# Solution to Assignment 0

1. **Number of cycles to execute one iteration of the loop**

If $T$ is the runtime of the code in s, $n$ is the number of slices, and $f$ is the clock speed in cy/s, then the number of cycles per iteration is:

$$c = \frac{T}{n} \times f$$

Code:

```
double f = 2.2e9;   // clock frequency in cy/s
int n = 1000000000; // # of slices
double delta_x = 1./n, x, sum = 0., ct, wcs, wce, Pi;
wcs = getTimeStamp();
for (int i=0; i < n; i++) {
  x = (i+0.5)*delta_x;
  sum += (4.0 / (1.0 + x * x));
}
wce = getTimeStamp(); // T = wce-wcs
Pi = sum * delta_x;
printf("Pi=%.15lf in %.3lf s -> %.2lf cy/it\n",Pi,wce-wcs, \
           (wce-wcs)/n*f);
```

I have set the clock speed to 2.2 GHz as shown in the intro session (`f=2.2` feature in qsub).

Compilation and run:

```
$ icc -O3 -xHost div.c timing.c
$ ./a.out
Pi=3.141592653589828 in 3.189 s -> 7.02 cy/it
```

→ The loop executes in about 7 cycles per iteration.

2. **Appropriate performance metrics**

The loop body has six “flops”: 3 ADDs, 2 MULTs, and 1 DIV. Naively, one might be tempted to define performance as

$$P = \frac{6\,flops \times n}{T}.$$

However, it is unknown what the compiler did to this code; it could, e.g., expand the product in the first line of the body. Also it is unknown whether the target architecture has a divide unit at all -- a library could be called for that. Finally, the `i` in the first line of the body must be converted to floating point. Is that a flop, too? Flop/s is thus not a good metric.

Inverse runtime would be the next candidate, but it would change if the number of slices changes. A performance metric that changes with the problem size in a trivial way is undesirable, since important effects (such as a drop in efficiency when the problem size is very small) are hidden by the trivial dependence.

Hence, I consider "iterations per second" (it/s) or "iterations per cycle" (it/cy) to be reasonable performance metrics here.

### 3. Single precision

Code (necessary changes highlighted):

```
double f = 2.2e9;   // clock frequency in cy/s
int n = 1000000000; // # of slices
float delta_x = 1.f/n, x, sum = 0.f, ct, wcs, wce, Pi;
wcs = getTimeStamp();
for (int i=0; i < n; i++) {
  x = (i+0.5f)*delta_x;
  sum += (4.0f / (1.0f + x * x));
}
wce = getTimeStamp(); // T = wce-wcs
Pi = sum * delta_x;
printf("Pi=%.7f in %.3lf s -> %.2lf cy/it\n",Pi,wce-wcs, \
           (wce-wcs)/n*f);
```

Result:

```
$ ./a.out
Pi=1.0737418 in 0.512 s -> 1.13 cy/it
```

The code is over 6 times faster, which lets us speculate about how fast a divide can actually be on this CPU in different precisions. Obviously, single-precision divides are much faster.

The result is far off from $\pi$. This is because we are adding up $10^9$ positive numbers, all between 2 and 4, but our float variable for accumulation has only 7 decimal digits of accuracy. Most of the terms in the sum are thus lost (partially or entirely), and the result is far smaller than it should be.