

A Peek on Numerical Programming in Perl and Python

E. Christopher Dyken*

Abstract

In this note we peek at the capabilities for numerical programming in the two high-level scripting languages Perl and Python, both having numerical libraries providing increased efficiency for iterating over large arrays of data. We do a superficial investigation on the efficiency of both languages with and without using numerical libraries.

Using high-level languages can quite often increase the speed substantially for software development. High-level scripting languages makes rapid prototyping of new ideas and concepts possible with a minimal amount of effort. However, one crux of numerical software is efficient traversal of large amounts of data. High-level languages per se has a deficiency in the sense that such operations are notoriously slow. To overcome this, both Perl and Python has add-on libraries providing special data types that can hold large chunks of data efficient, in regard to both memory usage as well as access speed.

Given one can formulate one's algorithm as element-by-element operations over n -dimensional arrays, both Perl and Python provide functionality with performance comparable to compiled C code. Numerical Python[1] (NumPy) provides fast multi-dimensional capabilities to Python. A new implementation, numarray[2], is available as well. Perl has its own counterpart to NumPy, the Perl Data Language[3] (PDL). PDL brings number-crunching capabilities to Perl as well as an interactive shell and other goodies. To shed some light on the numerical capabilities of high-level scripting languages, we have implemented the trapezoidal quadrature rule in Python and Perl, both with and without add-on libraries, as well as in standard C for reference.

We used a formulation of the quadrature rule which takes advantage of the type of element-by-element operations over arrays that are optimized by the add-on libraries,

$$\int_a^b f(x) dx = h \left(\sum_{i=0 \dots N} f(a + h \cdot i) - \frac{f(a) + f(b)}{2} \right),$$

where $h = (b - a)/(N - 1)$ and N is the number of samples. The idea is to make each implementation have a code path as similar to each other as possible.

We used three different integrands f_1 , f_2 and f_3 , defined as

*Centre of Mathematics for Applications, University of Oslo

$$\begin{aligned}
f_1(x) &= x, \\
f_2(x) &= x^2 \quad \text{and} \\
f_3(x) &= \cos(x^2) \sin(x),
\end{aligned}$$

where f_1 and f_2 are simple polynomials, while f_3 is a bit more complex involving some trigonometric functions.

From this we made the following implementations:

A **Standard C** implementation was used for reference, compiled with GNU gcc version 3.2.2 using no optimizations whatsoever. The source code can be found in Appendix A.1.

A **Optimized C** implementation, identical to the standard C version, but compiled with `-O3` optimization.

A **Standard Python** implementation was run using Python version 2.3.3. The source code can be found in Appendix A.2.

A **Python with NumPy** implementation was run using Python version 2.3.3 and NumPy version 23.1. The source code can be found in Appendix A.3.

A **Python with numarray** implementation was run using Python version 2.3.3 and numarray version 0.9. The source code can be found in Appendix A.4.

A **Standard Perl** implementation was run using Perl version 5.8.0. The source code can be found in Appendix A.5.

A **Perl with PDL** implementation was run using Perl version 5.8.0 and PDL version 2.4.1. The source code can be found in Appendix A.6.

In order to make the tests as fair as possible, the `*nix` system call `gettimeofday` was used since it is available in all languages used in this test. This system call returns the number of seconds and microseconds of wall-clock time since the epoch. It is questionable to use the wall-clock time and not the processor time spent. However, since all tests were run on the same system under the same levels of system load (virtually none) several times, it should give a fair indication.

The experiment was carried out by performing each test 100 times for each combination of sampling density, programming language and integrand on a Intel Pentium 4 running at 3.20 GHz with 2 GB of RAM. The performance of each program is plotted in Figures 1, 2 and 3 where average execution time is plotted versus number of samples.

From the results, we see that plain vanilla Python and Perl doesn't present a performance comparable to C. This comes as no surprise. However, by using the extensions, the picture changes. Both numarray and PDL exhibits performance relatively close to C.

NumPy shows a worse performance on the simplest function, but this gap is reduced when the integrand becomes more complex. Test functions 1 and 2 is composed

of only simple floating point arithmetic, and thus most time is spent in the surrounding control structures. However, test function 3 uses `cos` and `sin`, which is more costly and dominates the execution time, which in turn results in a reduction of the gap between the various implementations.

Thus, from this, we conclude that, given your problem can be formulated mostly as element-by-element operations on arrays, Python or Perl with add-on libraries could be a viable alternative to the more traditional approach of C or C++, an alternative that is definitely worth investigating.

References

- [1] The Numerical Python home page,
<http://www.numpy.org/>
- [2] The numarray home page,
http://www.stsci.edu/resources/software_hardware/numarray
- [3] The Perl Data Language home page,
<http://pdl.perl.org/>

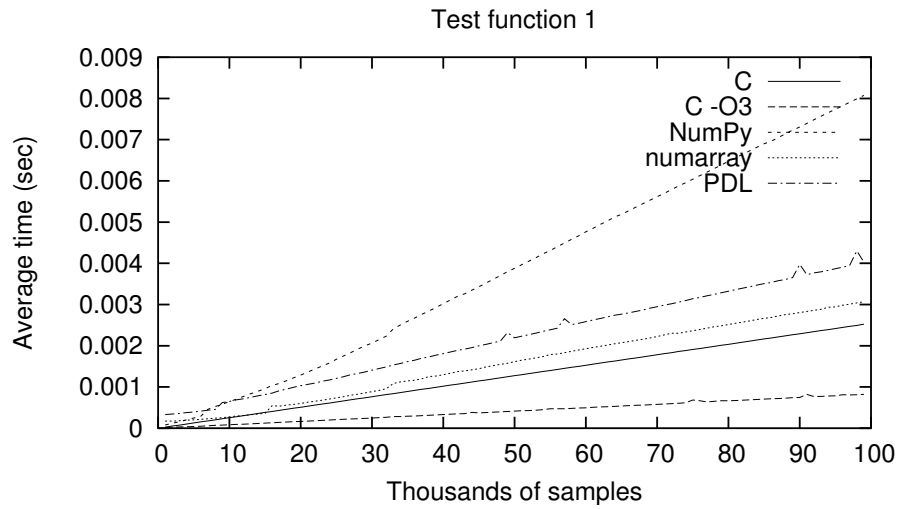
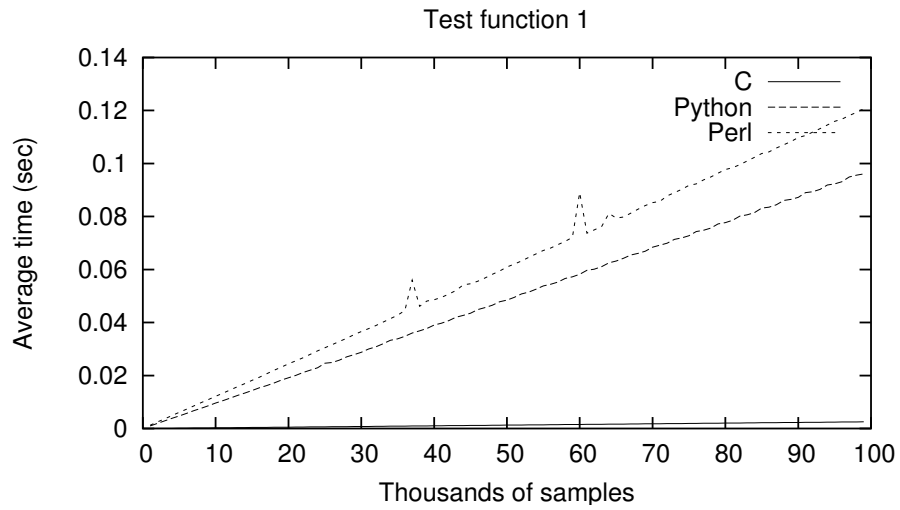


Figure 1: Numerical integration of $f(x) = x$.

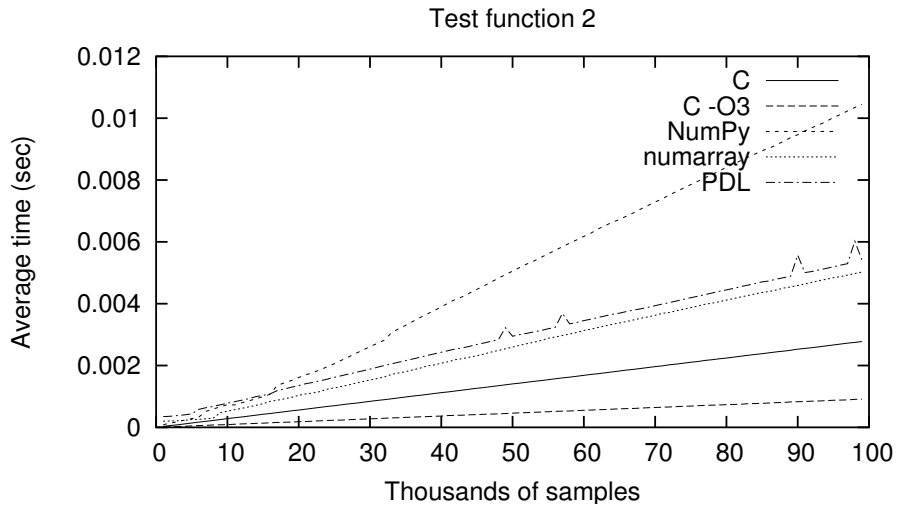
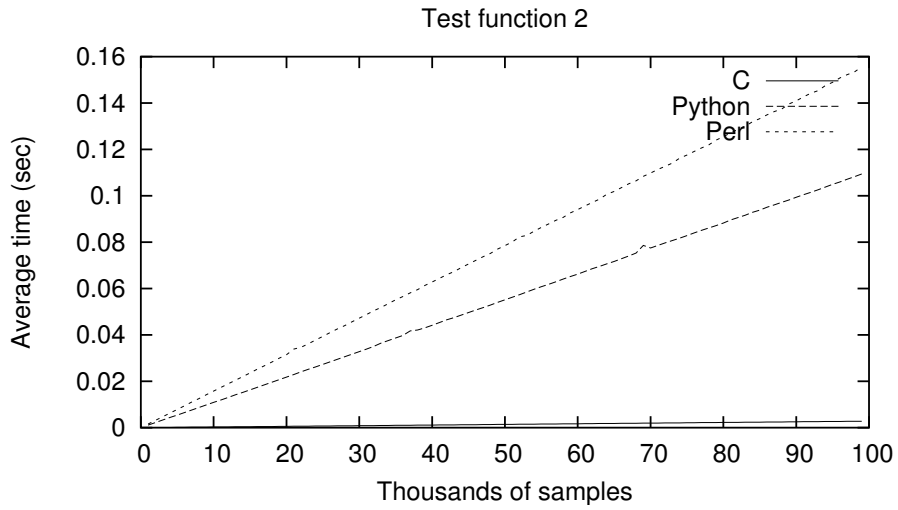


Figure 2: Numerical integration of $f(x) = x^2$.

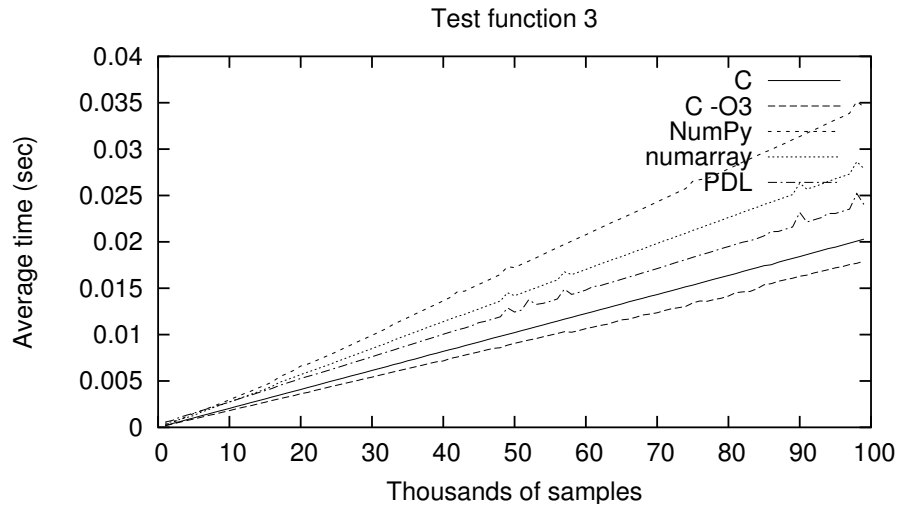
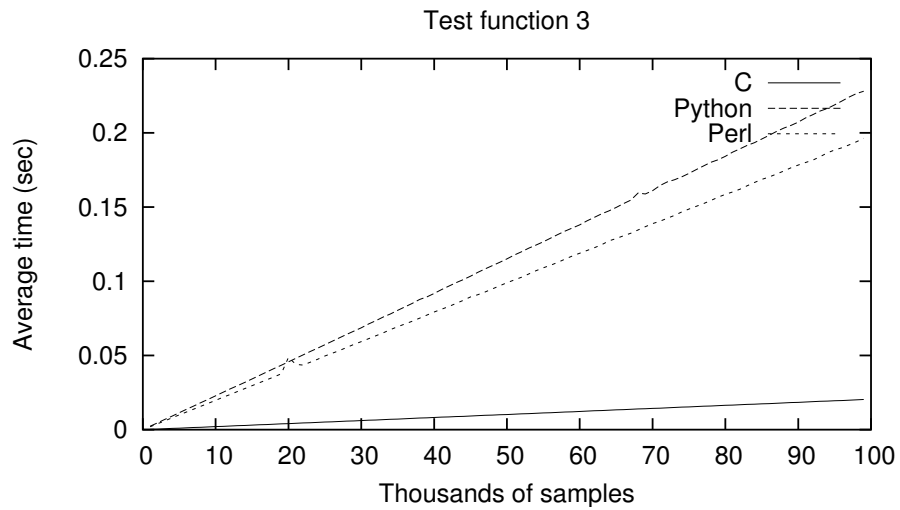


Figure 3: Numerical integration of $f(x) = \cos(x^2) \sin(x)$.

A Source code

A.1 C implementation

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

double integrate(double a, double b, double (*f)(double), int N) {
    double h = (b-a)/((double)N-1.0);
    double sum = 0.0;
    int i;
    for(i=0; i<N; i++)
        sum += f(a+h*(double)i);
    return h*(sum - 0.5*f(a) - 0.5*f(b));
}

double f1(double t) { return t; }
double f2(double t) { return t*t; }
double f3(double t) { return cos(t*t)*sin(t); }

int main(int argc, char **argv) {
    struct timeval begin, end;
    double (*f)(double t), res, a=0.0, b=1.0;
    int N, M, i;

    if(argc != 3) return -1;
    switch(atoi(argv[1])) {
    case 0: f = f1; break;
    case 1: f = f2; break;
    case 2: f = f3; break;
    }

    M = atoi(argv[2]);
    for(N=1000; N<100000; N+=1000) {
        gettimeofday(&begin, NULL);
        for(i=0; i<M; i++)
            res = integrate(0,1,f,N);
        gettimeofday(&end, NULL);
        printf("%f %f\n", res,
            ((end.tv_sec - begin.tv_sec) +
            1.0E-6*(end.tv_usec - begin.tv_usec))/(double)M);
    }
}
```

A.2 Python implementation

```
import sys;
from math import *;

def integrate(a,b,f,n):
    h = (b-a)/(n-1.0);
    sum = 0.0;
    for i in range(n):
        sum += f(a+(i)*h);
    return h*(sum - 0.5*f(a) -0.5*f(b));

if __name__ == '__main__':
    from time import time;

    def f1(x): return x;
    def f2(x): return x*x;
    def f3(x): return cos(x*x)*sin(x);

    if len(sys.argv) == 3:
        if int(sys.argv[1]) == 0:
            f = f1
        elif int(sys.argv[1]) == 1:
            f = f2
        else:
            f = f3

    a = 0
    b = 1
    M = int(sys.argv[2])

    for N in range(1000, 100000, 1000):

        begin = time()
        for i in range(M):
            res = integrate(a,b,f,N)
        avg = (time()-begin)/M

        print "%f %f" % (res, avg)
```


A.3 NumPy implementation

```
import sys;
from Numeric import *;

def integrate(a,b,f,n):
    h = (b-a)/(n-1.0);
    vals = f(h*arange(n));
    return h*(sum(vals) - 0.5*vals[0] - 0.5*vals[len(vals)-1]);

if __name__ == '__main__':
    from time import time;

    def f1(x): return x;
    def f2(x): return x*x;
    def f3(x): return cos(x*x)*sin(x);

    if len(sys.argv) == 3:

        if int(sys.argv[1]) == 0:
            f = f1
        elif int(sys.argv[1]) == 1:
            f = f2
        else:
            f = f3

        a = 0
        b = 1
        M = int(sys.argv[2]);

        for N in range(1000, 100000, 1000):

            begin = time();
            for i in range(M):
                res = integrate(a,b,f,N);
            avg = (time()-begin)/M;

            print "%f %f" % (res, avg)
```

A.4 Numarray implementation

```
import sys;
from numarray import *;

def integrate(a,b,f,n):
    h = (b-a)/(n-1.0);
    vals = f(h*arange(n));
    return h*(sum(vals) - 0.5*vals[0] - 0.5*vals[len(vals)-1]);

if __name__ == '__main__':
    from time import time;

    def f1(x): return x;
    def f2(x): return x*x;
    def f3(x): return cos(x*x)*sin(x);

    if len(sys.argv) == 3:

        if int(sys.argv[1]) == 0:
            f = f1
        elif int(sys.argv[1]) == 1:
            f = f2
        else:
            f = f3

        a = 0
        b = 1
        M = int(sys.argv[2]);

        for N in range(1000, 100000, 1000):

            begin = time()
            for i in range(M):
                res = integrate(a,b,f,N)
            avg = (time()-begin)/M

            print "%f %f" % (res, avg)
```

A.5 Perl implementation

```
#!/usr/bin/perl
use strict;
use warnings;
use Time::HiRes qw(gettimeofday tv_interval);

sub integrate {
    my ($a, $b, $f_ref, $n) = @_;
    my $h = ($b-$a)/($n-1.0);
    my $sum = 0;
    for my $i (0..$n-1) {
        $sum += $f_ref->($a + $h*$i);
    }
    return $h*( $sum - 0.5*$f_ref->($a) -0.5*$f_ref->($b));
}

die unless @ARGV == 2;

my @F      = ( sub {shift; },
               sub {my $x=shift; $x*$x; },
               sub {my $x=shift; cos($x*$x)*sin($x)} );
my $f      = $F[shift];
my $M      = shift;
my ($a, $b) = (0.0, 1.0);
my ($res, $begin, $end, $avg);

for (my $N=1000; $N<100000; $N+=1000) {

    $begin = [gettimeofday()];
    for my $i (1..$M) {
        $res = integrate($a,$b,$f,$N);
    }
    $avg = tv_interval($begin)/$M;

    print "$res $avg\n";
}
```

A.6 PDL implementation

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use Time::HiRes qw(gettimeofday tv_interval);

sub integrate {
    my ($a, $b, $f_ref, $n) = @_;
    my $h = ($b-$a)/($n-1.0);
    my $vals = $f_ref->($h*xvals($n));
    return $h*(sum($vals) - 0.5*$vals->index(0) - 0.5*$vals->index($n-1));
}

die unless @ARGV == 2;

my @F      = ( sub {shift; },
               sub {my $x=shift; $x*$x; },
               sub {my $x=shift; cos($x*$x)*sin($x)} );
my $f      = $F[shift];
my $M      = shift;
my ($a, $b) = (0.0, 1.0);
my ($res, $begin, $avg);

for(my $N=1000; $N<100000; $N+=1000) {

    $begin = [gettimeofday()];
    for my $i (1..$M) {
        $res = integrate($a,$b,$f,$N);
    }
    $avg = tv_interval($begin)/$M;

    print "$res $avg\n";
}
```