

will wait for the function to complete on the remote process, and produce the function's result. The following script creates three new processes,⁴ and instructs process #4 to factor a large number:⁵

```
addprocs(3)
using Primes.factor
@assert nprocs() == 4
future = remotecall(factor, 4, 21883298135690819)
@assert isa(future, Future)
factors = fetch(future)
@assert factors == Dict{234711901=>1,93234719=>1}
```

The `@spawn` macro is often used in place of `remotecall`; it not only takes advantage of a clearer syntax (since macros take unevaluated expressions as arguments), but it chooses a process for you:

```
addprocs(3)
using Primes.factor
ref = @spawn factor(21883298135690819)
factors = fetch(ref)
@assert factors == Dict{234711901=>1,93234719=>1}
```

Between the remote call and fetching the result, the calling process is free to do other work.

A common example of parallel programming across multiple cores computes an approximation of π by generating random points in the square $(0,0)\dots(1,1)$. Counting the number of random points within the inscribed arc (the points (x,y) for which $x^2 + y^2 \leq 1$ in Figure 6.3) divided by the total number of points within the square approximates the value $\frac{\pi}{4}$. Let's generate a million random points in parallel on three cores:

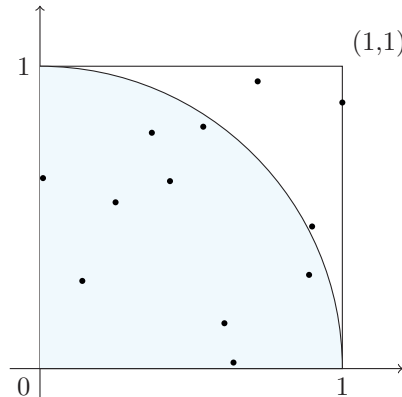


Figure 6.3 Approximation of $\frac{\pi}{4}$

⁴Calling `addprocs` with an integer argument creates processors on the local machine to take advantage of multiple cores. There are other variants of `addprocs` to add processors on remote machines either via user-host-port strings or cluster managers; we will not cover these alternatives in this text.

⁵The `factor` function resides in the package `Primes`. You must first install this package by executing `Pkg.add("Primes")` in an interactive shell, or `julia -e 'Pkg.add("Primes")'` from the command line.