FROM: JOHN D. WATTON
APPLIED MATHEMATICS AND
COMPUTER TECHNOLOGY CENTER
ALCOA TECHNICAL CENTER - D

TO: DISTRIBUTION

May 19, 1997

RE: COMPARING C, FORTRAN, LISP, JAVA, PYTHON, PERL, SCHEME (GUILE), AND TCL USING A FLOATING POINT NUMERICAL TEST - POINT INSIDE POLYGON

What follows is a small study I did to compare various computing languages. I was especially interested in the newer interpreted ones that seem to generate so much interest for one reason or another. I wanted to know how their syntax and performance compared to each other and also how they compare to languages that have a native compiler available. This test very much favors speed of floating point numerical calculation and is just one lone data point on the subject of language comparison. Still I believe it is helpful. First some qualifying statements: it does not consider that one can embed the C version into Perl, Tcl, etc.; also this test does not consider the various ...to-C compilers available for Tcl, Perl, and Scheme or the native Scheme compilers available (some for free).

The benchmark test is actually some very useful code in computer graphics. It is a function to determine if a 2D point is inside a closed 2D polygon using the ray crossing algorithm. The polygon need not be convex. The original C version came from the comp.graphics.algorithms FAQ and all of the other versions were written by me with some improvements by reviewers on Usenet news groups. The function is called on 12 different points half in and half out (alternating) for 100000 iterations summing up how many points are inside - which comes to 12*1/2*100000 or 600000.

Figure 1 is a table comparing execution speed. Figures 2-9 contain the source code for each language. I don't want to prejudice anyone's interpretation of the data so I'll only make some mild observations:
1. Unsurprisingly, native compiled code is much faster than interpreted code.
2. Generally typed interpreted languages such as Java and C interpreted by CINT, outperform untyped interpreted languages.
3. Going into Tcl, Perl, Java, and Python with little or no prior experience - Python was fastest to pick up on and debug; Tcl the slowest.

JOHN D. WATTON

MACHINE-TYPE	SPARCstation10 100MHz 128Mb RAM
SOFTWARE-VERSION	SunOS 5.4 sun4m

native compiler	**time (sec)**	**language**	**HTML**

gcc -O3 (2.7.2)	6.0	C	www.gnu.ai.mit.edu
f77 -O (SC3.0.1)	7.5	Fortran	
Franz ACL 4.3	8.4	Common Lisp	www.franz.com
cc -O (SC3.0.1)	8.5	C	
interpreter			
Kaffe 0.8.4	77.0	Java	www.kaffe.org
Sun's JDK 1.1.1	80.0	Java	www.javasoft.com
Sun's JDK 1.0.2	163.0	Java	www.javasoft.com
CINT 5.13	530.0	C	hpsalo.cern.ch/root/Cint
Python1.4	1280.0	Python	www.python.org
Perl5.003	1710.0	Perl	www.perl.com
Guile1.0	2060.0	Scheme	www.gnu.ai.mit.edu
Tcl8.0a2	2240.0	Tcl	sunscript.sun.com
Tcl7.4	24600.0	Tcl	sunscript.sun.com

Figure 1. Table comparing speed of execution for the pnpoly test code in Figures 2-9. These results are for one hardware platform and one test function very much favoring numerical floating point calculations. They are broken into the broad categories of native compiled and interpreted. Native compiled means that the compiler generates machine instructions specific to the hardware architecture. Interpreted generally means that the source code is interpreted by a program that executes the instructions. All of the listed interpreters accept Tcl7.4 use a technique called machine independent byte compilation to increase interpretation performance. In addition Kaffe uses an additional technique called JIT (Just In Time) compilation to transform the machine independent byte code into native machine instructions. All of the language interpreters in this study are free and can be downloaded from the listed web site. In addition gcc is a free C compiler. The other C and Fortran compilers are supplied commercially by Sun. While there are many free Lisp compilers and interpreters Franz Allegro Common Lisp is also a commercial product. Additional notes can be found in the subsequent figure captions.

```
int pnpoly(int npol, double *xp, double *yp, double x, double y)
{
  int i, j, c = 0;
  for (i = 0, j = npol-1; i < npol; j = i++) {
    if ((((yp[i]<=y) && (y<yp[j])) ||
         ((yp[j]<=y) && (y<yp[i]))) &&
        (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))
      c = !c;
  }
  return c;
}

main() {
  int npol=20, count=0;
  double xp[20]= {0.0,1.0,1.0,0.0,0.0,1.0,-0.5,-1.0,-1.0,-2.0,
                  -2.5,-2.0,-1.5,-.5,1.0,1.0,0.0,-0.5,-1.0,-.5};
  double yp[20]= {0.0,0.0,1.0,1.0,2.0,3.0,2.0,3.0,0.0,-.5,
                  -1.0,-1.5,-2.0,-2.0,-1.5,-1.0,-.5,-1.0,-1.0,-.5};
  int i=0;
  for(i=0;i<100000;i++) {
    if (pnpoly(npol,xp,yp,0.5,0.5)) count++;
    if (pnpoly(npol,xp,yp,0.5,1.5)) count++;
    if (pnpoly(npol,xp,yp,-0.5,1.5)) count++;
    if (pnpoly(npol,xp,yp,0.75,2.25)) count++;
    if (pnpoly(npol,xp,yp,0.0,2.01)) count++;
    if (pnpoly(npol,xp,yp,-0.5,2.5)) count++;
    if (pnpoly(npol,xp,yp,-1.0,-0.5)) count++;
    if (pnpoly(npol,xp,yp,-1.5,0.5)) count++;
    if (pnpoly(npol,xp,yp,-2.25,-1.0)) count++;
    if (pnpoly(npol,xp,yp,0.5,-0.25)) count++;
    if (pnpoly(npol,xp,yp,0.5,-1.25)) count++;
    if (pnpoly(npol,xp,yp,-0.5,-2.5)) count++;
  }
  printf("count %d \n", count);
  return(0);
}
```

Figure 2. C code for point inside a nonconvex polygon. The code is attributed to Wm. Randolph Franklin <wrf@ecse.rpi.edu> in the comp.graphics.algorithms FAQ with additional references [“Graphics Gems IV” pp. 24-46] and [J. O'Rourke, “Computational Geometry in C” pp. 233-238]. C has had an ANSI standard since 1989. The language has a reputation as a portable assembler and indeed the interpreted languages examined in this memo are written in C for the most part. C has a terse syntax and excellent compilers available commercially and for free from the GNU Free Software Foundation. While most people know C as solely a compiled language there are interpreters available. In this memo I use a free one (written in C) called CINT. It was created by Masaharu Goto of HP Japan and can be useful for developing C code. Note: CINT 5.13 has a bug: replace c = !c; with c ^= 1; to get around.

```
      logical function pnpoly(npol, xp, yp, x, y)
      integer npol
      double precision x, y, xp(*), yp(*)
      pnpoly=.false.
      j=npol
      do 10 i=1,npol
         if ((((yp(i).le.y).and.(y.lt.yp(j))).or.
     1        ((yp(j).le.y).and.(y.lt.yp(i)))).and.
     2        (x.lt.((xp(j)-xp(i))*((y-yp(i))/(yp(j)-yp(i)))+xp(i)))) pnpoly=.not.pnpoly
        j=i
10    continue
      end

      program pntest
      integer npol, count
      logical pnpoly
      double precision xp(20), yp(20)
      data npol,count/20, 0/
      data  xp/0.0d0,1.0d0,1.0d0,0.0d0,0.0d0,1.0d0,-0.5d0,-1.0d0,-1.0d0,-2.0d0,
     1     -2.5d0,-2.0d0,-1.5d0,-0.5d0,1.0d0,1.0d0,0.0d0,-0.5d0,-1.0d0,-0.5d0/
      data  yp/0.0d0,0.0d0,1.0d0,1.0d0,2.0d0,3.0d0,2.0d0,3.0d0,0.0d0,-0.5d0,
     1     -1.0d0,-1.5d0,-2.0d0,-2.0d0,-1.5d0,-1.0d0,-0.5d0,-1.0d0,-1.0d0,-0.5d0/
      do 20 i=1,100000
         if (pnpoly(npol,xp,yp,0.5d0,0.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,0.5d0,1.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,-0.5d0,1.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,0.75d0,2.25d0)) count=count+1
         if (pnpoly(npol,xp,yp,0.0d0,2.01d0)) count=count+1
         if (pnpoly(npol,xp,yp,-0.5d0,2.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,-1.0d0,-0.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,-1.5d0,0.5d0)) count=count+1
         if (pnpoly(npol,xp,yp,-2.25d0,-1.0d0)) count=count+1
         if (pnpoly(npol,xp,yp,0.5d0,-0.25d0)) count=count+1
         if (pnpoly(npol,xp,yp,0.5d0,-1.25d0)) count=count+1
         if (pnpoly(npol,xp,yp,-0.5d0,-2.5d0)) count=count+1
20    continue
      print *, 'count = ', count
      end
```

Figure 3. Fortran version of pnpoly. A Fortran version is included only because when I posted an early version of the pnpoly benchmarks to various Usenet news groups for review, in order to solicit help in improving the pnpoly source for languages new to me, a few people claimed that Fortran is much faster than C. According to this benchmark, Fortran is only comparable to C in performance.

```
(defun pnpoly (npol xp yp x y)
 (declare (optimize (speed 3) (safety 0))
          (fixnum npol)
          (double-float x y)
          (type (simple-array double-float (*)) xp yp))
 (let* ((c nil)
        (j (1- npol)))
  (declare (fixnum j))
  (dotimes (i npol c)
   (declare (fixnum i))
   (if (and (or (and (<= (aref yp i) y) (< y (aref yp j)))
                (and (<= (aref yp j) y) (< y (aref yp i))))
            (< x (+ (aref xp i) (/ (* (- (aref xp j) (aref xp i)) (- y (aref yp i)))
                                   (- (aref yp j) (aref yp i))))))
       (setq c (not c)))
   (setq j i))))

(defun pnpolytest ()
 (declare (optimize (speed 3) (safety 0)))
 (let ((npol 20)
       (count 0)
       (xp (make-array 20 :element-type 'double-float
                          :initial-contents '(0.0d0 1.0d0 1.0d0 0.0d0 0.0d0 1.0d0 -0.5d0 -1.0d0
                                              -1.0d0 -2.0d0 -2.5d0 -2.0d0 -1.5d0 -0.5d0 1.0d0
                                              1.0d0 0.0d0 -0.5d0 -1.0d0 -.5d0)))
       (yp (make-array 20 :element-type 'double-float
                          :initial-contents '(0.0d0 0.0d0 1.0d0 1.0d0 2.0d0 3.0d0 2.0d0 3.0d0
                                              0.0d0 -0.5d0 -1.0d0 -1.5d0 -2.0d0 -2.0d0 -1.5d0
                                              -1.0d0 -0.5d0 -1.0d0 -1.0d0 -0.5d0))))
  (declare (fixnum npol count)
           (type (simple-array double-float (20)) xp yp))
  (dotimes (i 100000)
   (if (pnpoly npol xp yp 0.5d0 0.5d0) (incf count))
   (if (pnpoly npol xp yp 0.5d0 1.5d0) (incf count))
   (if (pnpoly npol xp yp -0.5d0 1.5d0) (incf count))
   (if (pnpoly npol xp yp 0.75d0 2.25d0) (incf count))
   (if (pnpoly npol xp yp 0.0d0 2.01d0) (incf count))
   (if (pnpoly npol xp yp -0.5d0 2.5d0) (incf count))
   (if (pnpoly npol xp yp -1.0d0 -0.5d0) (incf count))
   (if (pnpoly npol xp yp -1.5d0 0.5d0) (incf count))
   (if (pnpoly npol xp yp -2.25d0 -1.0d0) (incf count))
   (if (pnpoly npol xp yp 0.5d0 -0.25d0) (incf count))
   (if (pnpoly npol xp yp 0.5d0 -1.25d0) (incf count))
   (if (pnpoly npol xp yp -0.5d0 -2.5d0) (incf count)))
  (princ "Count ") (princ count)
  count))
```

Figure 4. Common Lisp version of pnpoly. Common Lisp has been an ANSI standard since 1993 although Lisp history goes back to the 1950's. Lisp in it's modern form has many attractive features: (1) Interactive development environments. (2) Dynamic linking and debugging that allows a programmer to inspect the state of a program at its point of failure, to make an incremental fix, and continue from that spot. (3) A broad spectrum of built in tools and runtime variable typing that is especially suited to fast

prototyping. (4) Ability to emphasize performance and robustness when needed. (5) Ease of embedding other domain- or problem-specific languages while retaining all the capabilities of the underlying Lisp language. (6) Automatic memory management, guaranteeing both the integrity of memory and its efficient reuse. (7) Ability to express functions as objects. (8) A natural and simple means of reflecting and manipulating its own syntax making powerful macros available.

```
public class Pnpoly {
  public static boolean pnpoly(int npol, double[] xp, double[] yp, double x, double y)
    {
      int i, j;
     boolean c = false;
     for (i = 0, j = npol-1; i < npol; j = i++) {
           if ((((yp[i]<=y) && (y<yp[j])) ||
                ((yp[j]<=y) && (y<yp[i]))) &&
               (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))
             c = !c;
     }
     return c;
    }

  public static void main(String args[]) {
    int npol=20, count=0;
    double[] xp = {0.0,1.0,1.0,0.0,0.0,1.0,-0.5,-1.0,-1.0,-2.0,
                   -2.5,-2.0,-1.5,-0.5,1.0,1.0,0.0,-0.5,-1.0,-0.5};
    double[] yp = {0.0,0.0,1.0,1.0,2.0,3.0,2.0,3.0,0.0,-0.5,-1.0,
                   -1.5,-2.0,-2.0,-1.5,-1.0,-0.5,-1.0,-1.0,-0.5};
    for(int i=0;i<100000;i++) {
      if (Pnpoly.pnpoly(npol,xp,yp,0.5,0.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,0.5,1.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-0.5,1.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,0.75,2.25)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,0.0,2.01)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-0.5,2.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-1.0,-0.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-1.5,0.5)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-2.25,-1.0)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,0.5,-0.25)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,0.5,-1.25)) count++;
      if (Pnpoly.pnpoly(npol,xp,yp,-0.5,-2.5)) count++;
    }
    System.out.println("count " + count);
  }
}
```

Figure 5. Java version of pnpoly. The Java language is a small, simple, object-oriented programming language chiefly designed and implemented by James Gosling (a Canadian born CMU graduate). The Java language is designed to allow a Web user to access programs. A click in a Java-capable browser can retrieve a program to be executed on the local machine. This program can then compute screen presentations and interact with the user. In appearance, the Java language looks very much like a restricted subset of C++. Because of security issues Java is designed to prevent dangerous

coding practices, for example, array accesses and type casts that cannot be checked for correctness at compile time are checked at run time, and the runtime system provides automatic garbage collection. Compiled Java programs are subjected to a verification process before execution by a browser. The verifier ensures that the compiled code in fact satisfies the Java type safety constraints and contains the required code for runtime checks. The Java language explicitly supports multithreaded programming and synchronized access to shared data. Standard Java libraries support text I/O, simple two-dimensional graphics, user interface devices (such as windows and buttons), and network access.

```
def pnpoly(npol, xp, yp, x, y):
    c = 0
    j = npol-1
    for i in range(npol):
            if ((yp[i] <= y < yp[j] or yp[j] <= y < yp[i]) and
                (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i])):
                c = not c
            j = i
    return c

def pnpolytest():
    count=0
    npol=20
     xp= [0.0,1.0,1.0,0.0,0.0,1.0,-.5,-1.0,-1.0,-2.0,-2.5,-2.0,-1.5,-.5,1.0,1.0,0.0,-.5,-1.0,-.5]
     yp= [0.0,0.0,1.0,1.0,2.0,3.0,2.0,3.0,0.0,-.5,-1.0,-1.5,-2.0,-2.0,-1.5,-1.0,-.5,-1.0,-1.0,-.5]
    for i in range(100000):
            if (pnpoly(npol,xp,yp,0.5,0.5)): count=count+1
            if (pnpoly(npol,xp,yp,0.5,1.5)): count=count+1
            if (pnpoly(npol,xp,yp,-0.5,1.5)): count=count+1
            if (pnpoly(npol,xp,yp,0.75,2.25)): count=count+1
            if (pnpoly(npol,xp,yp,0.0,2.01)): count=count+1
            if (pnpoly(npol,xp,yp,-0.5,2.5)): count=count+1
            if (pnpoly(npol,xp,yp,-1.0,-0.5)): count=count+1
            if (pnpoly(npol,xp,yp,-1.5,0.5)): count=count+1
            if (pnpoly(npol,xp,yp,-2.25,-1.0)): count=count+1
            if (pnpoly(npol,xp,yp,0.5,-0.25)): count=count+1
            if (pnpoly(npol,xp,yp,0.5,-1.25)): count=count+1
            if (pnpoly(npol,xp,yp,-0.5,-2.5)): count=count+1
    print 'count ', count
```

Figure 6. Python version of pnpoly. The following introduction to Python is lifted from its web site <www.python.org>: Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC, STDWIN). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface. The Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Mac, Amiga, etc. Python was developed over the past five years at CWI in Amsterdam by Guido van Rossum.

```
sub pnpoly {
    my($npol, $xxp, $yyp, $x, $y) = @_;
    my($j, $c, $i) = ($npol - 1, 0, 0);
    for ($i = 0; $i < $npol; $i++) {
            if ((($$yyp[$i] <= $y) && ($y < $$yyp[$j])) ||
                (($$yyp[$j] <= $y) && ($y < $$yyp[$i]))) &&
                ($x < (($$xxp[$j] - $$xxp[$i]) *
                        ($y - $$yyp[$i]) /
                        ($$yyp[$j] - $$yyp[$i]) + $$xxp[$i]))) {
                $c = !$c;}
            $j = $i;
    }
    $c;
}

sub pnpolytest {
    local($npol, $count, @xp, @yp);
    $npol=20;
    $count=0;
     @xp = (0.0,1.0,1.0,0.0,0.0,1.0,-.5,-1.0,-1.0,-2.0,
               -2.5,-2.0,-1.5,-.5,1.0,1.0,0.0,-.5,-1.0,-.5);
     @yp = (0.0,0.0,1.0,1.0,2.0,3.0,2.0,3.0,0.0,-.5,
               -1.0,-1.5,-2.0,-2.0,-1.5,-1.0,-.5,-1.0,-1.0,-.5);
    for($ii = 0; $ii < 100000; $ii++) {
            if (pnpoly($npol,\@xp,\@yp,0.5,0.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,0.5,1.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-0.5,1.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,0.75,2.25)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,0.0,2.01)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-0.5,2.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-1.0,-0.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-1.5,0.5)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-2.25,-1.0)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,0.5,-0.25)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,0.5,-1.25)) {$count++};
            if (pnpoly($npol,\@xp,\@yp,-0.5,-2.5)) {$count++};
    }
    print "\n count ", $count, "\n";
}
```

Figure 7. Perl version of pnpoly. The following introduction to Perl is summarized from its web site <www.perl.com>: Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). It combines some of the best features of C, sed, awk, and sh, so people familiar with those languages should have little difficulty with it. Expression syntax corresponds quite closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data--if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the hash tables used by associative arrays grow as necessary to prevent

degraded performance. Perl uses sophisticated pattern matching techniques to scan large amounts of data very quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like associative arrays (where dbm is available). If you have a problem that would ordinarily use sed or awk or sh, but it exceeds their capabilities or must run a little faster, and you don't want to write the thing in C, then Perl may be for you. Perl was created by Larry Wall.

```
(define pnpoly
  (lambda (npol xp yp x y)
    (let ((c #f)
          (j (- npol 1)))
      (do ((i 0 (+ i 1)))
          ((= i npol) c)
        (if (and (or (and (<= (vector-ref yp i) y) (< y (vector-ref yp j)))
                     (and (<= (vector-ref yp j) y) (< y (vector-ref yp i))))
                 (< x (+ (vector-ref xp i)
                         (/ (* (- (vector-ref xp j) (vector-ref xp i)) (- y (vector-ref yp i)))
                            (- (vector-ref yp j) (vector-ref yp i))))))
            (set! c (not c)))
        (set! j i))
      )))

(define pnpolytest
  (lambda ()
    (let ((npol 20)
          (count 0)
          (xp '#1(0.0 1.0 1.0 0.0 0.0 1.0 -0.5 -1.0 -1.0 -2.0
                  -2.5 -2.0 -1.5 -0.5 1.0 1.0 0.0 -0.5 -1.0 -0.5))
          (yp '#1(0.0 0.0 1.0 1.0 2.0 3.0 2.0 3.0 0.0 -0.5 -1.0
                  -1.5 -2.0 -2.0 -1.5 -1.0 -0.5 -1.0 -1.0 -0.5)))
      (do ((i 0 (+ i 1)))
          ((= i 100000))
        (if (pnpoly npol xp yp 0.5 0.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp 0.5 1.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -0.5 1.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp 0.75 2.25) (set! count (+ count 1)))
        (if (pnpoly npol xp yp 0.0 2.01) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -0.5 2.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -1.0 -0.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -1.5 0.5) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -2.25 -1.0) (set! count (+ count 1)))
        (if (pnpoly npol xp yp 0.5 -0.25) (set! count (+ count 1)))
        (if (pnpoly npol xp yp 0.5 -1.25) (set! count (+ count 1)))
        (if (pnpoly npol xp yp -0.5 -2.5) (set! count (+ count 1)))
        )
      (display "count ") (write count) (newline)
      count)))
```

Figure 8. Scheme version of pnpoly. Scheme is a dialect of Lisp invented in 1975 by Guy L. Steele Jr. and Gerald J. Sussman. There are many implementations of Scheme to choose from including native machine code compilers. I used Guile 1.0 because it is an effort of GNU to compete with

Tcl and other scripting languages. More on Guile summarized from < www.gnu.ai.mit.edu>: GUILE, GNU's Ubiquitous Intelligent Language for Extension, is a library that implements the Scheme language plus various convenient facilities. It's designed so that you can link it into an application or utility to make it extensible. Libraries that provide an interpreter for extensibility are not new but most of them implement ``scripting languages'' that were not designed to be as powerful as a real programming language. The big advantage of Guile is that it allows support for multiple languages. This is because Scheme is powerful enough that other languages can conveniently be translated into it. One translator, CTAX, is implemented which understands simple C-like language. There are plans to have translators for Perl, Python, Tcl, REXX and Emacs Lisp. Guile supports an interface to Tk, so it can be used for graphical programs.

```
proc pnpoly {npol xp yp x y} {
   set c 0
   set j [expr $npol-1]
   for {set i 0} {$i < $npol} {incr i 1} {
      if {(((([lindex $yp $i] <= $y) && ($y < [lindex $yp $j])) ||
               (([lindex $yp $j] <= $y) && ($y < [lindex $yp $i])))
              && ($x < (([lindex $xp $j] - [lindex $xp $i]) * ($y - [lindex $yp $i]) /
                       ([lindex $yp $j] - [lindex $yp $i]) + [lindex $xp $i])))} {
            set c [expr !$c]
         }
         set j $i
   }
   return $c
}

proc pnpolytest {} {
   set count 0
   set npol 20
   set xp {0.0 1.0 1.0 0.0 0.0 1.0 -0.5 -1.0 -1.0 -2.0
         -2.5 -2.0 -1.5 -0.5 1.0 1.0 0.0 -0.5 -1.0 -0.5}
   set yp {0.0 0.0 1.0 1.0 2.0 3.0 2.0 3.0 0.0 -0.5
         -1.0 -1.5 -2.0 -2.0 -1.5 -1.0 -0.5 -1.0 -1.0 -0.5}
   for {set i 0} {$i<100000} {incr i} {
         if {[pnpoly $npol $xp $yp 0.5 0.5]} {incr count}
         if {[pnpoly $npol $xp $yp 0.5 1.5]} {incr count}
         if {[pnpoly $npol $xp $yp -0.5 1.5]} {incr count}
         if {[pnpoly $npol $xp $yp 0.75 2.25]} {incr count}
         if {[pnpoly $npol $xp $yp 0.0 2.01]} {incr count}
         if {[pnpoly $npol $xp $yp -0.5 2.5]} {incr count}
         if {[pnpoly $npol $xp $yp -1.0 -0.5]} {incr count}
         if {[pnpoly $npol $xp $yp -1.5 0.5]} {incr count}
         if {[pnpoly $npol $xp $yp -2.25 -1.0]} {incr count}
         if {[pnpoly $npol $xp $yp 0.5 -0.25]} {incr count}
         if {[pnpoly $npol $xp $yp 0.5 -1.25]} {incr count}
         if {[pnpoly $npol $xp $yp -0.5 -2.5]} {incr count}
   }
   puts “Count $count”
   return $count
}
```

Figure 9. Tcl version of pnpoly. The following description of Tcl is summarized from the web site <sunscript.sun.com>: Tcl's purpose is to "glue" existing components together. It is useful for wrapping program components together and adding the GUI with the Tk GUI library. Tcl tends to be less efficient, in part because it uses an interpreter instead of a native compiler and because its basic components are chosen for power and ease of use rather than an efficient mapping onto the underlying hardware. For example, Tcl tends to use variable-length strings in situations where a system programming language would use a binary value that fits in a single machine word, and Tcl often uses hash tables instead of indexed arrays. Fortunately, the performance of Tcl isn't usually a major issue. It is the ability to easily add a Tcl interpreter to your application that sets it apart from other shells. Tcl fills the role of an extension language that is used to configure and customize applications. There is no need to invent a command language for your new application, or struggle to provide some sort of user-programmability for your tool. Instead, by adding a Tcl interpreter, you structure your application as a set of primitive operations that can be composed by a script to best suit the needs of your users. It also allows other programs to have programmatic control over your application, leading to suites of applications that work well together. Tcl was created by John Ousterhout.