Numeric performance in C, C# and Java

Peter Sestoft (sestoft@itu.dk)

IT University of Copenhagen
Denmark

Version 0.9.1 of 2010-02-19

Abstract: We compare the numeric performance of C, C# and Java on three small cases.

1 Introduction: Are Java and C# slower than C and C++?

Managed languages such as C# and Java are easier and safer to use than traditional languages such as C or C++ when manipulating dynamic data structures, graphical user interfaces, and so on. Moreover, it is easy to achieve good performance thanks to their built-in automatic memory management.

For numeric computations involving arrays or matrices of floating-point numbers, the situation might seem less favorable. Compilers for Fortran, C and C++ make serious efforts to optimize inner loops that involve array accesses: register allocation, reduction in strength, common subexpression elimination and so on. By contrast, the just-in-time (JIT) compilers of the C# and Java runtime systems do not spend much time on optimizing inner loops, and this hurts numeric code. Moreover, in C# and Java there must be an index check on every array access, and this not only requires execution of extra instructions, but can also lead to branch mispredictions and pipeline stalls on the hardware, further slowing down the computation.

This note compares the numeric performance of Java and C# to that of C on standard laptop hardware. It shows that Java and C# compete well with C also on numeric code; that the choice of execution environment (virtual machine, JIT-compiler) is very important; and that a small amount of unsafe code can seriously improve the speed of some C# programs.

1.1 Case study 1: matrix multiplication

We take matrix multiplication as a prime example of numeric computation. It involves triply nested loops, many array accesses, and floating-point computations, yet the code is so compact that one can study the generated machine code. We find that C performs best, that C# can be made to perform reasonably well, and that Java can perform better than C#. See sections 2 through 5.4.

1.2 Case study 2: a division-intensive loop

We also consider a simple loop that performs floating-point division, addition and comparison, but no array accesses. We find that C# and Java implementations perform better than C, but it turns out that the computation time is dominated by the floating-point division.

1.3 Case study 3: polynomial evaluation

We next consider repeated evaluation of a polynomial of high degree, on which almost all implementations do equally well, with C and Microsoft C# being equally fast, and Java only slightly slower.

1.4 Case study 4: a distribution function

Finally we consider the evaluation the cumulative distribution function for the normal (Gaussian) distribution. Here we find that C is fastest, with both Microsoft C# and Sun Hotspot Java closely following.

2 Matrix multiplication in C

In C, a matrix can be represented by a struct holding the number of rows, the number of columns, and a pointer to a `malloc`'ed block of memory that holds the elements of the matrix as a sequence of doubles:

```
typedef struct {
  int rows, cols;
  double *data;     // (rows * cols) doubles, row-major order
} matrix;
```

If the dimensions of the matrix are known at compile-time, a more static representation of the matrix is possible, but experiments show that for some reason this does not improve speed, quite the contrary.

Given the above struct type, and declarations

```
matrix R, A, B;
```

we can compute the matrix product $R = AB$ in C with this loop:

```
for (r=0; r<rRows; r++) {
  for (c=0; c<rCols; c++) {
    double sum = 0.0;
    for (k=0; k<aCols; k++)
      sum += A.data[r*aCols+k] * B.data[k*bCols+c];
    R.data[r*rCols+c] = sum;
  }
}
```

Note that the programmer must understand the layout of the matrix (here, row-major) and it is his responsibility to get the index computations right.

3 Matrix multiplication in C#

3.1 Straightforward matrix multiplication in C# (matmult1)

In C# we can represent a matrix as a two-dimensional rectangular array of doubles, using type `double[,]`. Assuming the declaration

```
double[,] R, A, B;
```

we can compute $R = AB$ with this loop:

```
for (int r=0; r<rRows; r++) {
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += A[r,k] * B[k,c];
    R[r,c] = sum;
  }
}
```

The variables `rRows`, `rCols` and `aCols` have been initialized from the array dimensions before the loop as follows:

```
int aCols = A.GetLength(1),
    rRows = R.GetLength(0),
    rCols = R.GetLength(1);
```

3.2 Unsafe but faster matrix multiplication in C# (matmult2)

The C# language by default requires array bounds checks and disallows pointer arithmetics, but the language provides an escape from these strictures in the form of so-called unsafe code. Hence the C# matrix multiplication code above can be rewritten closer to C style as follows:

```
for (int r=0; r<rRows; r++) {
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    unsafe {
      fixed (double* abase = &A[r,0], bbase = &B[0,c]) {
        for (int k=0; k<aCols; k++)
          sum += abase[k] * bbase[k*bCols];
      }
      R[r,c] = sum;
    }
  }
}
```

Inside the `unsafe { ... }` block, one can use C-style pointers and pointer arithmetics. The header of the `fixed (...) { ... }` block obtains pointers `abase` and `bbase` to positions within the `A` and `B` arrays, and all indexing is done off these pointers using C/C++-like notation such as `abase[k]` and `bbase[k*bCols]`. The `fixed` block makes sure that the .NET runtime memory management does not move the arrays `A` and `B` while the block executes. (This risk does not exist in C and C++, where `malloc`'ed blocks stay where they are).

Indexing off a pointer as in `abase[k]` performs no index checks, so this code is riskier but faster than that of the previous section.

Notice that we did not have to change the matrix representation to use unsafe code; we continue to use the `double[,]` representation that is natural in C#.

The `unsafe` keyword may seem scary, but note that *all* code in C and C++ is unsafe in the sense of this keyword. To compile a C# program containing unsafe code, one must pass the `-unsafe` option to the compiler:

```
csc /o /unsafe MatrixMultiply3.cs
```

3.3 Java-style matrix multiplication in C# (matmult3)

Finally, we consider a version of matrix multiplication in C# that uses no unsafe code but works on an array-of-arrays representation, as required in Java (section 4)), like this:

```
double[][] R, A, B;
```

rather than the rectangular array implementations shown in section 3.1 above. The multiplication loop then looks like this:

```
for (int r=0; r<rRows; r++) {
  double[] Ar = A[r], Rr = R[r];
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += Ar[k]*B[k][c];
    Rr[c] = sum;
  }
}
```

4 Matrix multiplication in Java

The Java and C# programming languages are managed languages and very similar: same machine model, managed platform, mandatory array bounds checks and so on. There's considerable evidence that Java numeric code can compete with C/C++ numeric code [1, 2, 3].

Some features of Java would seem to make it harder to obtain good performance in Java than in C#:

- Java has only one-dimensional arrays, so a matrix must be represented either as an array of references to arrays of doubles (type `double[][]`) or as a flattened C-style array of doubles (type `double[]`). The former representation can incur a considerable memory access overhead, and the latter representation forces the programmer to explicitly perform index computations.
- Java does not allow unsafe code, so in Java, array bounds checks cannot be circumvented in the way it was done for C# in section 3.2 above.

On the other hand, there is a wider choice of high-performance virtual machines available for Java than for C#. For instance, the "standard" Java virtual machine, namely Hotspot [7] from Sun Microsystems, will aggressively optimize the JIT-generated x86 code if given the `-server` option:

```
java -server MatrixMultiply 80 80 80
```

On Windows, the Sun Hotspot Java virtual machine defaults to `-client` which favors quick start-up over fast generated code, as preferable for most interactive programs. On MacOS it defaults to `-server` for some reason.

Also, IBM's Java virtual machine [8] appears to perform considerable optimizations when generating machine code from the bytecode. There are further high-performance Java virtual machines, such as BEA's jrockit [9], but we have not tested them.

As mentioned, the natural Java representation of a two-dimensional matrix is an array of references to arrays (rows) of doubles, that is, Java type `double[][]`. Assuming the declaration

```
double[][] R, A, B;
```

The corresponding matrix multiplication code looks like this:

```
for (int r=0; r<rRows; r++) {
  double[] Ar = A[r], Rr = R[r];
  for (int c=0; c<rCols; c++) {
    double sum = 0.0;
    for (int k=0; k<aCols; k++)
      sum += Ar[k]*B[k][c];
    Rr[c] = sum;
  }
}
```

Here we have made a small optimization, in that references `Ar` and `Rr` to the arrays `A[r]` and `R[r]`, which represent rows of `A` and `R`, are obtained at the beginning of the outer loop.

This array-of-arrays representation seems to give the fastest matrix multiplication in Java.

5 Compilation of matrix multiplication code

This section presents the bytecode and machine code obtained by compiling the matrix multiplication source codes shown in the previous section, and discusses the speed and deficiencies of this code.

5.1 Compilation of the C matrix multiplication code

Recall the inner loop

```
for (k=0; k<aCols; k++)
  sum += A.data[r*aCols+k] * B.data[k*bCols+c];
```

of the C matrix multiplication code in section 2. The x86 machine code generated for this inner loop by the gcc 4.2.1 compiler with full optimization (`gcc -O3`) is quite remarkably brief:

```
      <loop header not shown>
L7:
      movslq  %edi,%rax               ; move k*bCols to %rax
      movsd   (%r8), %xmm0            ; move A.data[r*aCols+k] to %xmm0
      mulsd   (%r9,%rax,8), %xmm0     ; multiply it with B.data[k*bCols+c]
      addsd   %xmm0, %xmm1            ; add result to sum (in %xmm1)
      incl    %edx                    ; add 1 to k
      addq    $8, %r8                 ; add 8 to A.data index
      addl    %r11d, %edi             ; add bCols to %edi
      cmpl    %r10d, %edx             ; if k!=aCols goto L7
      jne     L7
```

Each iteration of this loop takes 1.4 ns on a 2,660 MHz Intel Core 2 Duo CPU, that is, less than 4 CPU cycles. So it also exploits the CPU's functional units, the caches, and the data buses very well. See also section 10.

5.2 Compilation of the safe C# code

C# source code, like Java source code, gets compiled in two stages:

- First the C# code is compiled to stack-oriented bytecode in the .NET Common Intermediate Language (CIL), using the Microsoft `csc` compiler [6], possibly through Visual Studio, or using the Mono C# compiler `gmcs` [10]. The result is a so-called Portable Executable file, named Matrix-Multiply.exe or similar, which consists of a stub to invoke the .NET Common Language Runtime, some bytecode, and some metadata.

- Second, when the compiled program is about to be executed, the just-in-time compiler of the Common Language Runtime will compile the stack-oriented bytecode to register-oriented machine code for the real hardware (typically some version of the x86 architecture). Finally the generated machine code is executed. The just-in-time compilation process can be fairly complicated and unpredictable, with profiling-based dynamic optimization and so on.

Recall the inner loop of the straightforward C# matrix multiplication (matmult1) in section 3.1:

```
for (int k=0; k<aCols; k++)
  sum += A[r,k] * B[k,c];
```

The corresponding CIL bytecode generated by the Microsoft C# compiler `csc -o` looks like this:

```
<loop header not shown>
IL_005a:  ldloc.s    V_8            // load sum
IL_005c:  ldarg.1                   // load A
IL_005d:  ldloc.s    V_6            // load r
IL_005f:  ldloc.s    V_9            // load k
IL_0061:  call float64[,]::Get(,)   // load A[r,k]
IL_0066:  ldarg.2                   // load B
IL_0067:  ldloc.s    V_9            // load k
IL_0069:  ldloc.s    V_7            // load c
IL_006b:  call float64[,]::Get(,)   // load B[k,c]
IL_0070:  mul                       // A[r,k] * B[k,c]
IL_0071:  add                       // sum + ...
IL_0072:  stloc.s    V_8            // sum = ...
IL_0074:  ldloc.s    V_9            // load k
IL_0076:  ldc.i4.1                  // load 1
IL_0077:  add                       // k+1
IL_0078:  stloc.s    V_9            // k = k+1
IL_007a:  ldloc.s    V_9            // load k
IL_007c:  ldloc.1                   // load aCols
IL_007d:  blt.s      IL_005a        // jump if k<aCols
```

As can be seen, this is straightforward stack-oriented bytecode which hides the details of array bounds checks and array address calculations inside the `float64[,]::Get(,)` method calls.

One can obtain the x86 machine code generated by the Mono 2.6 runtime's just-in-time compiler by invoking it as `mono -v -v`. The resulting x86 machine code is rather cumbersome (and slow) because of the array address calculations and the array bounds checks. These checks and calculations are explicit in the x86 code below; the `Get(,)` method calls in the bytecode have been inlined:

```
<loop header not shown>
0e8        fldl    0xe0(%rbp)              ; load sum on fp stack
0eb        movl    0x08(%rsi),%eax         ; array bounds check
0ee        movl    0x04(%rax),%ecx         ; array bounds check
0f1        movl    0xec(%rbp),%edx         ; array bounds check
0f4        subl    %ecx,%edx               ; array bounds check
0f6        movl    (%rax),%ecx             ; array bounds check
0f8        cmpl    %edx,%ecx               ; array bounds check
0fa        jbeq    0x00000213              ; array bounds check
100-112                                    ; array bounds check
118        imull   %edx,%eax
11b        addl    %ecx,%eax               ; add k
11d        shll    $0x03,%eax              ; multiply by sizeof(double)
120        addl    %esi,%eax               ;
122        addl    $0x00000010,%eax        ;
127        fldl    (%rax)                  ; load A[r][k] on fp stack
129-13d                                    ; array bounds check
143-156                                    ; array bounds check
15c        imull   %ecx,%eax
15f        movl    0xc8(%rbp),%ecx         ; B
162        addl    %edx,%eax
164        shll    $0x03,%eax              ; multiply by sizeof(double)
167        addl    %ecx,%eax
169        addl    $0x00000010,%eax
16e        fldl    (%rax)                  ; load B[k][c] on fp stack
170        fmulp   %st,%st(1)              ; multiply
172        faddp   %st,%st(1)              ; add to sum
174        fstpl   0xe0(%rbp)              ; store sum
177        cmpl    0xd8(%r13),%r15d
17b        jlq     0x000000e8              ; jump if k<aCols
```

For brevity, some repetitive sections of code are not shown.

One drawback of this Mono-generated code is that it uses the "old" floating-point instructions `fmulp` and `faddp` that work on the x87 floating-point stack, rather than the "new" instructions `mulsd` and `addsd` that work on the x86-64 floating-point registers. According to experiments, this x86 code was approximately 6.6 times slower than the code generated from C source by `gcc -O3` and shown in section 5.1. The x86 code generated by Microsoft's just-in-time compiler is slower than the `gcc` code only by a factor of 4.6, and presumably also is neater.

5.3 Compilation of the unsafe C# code

Now let us consider the unsafe (matmult2) version of the C# matrix multiplication code from section 3.2. The inner loop looks like this:

```
fixed (double* abase = &A[r,0], bbase = &B[0,c]) {
  for (int k=0; k<aCols; k++)
    sum += abase[k] * bbase[k*bCols];
}
```

The CIL bytecode generated by Microsoft's C# compiler looks like this:

```
<loop header not shown>
IL_0079:  ldloc.s    V_8            // load sum
IL_007b:  ldloc.s    V_9            // load abase
IL_007d:  conv.i
```

```
IL_007e:  ldloc.s    V_11            // load k
IL_0080:  conv.i
IL_0081:  ldc.i4.8                   // load 8
IL_0082:  mul                        // 8*k
IL_0083:  add                        // abase+8*k
IL_0084:  ldind.r8                   // load abase[k]
IL_0085:  ldloc.s    V_10            // load bbase
IL_0087:  conv.i
IL_0088:  ldloc.s    V_11            // load k
IL_008a:  ldloc.3                    // load bCols
IL_008b:  mul                        // k*bCols
IL_008c:  conv.i
IL_008d:  ldc.i4.8                   // load 8
IL_008e:  mul                        // 8*k*bCols
IL_008f:  add                        // bbase+8*k*bCols
IL_0090:  ldind.r8                   // load bbase[k*bCols]
IL_0091:  mul                        // multiply
IL_0092:  add                        // add sum
IL_0093:  stloc.s    V_8             // sum = ...
IL_0095:  ldloc.s    V_11            // load k
IL_0097:  ldc.i4.1                   // load 1
IL_0098:  add                        // k+1
IL_0099:  stloc.s    V_11            // k = ...
IL_009b:  ldloc.s    V_11            // load k
IL_009d:  ldloc.1                    // load aCols
IL_009e:  blt.s      IL_0079         // jump if k<aCols
```

At first sight this appears even longer and more cumbersome than the matmult1 bytecode sequence in section 5.2, but note that the new code does not involve any calls to the `float64[,]::Get(,)` methods, and hence does not contain any hidden costs.

The corresponding x86 machine code generated by the Mono 2.6 runtime is much shorter in this case:

```
<loop header not shown>
0a8         fldl    0xe0(%rbp)              ; load sum on fp stack
0ab         movl    %esi,%ecx               ; load k
0ad         shll    $0x03,%ecx              ; 8 * k
0b0         movl    %ebx,%eax               ; load abase
0b2         addl    %ecx,%eax               ; abase+8*k
0b4         fldl    (%rax)                  ; abase[k]
0b6         movl    0xd4(%rbp),%eax         ; load bCols
0b9         movl    %esi,%ecx               ; load k
0bb         imull   %eax,%ecx               ; bCols*k
0be         shll    $0x03,%ecx              ; 8*bCols*k
0c1         movl    %edi,%eax               ; load bbase
0c3         addl    %ecx,%eax
0c5         fldl    (%rax)                  ; bbase[k*bCols]
0c7         fmulp   %st,%st(1)              ; multiply
0c9         faddp   %st,%st(1)              ; add sum
0cb         fstpl   0xe0(%rbp)              ; store into sum
0ce         cmpl    0xd8(%rbp),%r14d        ; jump if k<aCols
0d2         jl      0x000000a8
```

Clearly this unsafe code is far shorter than the x86 code in section 5.2 that resulted from safe bytecode. One iteration of this loop takes 3.8 ns on a 2,660 MHz Intel Core 2 duo, using the Mono 2.6 runtime.

However, one iteration of the corresponding x86 code generated by Microsoft's runtime takes only 2.3 ns, so presumably the corresponding machine code looks a little neater also.

Microsoft's Visual Studio development environment does allow one to inspect the x86 code generated by the just-in-time compiler, but only when debugging a C# program: Set a breakpoint in the method whose x86 you want to see, choose `Debug | Start debugging`, and when the process stops, choose `Debug | Windows | Disassembly`. Unfortunately, since this works only in debugging mode, the x86 code shown contains extraneous and wasteful instructions.

In fact, the x86 code in debugging mode is twice as slow as non-debugging code. Hence the x86 code obtained from Visual Studio during debugging does not give a good indication of the code quality that is actually achievable. To avoid truly bad code, make sure to check the `Optimize` checkbox in the `Project | Properties | Build` form in Visual Studio.

5.4 Compilation of the Java matrix multiplication code

The bytecode resulting from compiling the Java code in section 4 with the Sun Java compiler `javac` is fairly similar to the CIL bytecode code shown in section 5.2.

Remarkably, the straightforward Java implementation, which uses no unsafe code and a seemingly cumbersome array representation, performs of a par with the unsafe C# code when executed with Sun's Hotspot JVM. Presumably it would be even faster on the IBM Java virtual machine [8], but that is not available for the Mac OS platform (and only for Windows if running on IBM hardware).

The are development versions of the JVM (from Sun or OpenJDK) that can display the machine code generated by the JIT compiler, but we have not investigated this.

6 Controlling the runtime and the just-in-time compiler

I know of no publicly available options or flags to control the just-in-time optimizations performed by Microsoft's .NET Common Language Runtime, but surely options similar to Sun's `-client` and `-server` must exist internally. I know that there is (or was) a Microsoft-internal tool called `jitmgr` for configuring the .NET runtime and just-in-time compiler, but it does not appear to be publicly available. Presumably many people would just use it to shoot themselves in the foot.

Note that the so-called server build (mscorsvr.dll) of the Microsoft .NET runtime differs from the workstation build (mscorwks.dll) primarily in using a concurrent garbage collector. According to MSDN, the workstation build will always be used on uniprocessor machines, even if the server build is explicitly requested.

The Mono runtime accepts a range of JIT optimization flags, such as

```
mono --optimize=all MatrixMultiply 80 80 80
```

but at the time of writing (Mono version 2.6, February 2010), specifying such flags seem to make matrix multiplication performance worse. This is good in a sense: the default behavior is the best.

7 Case study 2: A division-intensive loop

Consider for a given M the problem of finding the least integer n such that

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq M$$

In C, Java and C# the problem can be solved by the following program fragment:

```
double sum = 0.0;
int n = 0;
while (sum < M) {
  n++;
  sum += 1.0/n;
}
```

For $M = 20$ the answer is $n = 272\ 400\ 600$ and the loop performs that many iterations. Each iteration involves a floating-point comparison, a floating-point division and a floating-point addition, as well as an integer increment.

The computation time is probably dominated by the double-precision floating-point division operation. The Intel performance documentation [5] says that the throughput for double precision floating-point division `DIVSD` is (less than) 20 cycles per division, on the Core 2 Duo. Since the loop condition depends on the addition and division, 20 cycles would require 7.5 ns per iteration on the 2,660 MHz CPU we are using. Indeed all implementations take between 7.7 ns and 10.6 ns per iteration.

8 Case study 3: Polynomial evaluation

A polynomial $c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$ can be evaluated efficiently and accurately using Horner's rule:

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n = c_0 + x \cdot (c_1 + x \cdot (\ldots + x \cdot (c_n + x \cdot 0) \ldots))$$

Polynomial evaluation using Horner's rule can be implemented in C, Java and C# like this:

```
double res = 0.0;
for (int i=0; i<cs.Length; i++)
  res = cs[i] + x * res;
```

where the coefficient array `cs` has length $n + 1$ and coefficient c_i is in element `cs[`$n - i$`]`.

The x86-64 code generated for the above polynomial evaluation loop by `gcc -O3` from C is this:

```
        <loop header not shown>
L37:
        cmpl    %r13d, %edx             ; if i<order
        jl      L27                     ; continue
L39:
        <leave loop>                    ; else leave loop
L27:
        movslq  %edx,%rax               ; move i into %rax
        mulsd   %xmm3, %xmm1            ; multiply x into res
        addsd   (%r14,%rax,8), %xmm1    ; add cs[i] to res
        incl    %edx                    ; i++
        jmp     L37                     ; goto L37
```

Note that the entire computation is done with `res` in a floating-point register; not once during the loop is anything written to memory. The array accesses happen in the `addsd` instruction which multiplies the floating-point number at `%r14+%rax*8` into the register holding `res`.

All implementations fare almost equally well on this problem, with C and C# (on Microsoft's .NET as well as Mono) being the fastest at 3.0 ns per loop iteration. Each iteration performs a floating-point addition and a multiplication, but here the multiplication uses the result of the preceding addition (via a loop-carried dependence), which may be the reason this is so much slower than the matrix multiplication loop in section 5.1.

The reason for the Microsoft implementation's excellent performance may be that it can avoid the array bounds check in `cs[i]`. The just-in-time compiler can recognize bytecode generated from loops of exactly this form:

```
for (int i=0; i<cs.Length; i++)
  ... cs[i] ...
```

and will not generate array bounds checks for the `cs[i]` array accesses [11]. Apparently this optimization is rather fragile; small deviations from the above code pattern will prevent the just-in-time compiler from eliminating the array bounds check. Also, experiments confirm that this optimization is useless in the safe matrix multiplication loop (section 3.1), where at least two of the four index expressions appear not to be bounded by the relevant array length (although in reality they are, of course).

9 Case study 4: Distribution function evaluation

The cumulative distribution of the normal (Gaussian) distribution can be implemented like this (here in C#; the Java and C versions are nearly identical):

```
public double F(double z) {
  double p, zabs = Math.Abs(z);
  if (zabs > 37)
    p = 0;
  else { // |z| <= 37
      double expntl = Math.Exp(zabs * zabs * -.5);
      double pdf = expntl / root2pi;
      if (zabs < cutoff)  // |z| < CUTOFF = 10/sqrt(2)
        p = expntl * ((((((p6 * zabs + p5) * zabs + p4) * zabs + p3) * zabs
              + p2) * zabs + p1) * zabs + p0) / ((((((q7 * zabs + q6) *
              zabs + q5) * zabs + q4) * zabs + q3) * zabs + q2) * zabs + q1)
               * zabs + q0);
      else // CUTOFF <= |z| <= 37
        p = pdf / (zabs + 1 / (zabs + 2 / (zabs + 3 / (zabs + 4 / (zabs
                  + .65)))));
    }
  if (z < 0)
    return p;
  else
    return 1-p;
}
```

The $\mathtt{p}i$ and $\mathtt{q}j$ are double precision floating-point constants. The `cutoff` is 7.071 so for arguments around -3 the function performs 15 multiplications, 2 divisions and 13 additions, and computes the exponential function.

10 Experiments

10.1 Matrix multiplication performance

This table shows the CPU time (in microseconds) per matrix multiplication, for multiplying two 80x80 matrices:

C (`gcc -O3`)	702
C# matmult1 Microsoft	3218
C# matmult1 Mono	4627
C# matmult2 Microsoft	1165
C# matmult2 Mono	1943
C# matmult3 Microsoft	1575
C# matmult3 Mono	2888
Java, Sun Hotspot -server	1180

We see that the best C# results are a factor of 1.65 slower than the best C results, using unsafe features of C#. The best Java results on Sun's Hotspot JVM are only marginally slower, which is impressive considering that no unsafe code is used, and that Java has a somewhat cumbersome array representation. (The IBM JVM is often even faster that Sun's Hotspot JVM, but unfortunately is not available for Mac OS 10.6).

Depending on circumstances, the resulting C# performance may be entirely acceptable, given that the unsafe code can be isolated to very small fragments of the code base, and the advantages of safe code and dynamic memory management can be exploited everywhere else. Also, in 2014 a standard workstation may have 16 or 32 CPUs, and then it will probably be more important to exploit parallel computation than to achieve raw single-processor speed.

10.2 Division-intensive loop performance

For the simple division-intensive loop shown in section 7 the execution times are as follows, in nanoseconds per iteration of the loop:

C (`gcc -O3`)	7.9
C# Microsoft	7.7
C# Mono	7.6
Java, Sun Hotspot -server	10.6

Here C and both implementations of C# perform equally well.

10.3 Polynomial evaluation performance

The execution times for evaluation of a polynomial of order 1000 (microseconds per polynomial evaluation), implemented as in section 8 are as follows:

C (`gcc -O3`)	3.0
C# Microsoft	3.1
C# Mono	5.3
Java, Sun Hotspot -server	3.0

The performance of C, Microsoft C# and Sun's Hotspot `-server` must be considered identical. The Mono C# implementation is a factor of 1.7 slower than the best performance in this case.

10.4 Distribution function evaluation performance

The execution times for evaluation of the distribution function at $-3 + i \cdot 10^{-9}$ for 10,000,000 calls (nanoseconds per function evaluation), implemented as in section 9 are as follows:

C (`gcc -O3`)	54
C# Microsoft	64
C# Mono	146
Java, Sun Hotspot -server	69

The performance of C is best with Microsoft C# and Sun's Hotspot `-server` closely behind. The Mono C# implementation is a factor of 2.7 slower than the best performance in this case.

10.5 Details of the experimental platform

- Main hardware platform: Intel Core 2 Duo (presumably family 6 model 15) at 2,660 MHZ, 3 MB L2 cache, 4 GB RAM.
- Operating system: Mac OS X 10.6.2.
- Alternate operating system: Windows XP SP3 under Parallels VM under Mac OS.
- C compiler: gcc 4.2.1 optimization level -O3
- Microsoft C# compiler 4.0.21006.1 with compile options `-o -unsafe`; and MS .NET runtime 4.0.
- Mono runtime: `mono` version 2.6 for MacOS.
- Java compiler and runtime Sun Hotspot 64-Bit Server VM 1.6.0-17 (build 14.3-b01-101, mixed mode) for MacOS X.

11 Conclusion

The experiments show that there is no obvious relation between the execution speeds of different software platforms, even for the very simple programs studied here: the C, C# and Java platforms are variously fastest and slowest.

Some points that merit special attention:

- Given Java's cumbersome array representation and the absence of unsafe code, it is remarkable how well the Sun Hotspot `-server` and virtual machine performs.
- Microsoft's C#/.NET runtime in generally performs very well, but there is much room for improvement in the safe code for matrix multiplication.
- The Mono C#/.NET runtime now is very reliable, and in version 2.6 the general performance is good.

References

[1] J.P. Lewis and Ulrich Neumann: Performance of Java versus C++. University of Southern California 2003.
http://www.idiom.com/~zilla/Computer/javaCbenchmark.html

[2] National Institute of Standards and Technology, USA: JavaNumerics.
http://math.nist.gov/javanumerics/

[3] CERN and Lawrence Berkeley Labs, USA: COLT Project, Open Source Libraries for High Performance Scientific and Technical Computing in Java.
http://dsd.lbl.gov/~hoschek/colt/

[4] P. Sestoft: Java performance. Reducing time and space consumption. KVL 2005.
http://www.dina.kvl.dk/~sestoft/papers/performance.pdf

[5] Intel 64 and IA-32 Architectures Optimization Reference Manual. November 2006.
http://www.intel.com/design/processor/manuals/248966.pdf

[6] Microsoft Developer Network: .NET Framework Developer Center.
http://msdn.microsoft.com/netframework/

[7] The Sun Hotspot Java virtual machine is found at http://java.sun.com

[8] The IBM Java virtual machine is found at
http://www-128.ibm.com/developerworks/java/jdk/

[9] The BEA jrockit Java virtual machine is found at
http://www.bea.com/content/products/jrockit/

[10] The Mono implementation of C# and .NET is found at http://www.mono-project.com/

[11] Gregor Noriskin: Writing high-performance managed applications. A primer. Microsoft, June 2003. At http://msdn2.microsoft.com/en-us/library/ms973858.aspx