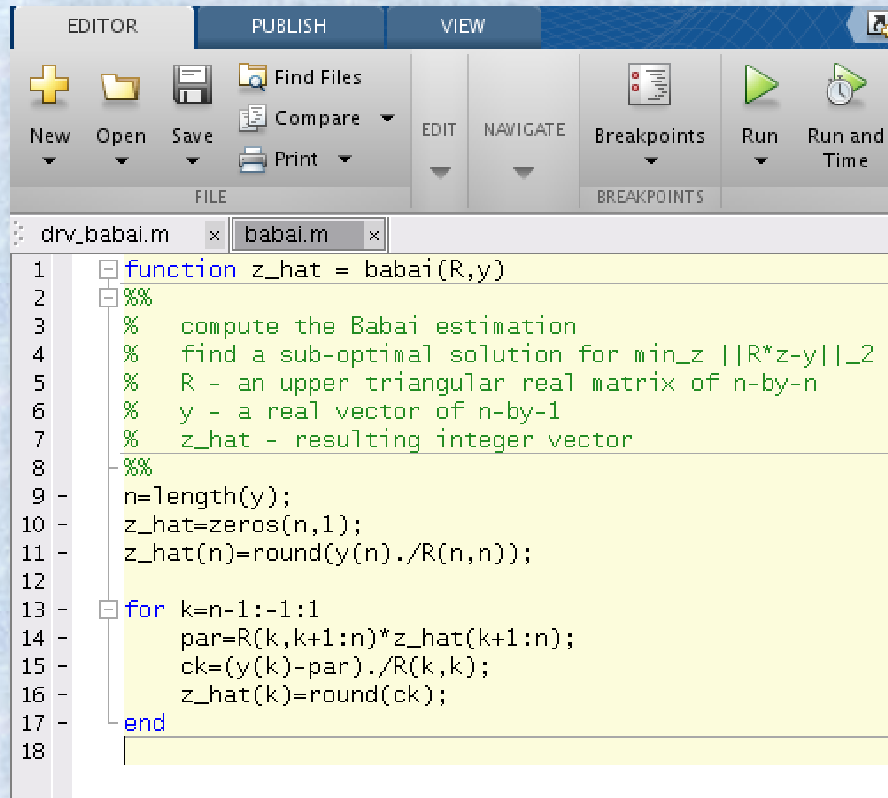
- When problem size grows bigger, like
  - function be called a large number of times in one second;
  - large-sized input arrays.
- Another open source alternative!

Why Fortran?

- History between MATLAB and Fortran;
- Similar syntax;
- Both in column-major order;
- Optimizing Fortran libraries for solving linear algebra problem, like BLAS and LAPACK;
- Numerous optimizing Fortran compilers, including open source compilers like GFortran;

There are challenges…

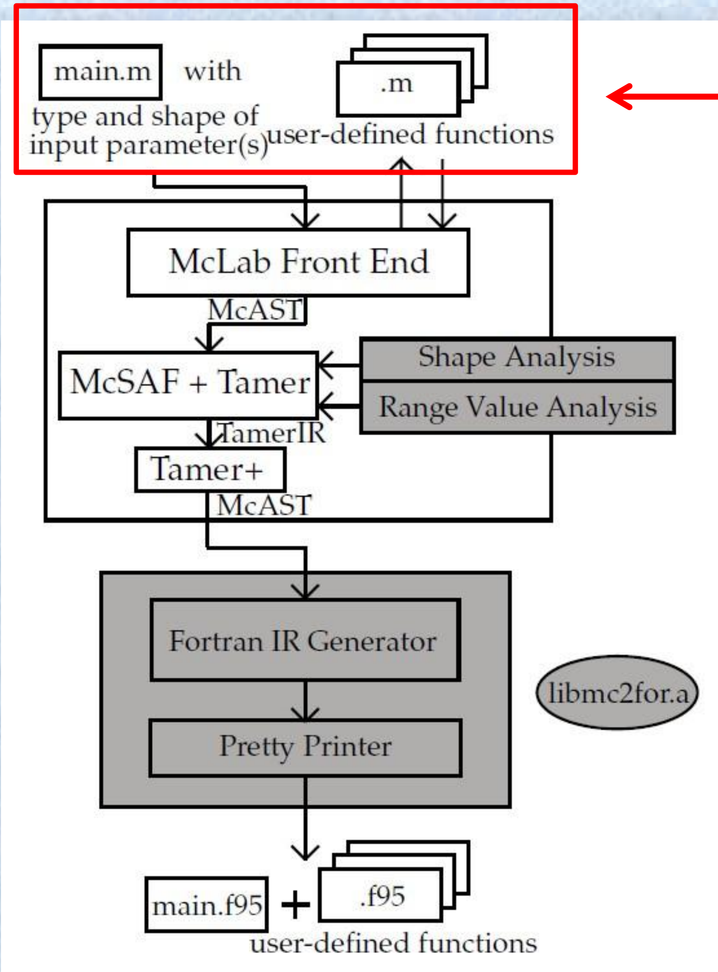
- Dynamic features in MATLAB:
  - no type declaration for variables;
  - arrays can be grown by out-of-bound index;
  - linear array indexing;
  - numerous overloaded built-in functions.



(Babai nearest plane algorithm)

Here comes Mc2For!

Overview of Mc2For

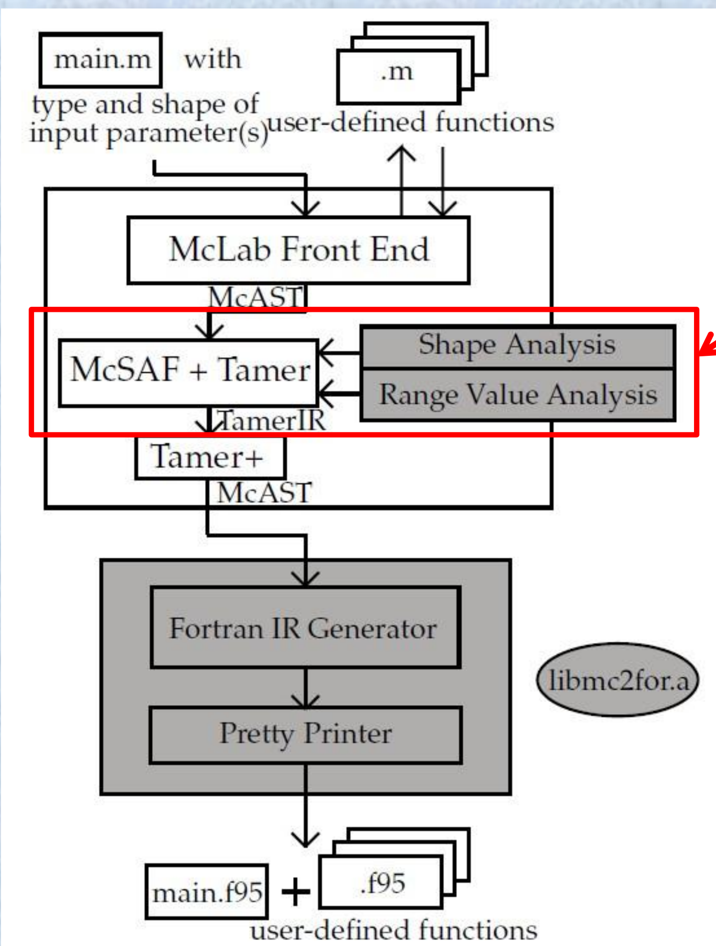



```
function simple(n)

a=2+2;

end
```

\+

```
{n=(double,[1, 1])}
```

Overview of Mc2For

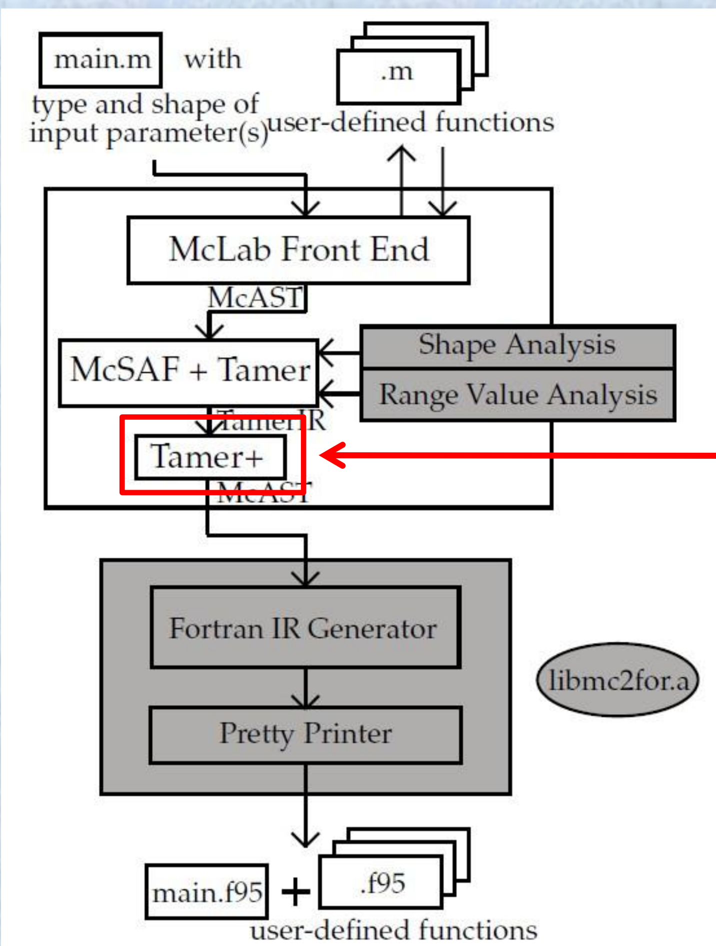



```
function simple(n)

a=2+2;

end
```

➕ `{n=(double,[1, 1])}`

```
% args: {n=(double,[1, 1])}
function  [] = simple(n)
  mc_t0 = 2;                          % mc_t0=(double,2.0,[1, 1],<2, 2>,REAL)
  mc_t1 = 2;                          % mc_t1=(double,2.0,[1, 1],<2, 2>,REAL)
  [a] = plus(mc_t0, mc_t1);           % a=(double,4.0,[1, 1],<4, 4>,REAL)
end
% results: []
```

Overview of Mc2For




```
function simple(n)

a=2+2;

end
```

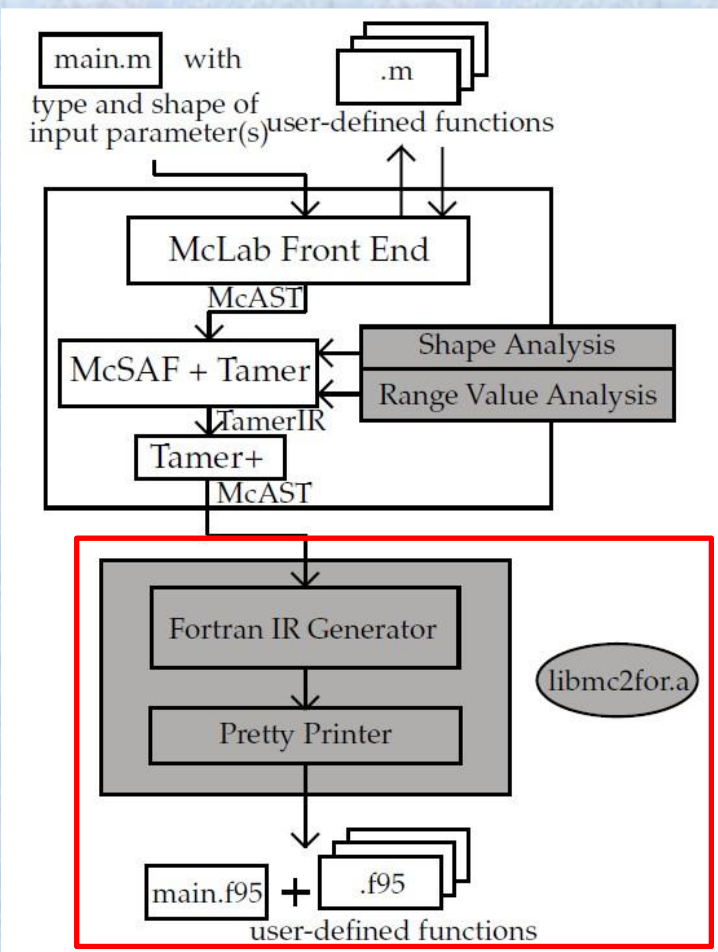
+

```
{n=(double,[1, 1])}
```

```
% args: {n=(double,[1, 1])}
function  [] = simple(n)
  mc_t0 = 2;                          % mc_t0=(double,2.0,[1, 1],<2, 2>,REAL)
  mc_t1 = 2;                          % mc_t1=(double,2.0,[1, 1],<2, 2>,REAL)
  [a] = plus(mc_t0, mc_t1);           % a=(double,4.0,[1, 1],<4, 4>,REAL)
end
% results: []
```

```
function [] = simple(n)
  [a] = plus(2, 2);
end
```

Overview of Mc2For




```
function simple(n)

a=2+2;

end
```

+ `{n=(double,[1, 1])}`

```
% args: {n=(double,[1, 1])}
function  [] = simple(n)
  mc_t0 = 2;                        % mc_t0=(double,2.0,[1, 1],<2, 2>,REAL)
  mc_t1 = 2;                        % mc_t1=(double,2.0,[1, 1],<2, 2>,REAL)
  [a] = plus(mc_t0, mc_t1);         % a=(double,4.0,[1, 1],<4, 4>,REAL)
end
% results: []
```

```
function [] = simple(n)
  [a] = plus(2, 2);
end
```

```
PROGRAM simple
IMPLICIT NONE
DOUBLE PRECISION :: a, n

a = (2 + 2);

END PROGRAM
```

Shape Analysis

- What is the shape analysis?
- Why we need the shape analysis?
- How we implement the shape analysis?
- Biggest challenge:
  - Need a mechanism to propagate shape information through MATLAB built-in functions.
    - i.e., what is the shape of z_hat after the statement of “z_hat = zeros(n, 1)” in the example?

Shape Propagation Equation Language

- length in “n = length(y)”:

  $|M → $

  *the shape of output depends on nothing

- round in “z_hat(k) = round(ck)”:

  $ → $ || M → M

  *depends on the value of input

- zeros in “z_hat = zeros(n, 1)”:

  [ ] → $ || ($,n=previousScalar(),add(n))+ → M

  *depends on the value of input

Shape Propagation Equation Language

The general structures and semantics of constructs in SPEL:

- CASELIST : := case1 || case2 || case3
- CASE ::= pattern list → shape output list
- PATTERN LIST ::= paExp1, paExp2, … paExpn
- PATTERN EXPRESSION:
  - shape matching expressions (SME), can be $, uppercases, and [m,…n],
  - helper function calls, and
  - assignment expressions
- SHAPE OUTPUT LIST ::= ouExp1, ouExp2, … ouExpn
  - same representation as SME, can be $, uppercases, and [m,…n]
- OPERATORS:
  - "()", "?", "*", "+", and "|".

Range Value Analysis

- What is the range value analysis?
  - an extended constant propagation, which statically estimates the minimum and maximum values each scalar variable could take at each program point.
- Why we need the range value analysis?
  - to avoid generating unnecessary run-time array bounds checking code.
- How is the range value of a variable represented?

<minimum, maximum>

Range Value Analysis

- How we implement the range value analysis?
- We select a set of commonly used scalar built-in functions or operators and implement the RVA functions for each of them.

unary plus (+)	binary plus (+)
unary minus (-)	binary minus (-)
element-wise multiplication (.*)	matrix multiplication (*)
element-wise rdivision (./)	matrix rdivision (/)
natural logarithm (log(x))	exponential (exp(x))
absolute value (abs(x))	colon (:)

Tamer+: a Refactoring Component

- Tamer IR is suitable for static flow analysis, but maybe not ideal for code generation.

```
n=length(y);
z_hat=zeros(n,1);
z_hat(n)=round(y(n)./R(n,n));

for k=n-1:-1:1
   par=R(k,k+1:n)*z_hat(k+1:n);
   ck=(y(k)-par)./R(k,k);
   z_hat(k)=round(ck);
end
```

(Input MATLAB Code)

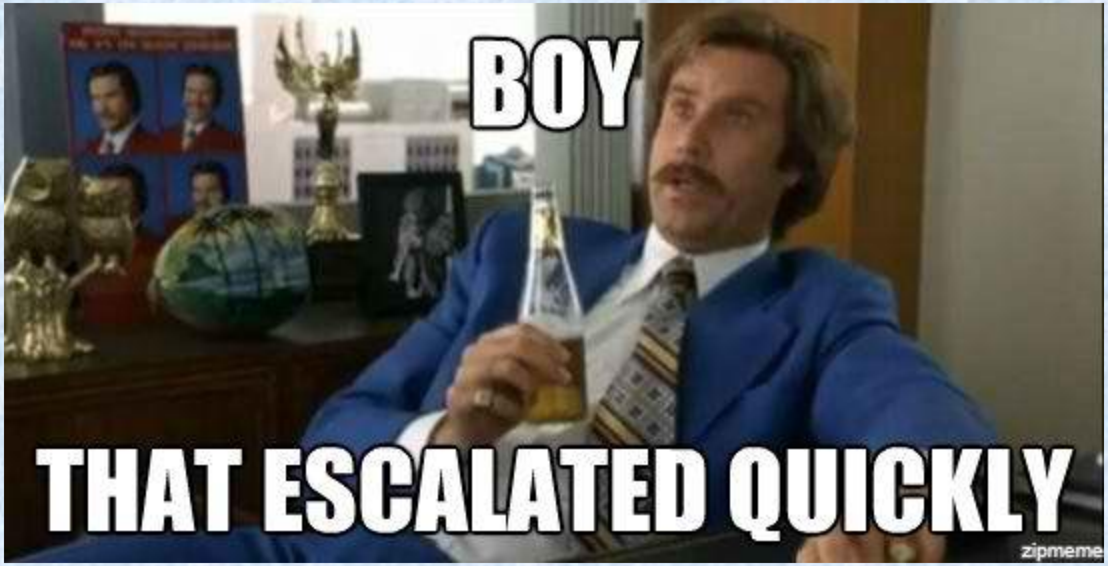
8 lines

```
[n] = length(y);
mc_t25 = 1;
[z_hat] = zeros(n, mc_t25);
[mc_t8] = y(n);
[mc_t9] = R(n, n);
[mc_t7] = rdivide(mc_t8, mc_t9);
[mc_t5] = round(mc_t7);
z_hat(n) = mc_t5;
mc_t26 = 1;
[mc_t23] = minus(n, mc_t26);
mc_t27 = 1;
[mc_t24] = uminus(mc_t27);
mc_t30 = 1;
for k = (mc_t23 : mc_t24 : mc_t30);
  mc_t15 = k;
  mc_t28 = 1;
  [mc_t17] = plus(k, mc_t28);
  mc_t18 = n;
  [mc_t16] = colon(mc_t17, mc_t18);
  [mc_t10] = R(mc_t15, mc_t16);
  mc_t29 = 1;
  [mc_t13] = plus(k, mc_t29);
  mc_t14 = n;
  [mc_t12] = colon(mc_t13, mc_t14);
  [mc_t11] = z_hat(mc_t12);
  [par] = mtimes(mc_t10, mc_t11);
  [mc_t21] = y(k);
  mc_t22 = par;
  [mc_t19] = minus(mc_t21, mc_t22);
  [mc_t20] = R(k, k);
  [ck] = rdivide(mc_t19, mc_t20);
  [mc_t6] = round(ck);
  z_hat(k) = mc_t6;
end
```

34 lines

(Transformed MATLAB code in Tamer IR Version)

Tamer+: a Refactoring Component



Tamer+: a Refactoring Component

- Special thanks to Amine;
- From low-level three-address IR to a high-level IR, Tamer+ IR;
- Based on static flow analysis of def-use and use-def chains.

```
n=length(y);
z_hat=zeros(n,1);
z_hat(n)=round(y(n)./R(n,n));

for k=n-1:-1:1
   par=R(k,k+1:n)*z_hat(k+1:n);
   ck=(y(k)-par)./R(k,k);
   z_hat(k)=round(ck);
end
```

(Input MATLAB Code)

```
[n] = length(y);
[z_hat] = zeros(n, 1);
z_hat(n) = round(rdivide(y(n), R(n, n)));

for k = (minus(n, 1) : uminus(1) : 1);
   [par] = mtimes(R(k, colon(plus(k, 1), n)), z_hat(colon(plus(k, 1), n)));
   [ck] = rdivide(minus(y(k), par), R(k, k));
   z_hat(k) = round(ck);
end
```

(Transformed MATLAB code in Tamer+ IR Version)

Code Generation

- An extensible Fortran code generation framework
  - converting Tamer+ IR to a simplified Fortran IR;
- Handles the general mappings
  - like types, commonly used operators, not-directly-mapped built-in functions, and standard constructs, like if-else, for loop and while loop;
- Handles some dynamic features of MATLAB
  - like run-time array bounds checking, run-time array growth, variable redefinition, and built-in function overloading.

Run-time ABC and Array Reallocation

```
n=length(y);
z_hat=zeros(n,1);
z_hat(n)=round(y(n)./R(n,n));

for k=n-1:-1:1
L1  par=R(k,k+1:n)*z_hat(k+1:n);
    ck=(y(k)-par)./R(k,k);
L2  z_hat(k)=round(ck);
end
```

(Original MATLAB Code)

```
! inline runtime ABC and error handle
IF (k > SIZE(R, 1) .OR. INT(n) > SIZE(R, 2)) THEN
   STOP "INDEX OUT OF BOUND";
END IF
IF (INT(n) > SIZE(z_hat, 1)) THEN
   STOP "INDEX OUT OF BOUND";
END IF
!
par = DOT_PRODUCT(R(k, (k + 1):INT(n)), z_hat((k + 1):INT(n), 1));
```

(Generated Fortran Code Snippet)

```
! inline runtime ABC and reallocation
z_hat_d1 = SIZE(z_hat, 1);
z_hat_d2 = 1;
IF (k > z_hat_d1) THEN
   IF (ALLOCATED(z_hat_bk)) THEN
      DEALLOCATE(z_hat_bk);
   END IF
   ALLOCATE(z_hat_bk(z_hat_d1, z_hat_d2));
   z_hat_bk = z_hat;
   DEALLOCATE(z_hat);
   z_hat_d1max = k;
   z_hat_d2max = 1;
   ALLOCATE(z_hat(z_hat_d1max, z_hat_d2max));
   z_hat(1:z_hat_d1, 1:z_hat_d2) = z_hat_bk(1:z_hat_d1, 1:z_hat_d2);
END IF
!
z_hat(k, 1) = NINT(ck);
```

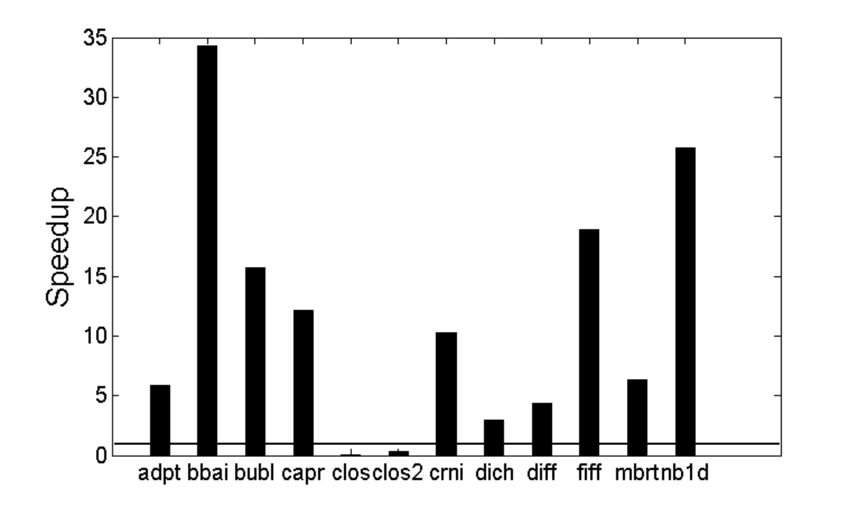
(Generated Fortran Code Snippet)

Mapping Built-in Functions

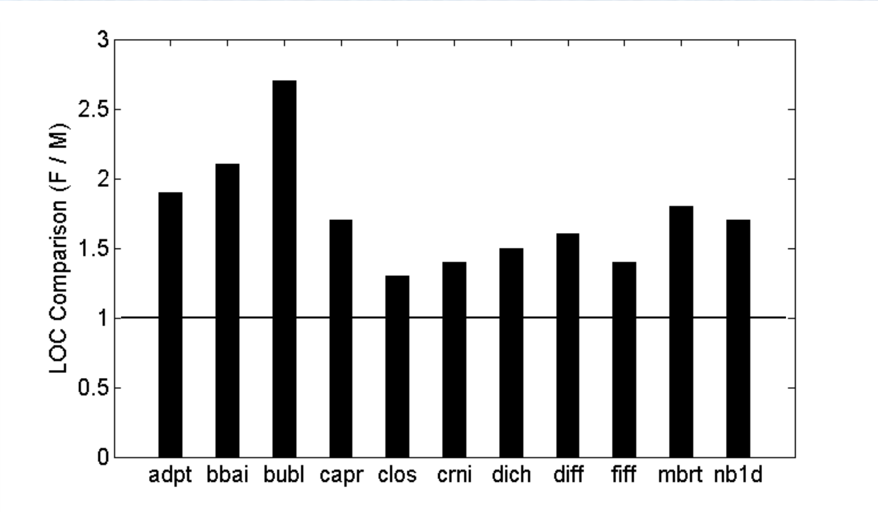
- Built-in function mapping framework:
  - directly-mapped operators;
  - easily-transformed and then inlined operators, like left division and colon;
  - not-directly-mapped built-ins, for most MATLAB built-in functions: leave a hole with same function signature.

- Overloading of built-ins:
  - using Fortran INTERFACE construct.

Performance & LOC Comparison



(Performance with same problem size)



(LOC with nocheck option)

- For most benchmarks, performance speedup is from around 5 to 30;
- For benchmark clos, 24 times slower, using MATMUL of Fortran;
- 3.5 times slower, using DGEMM from one BLAS library;
- MATLAB uses Intel MKL, which has a better implementation of BLAS on Intel Chips;
- The LOC of generated Fortran is in an acceptable range.

Future Work

- Constraint analysis
  - to further remove unnecessary inlined run-time ABC;
- Dependency analysis
  - to determine which MATLAB code block is free from dependency and safe to be transformed to parallel code;
- …

Thank You & Questions?

- Several useful links:
  - McLab: www.sable.mcgill.ca/mclab/
  - Mc2For: www.sable.mcgill.ca/mclab/mc2for.html
  - McLab on GitHub: https://github.com/Sable/mclab/tree/develop
- Convert some MATLAB to Fortran?
  - McLab list: mclab-list@sable.mcgill.ca
  - Xu Li: xu.li2@mail.mcgill.ca

- FOLLOWING SLIDES ARE BACKUP SLIDES.

Range Value Analysis (cont.)

- Domain of the range values:
  - A closed numeric value interval, ordered by

    -inf < all the real numbers < +inf
  - To support RVA through relational built-in functions, we add two superscript symbols, + and -, to the real numbers. For example, 5^-, which can be interpreted as $5-\varepsilon$, where ε is positive and close to 0, and of course, $5-\varepsilon < 5$.

Range Value Analysis (cont.)

```
function range_value_binary_plus(op_a, op_b)
  if both op_a and op_b have known range values
    <a,b> = get range value pair from op_a
    <c,d> = get range value pair from op_b
    return <a+c,b+d>
  else
    return unknown
  end if
end function
```

Note that, a, b, c and d are values in the ***domain of range values***, which is {-inf, real numbers, +inf}.

binary +: if any operand is -inf (+inf), the result will be -inf (+inf); if neither of the operands is -inf nor +inf, the + operator follows the rule as:

$x^- + y^-$, $x^- + y$ or $x + y^- \Rightarrow (x+y)^-$;

$x^+ + y^+$, $x^+ + y$ or $x + y^+ \Rightarrow (x+y)^+$;

$x + y \Rightarrow (x+y)$;

Benchmarks

- ***adpt*** finds the adaptive quadrature using Simpson's rule. This benchmark features an array whose size cannot be predicted before compilation.
- ***bbai*** solves the closest vector problem in linear algebra;
- ***bubl*** is the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations.
- ***capr*** computes the capacitance of a transmission line using finite difference and Gauss-Seidel method. It's a loop-based program that involves basic scalar operations on two small-sized arrays.
- ***clos*** calculates the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays.

Benchmarks (cont.)

- ***crni*** computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array.
- ***dich*** computes the Dirichlet solution to Laplace's Equation. It's also a loop-based program which involves basic scalar operation on a small-sized array.
- ***diff*** calculates the diffraction pattern of monochromatic light through a transmission grating for two slits. This benchmark also features an array hose size is increased dynamically like the benchmark adpt.
- ***fiff*** computes the finite-difference solution to the wave equation. It's a loop-based program which involves basic scalar operation on a 2-dimensional array.
- ***mbrt*** computes a mandelbrot set with specified number elements and number of iterations. This benchmark contains elementary scalar operations on complex type data.

Benchmarks (cont.)

- ***nb1d*** simulates the gravitational movement of a set of objects. It involves computations on vectors inside nested loops.