Lecture 26: Numerical integration/quadrature

Numerical integration (which is also called *quadrature*) refers to approximating integrals by sums. A *quadrature rule* is the pair of *nodes* $(x_1, \ldots, x_n)$ and *weights* $(w_1, \ldots, w_n)$ so that

$$\int_a^b f(x)dx \approx \sum_{k=1}^n w_k f(x_k)$$

In this lecture, we will focus on integrals over $[0, 1]$:

$$\int_0^1 f(x)dx$$

We will also use $[0, 2\pi)$ when we wish to emphasize periodicity:

$$\int_0^{2\pi} f(\theta)d\theta$$

Right-hand rule

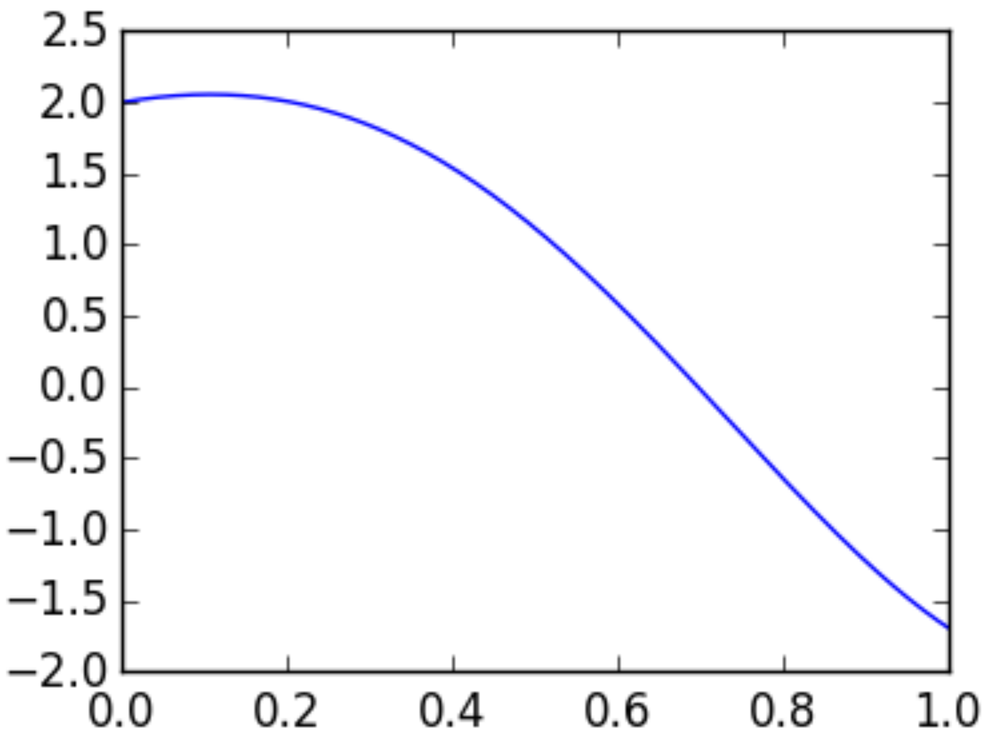
We begin with the simplest method the *right-hand rule*. This is the choice of nodes $x_k = kh$ and weights $w_k = h$ for $h \triangleq \frac{1}{n}$.

This rule corresponds to integrating rectangles which match the right-hand value in each panel, consisting of x between x_k and x_{k+1}. We will depict this now.

Consider a simple function on $[0, 1]$:

In [1]:

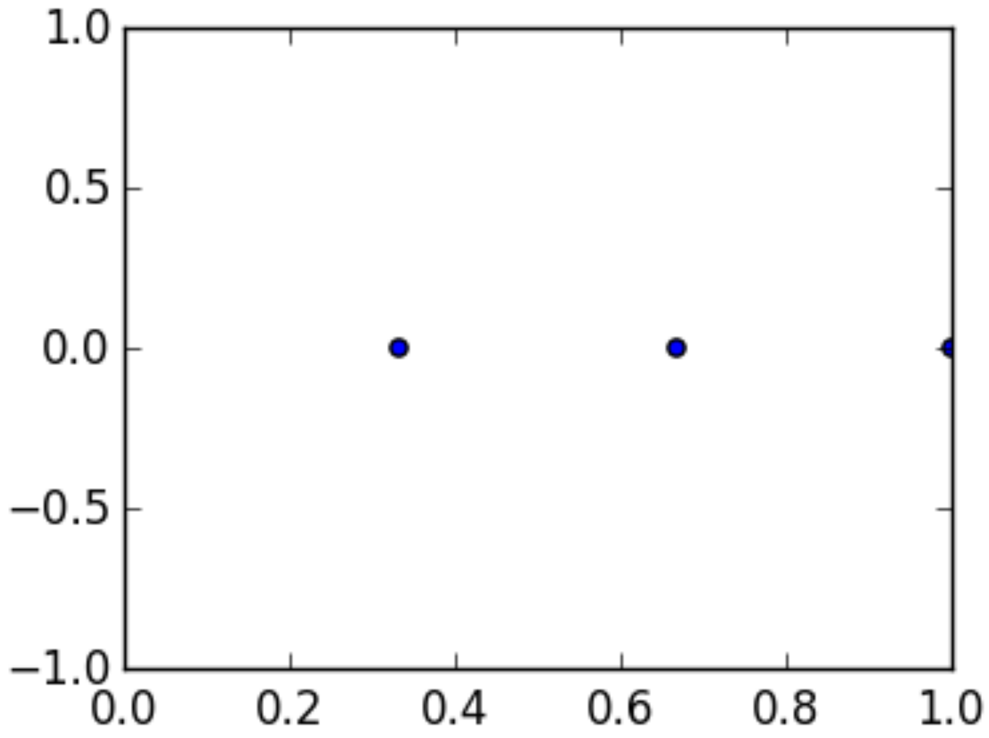
```
using PyPlot



f=x->cos(3x).*exp(x)+1

g=linspace(0.,1.,1000)  # the plotting grid, much finer than the quadrature
nodes
plot(g,f(g));
```



Our quadrature nodes are $(x_1, \ldots, x_n) = [h, 2h, \ldots, 1]$, as plotted here:

In [2]:

```
n=3
h=1/n
x=linspace(h,1.,n)    # creates n evenly spaced nodes between h and 1

scatter(x,zeros(x))
axis([0.,1.,-1.,1.]);
```



```
/usr/local/lib/python2.7/site-packages/matplotlib/collections.py:
590: FutureWarning: elementwise comparison failed; returning scal
ar instead, but in the future will perform elementwise comparison
  if self._edgecolors == str('face'):
```

To plot the rectangles, we create a function `s(vals,x,t)` that plots a piecewise step function which equals `vals[k]` between `x[k-1]` and `x[k]`:

```
In [3]:
```

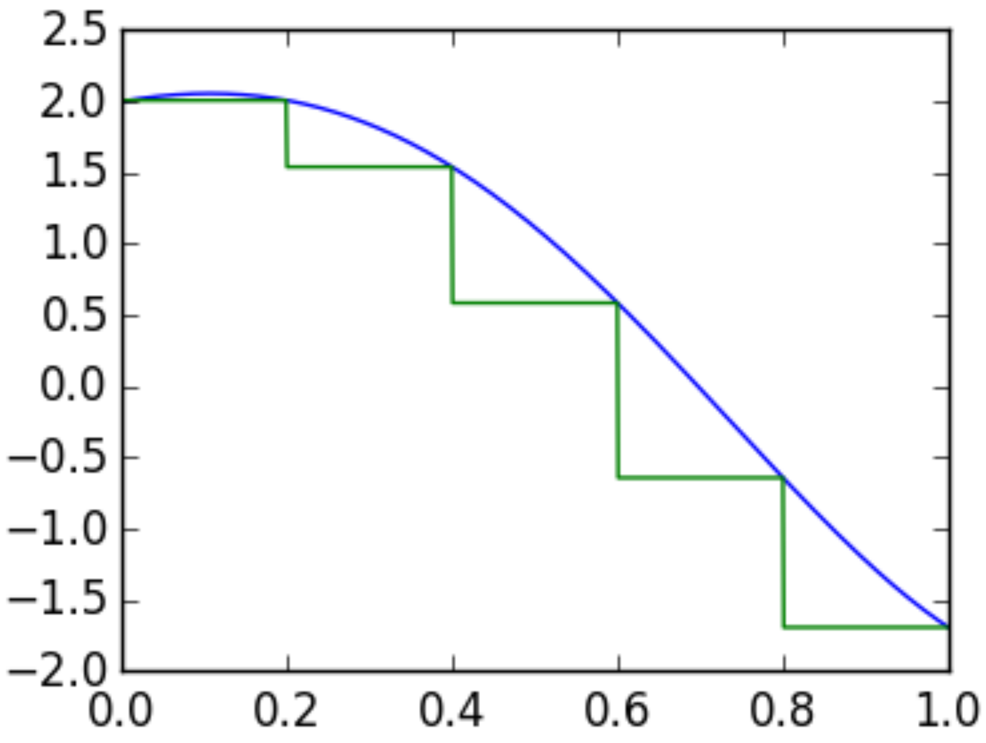
```julia
function s(vals::AbstractVector,x::AbstractVector,t::Number)
    n=length(x)

    if 0 ≤ t ≤ x[1]
        return vals[1]
    end

    for k=1:n-1
        # check each panel one by one
        if x[k] ≤ t ≤ x[k+1]
            return vals[k+1]
        end
    end

    return zero(typeof(vals[1]))  # return 0. of the same type as vals[k]
end
```

```
Out[3]:

s (generic function with 1 method)
```

We can plot the rectangle approximation versus the true function:

In [6]:

```
n=5
h=1/n
x=linspace(h,1.,n)


plot(g,f(g))
plot(g,Float64[s(f(x),x,t) for t in g])
```



Out[6]:

```
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x313244bd0>
```

The quadrature rule is given by the approximation

```
sum(f(x)*h)   # sum([v1,v2,v3]) = v1+v2+v3
```

We compare with an approximation of the true integrand, using `quadgk`:

In [7]:

```
ex,err=quadgk(f,0.,1.)

n=1000000
h=1/n
x=linspace(h,1.,n)

sum(f(x)*h)-ex
```

Out[7]:

```
-1.8455397103878113e-6
```

We see that our method is much slower than `quadgk`: one should not use this method in practice!

In [8]:

```
@time sum(f(x)*h)
```

```
  0.071964 seconds (169 allocations: 38.158 MB, 11.53% gc time)
```

Out[8]:

```
0.7459714791406495
```

In [9]:

```
@time quadgk(f,0.,1.)
```

```
  0.000036 seconds (165 allocations: 2.922 KB)
```

Out[9]:

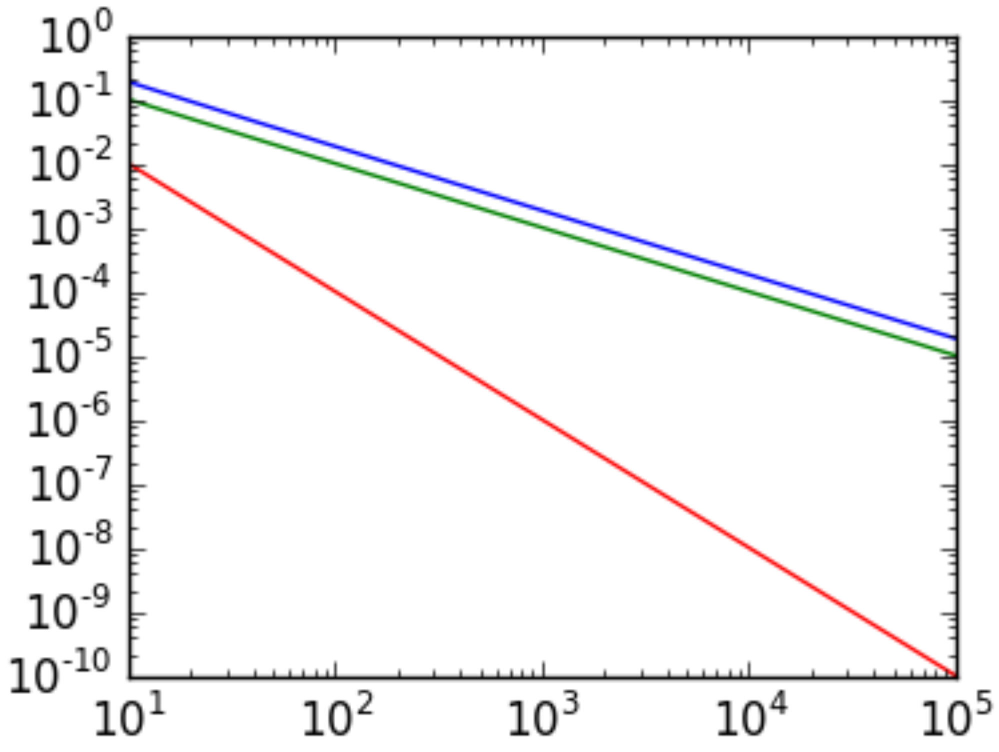
```
(0.7459733246803599,1.0242917625191694e-12)
```

We estimate the convergence rate using a `loglog` plot. The slope of a `loglog` plot tells us the rate α such that the error behaves like $O(n^{-\alpha})$.

Since the slopes match, the following plot is indication that the error decays like $O(n^{-1})$. We will prove this next lecture.

In [14]:

```
ns=round(Int,logspace(1,5,100))   # create 100 logarithmically spaced n's be
tween 10^1 and 10^5

err=zeros(length(ns))  # we will fill this Vector with the errors

for k=1:length(ns)
    n=ns[k]
    h=1/n
    x=linspace(h,1.,n)

    err[k]=abs(sum(f(x)*h)-ex    )
end

loglog(ns,err)    # The blue curve
loglog(ns,1./ns)  # The green curve.  Since the slope appears to match the g
reen curve, we
                  # conjecture that the blue curve is also O(1/n)
loglog(ns,1./ns.^2)  # The red curve. We also plot something which decays fa
ster so we
                     #  can see that the slope is clearly different;
```



Trapezium rule

The trapezium rule approximates by an affine function in each panel, which is then integrated exactly. This leads to the approximation

$$\int_0^1 f(x)dx \approx h\left(\frac{f(x_0)}{2} + \sum_{k=1}^{n-1} f(x_k) + \frac{f(x_n)}{2}\right)$$

where again $x_k = kh$. (Note: we will sometimes start our indices from zero when it simplifies notation. This is therefore an $n+1$ point quadrature rule.)

This expression follows since in each panel we have the affine approximation (using the first panel as example):

$$f(x) \approx f(0) + x\frac{f(h) - f(0)}{h}$$

which equals f at x_k and x_{k+1}. Integrating this approximation exactly gives us

$$\int_0^h \left(f(0) + x\frac{f(h) - f(0)}{h}\right) dx = f(0)h + \frac{h^2}{2}\frac{f(h) - f(0)}{h} = h\frac{f(0) + f(h)}{2}$$

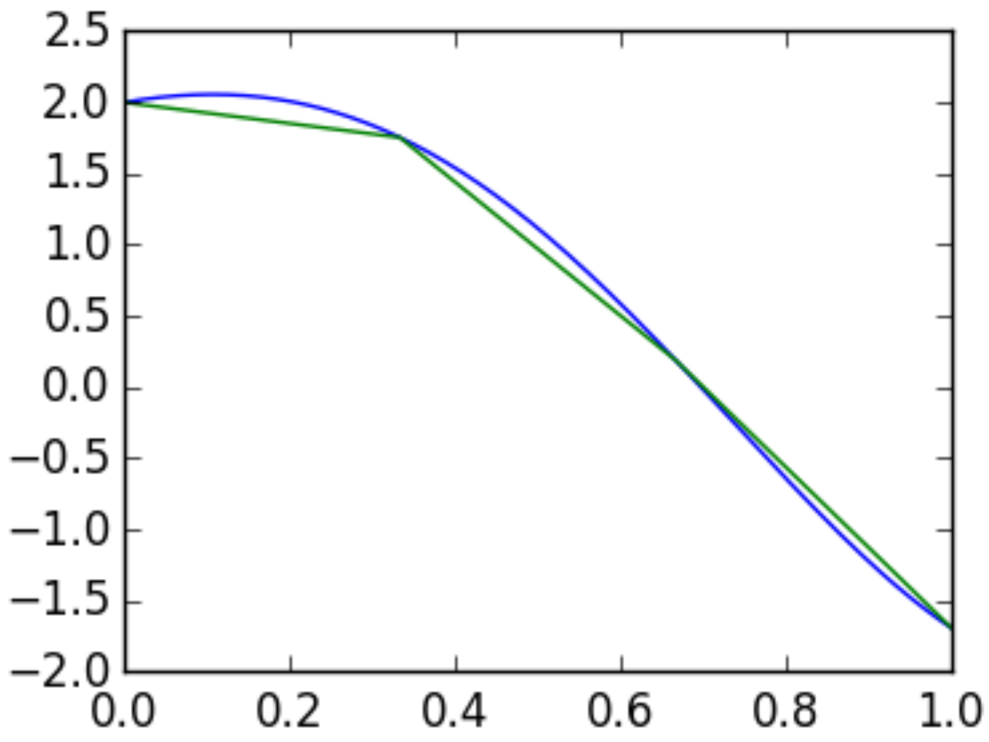
We thus have

$$\begin{aligned}\int_0^1 f(x)dx &\approx \sum_{k=1}^{n} \int_0^1 \left(f(x_{k-1}) + (x - x_{k-1})\frac{f(x_k) - f(x_{k-1})}{h}\right) dx \\ &= h\sum_{k=1}^{n} \frac{f(x_{k-1}) + f(x_k)}{2} \\ &= h\left(\frac{f(x_0)}{2} + \sum_{k=1}^{n-1} f(x_k) + \frac{f(x_n)}{2}\right)\end{aligned}$$

We will depict the approximation. Consider again a simple function. Now `plot` will plot the approximation by trapeziums:

```
In [17]:
```

```
plot(g,f(g))

n=3

h=1/n
x=linspace(0.,1.,n+1)  # the grid

plot(x,f(x));
```



The function `trap(f,n)` calculates the trapezoidal rule between 0 and 1 with `n` panels:

```
In [19]:
```

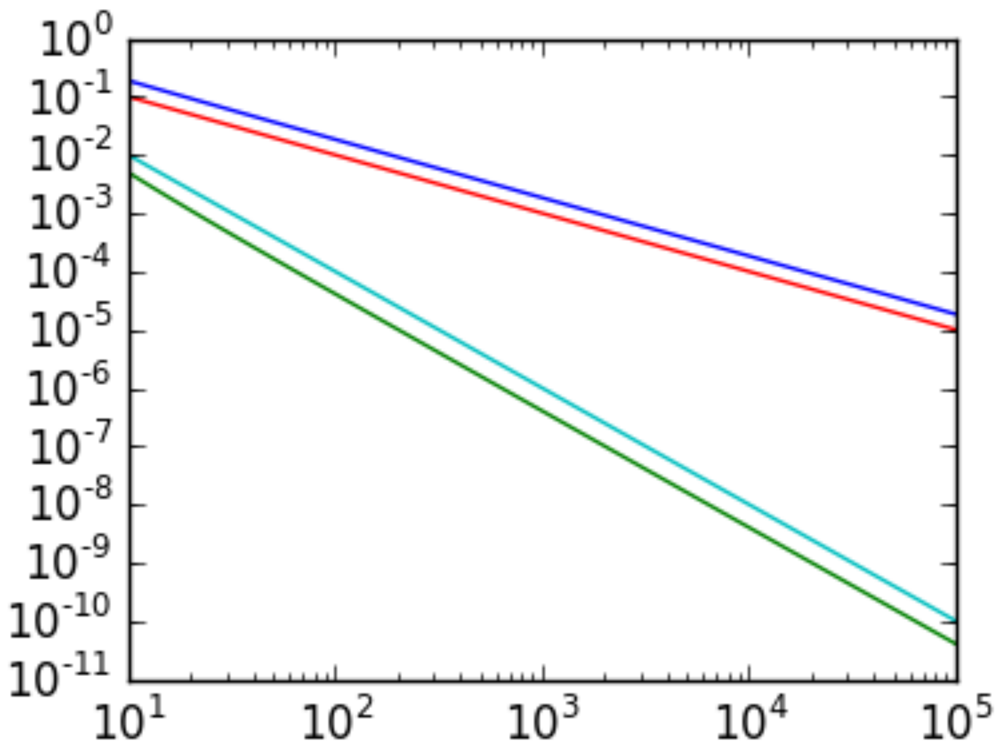
```
ex,err=quadgk(f,0.,1.)



n=1000

function trap(f,n)
    h=1/n
    x=linspace(0.,1.,n+1)

    v=f(x)  # assume f can be evaluated on Vectors.  Otherwise, use map(f,x)
    h/2*v[1]+sum(v[2:end-1]*h)+h/2*v[end]
end



trap(f,10000)-ex   # the error is smaller than before.
```

```
Out[19]:

-4.034908518413260 5e-9
```

Let's compare the error in `trap` to the right-hand rule. The following shows that the error is $O(n^{-2})$:

In [21]:

```
ns=round(Int,logspace(1,5,100))

err=zeros(length(ns))
errT=zeros(length(ns))

for k=1:length(ns)
    n=ns[k]
    h=1/n
    x=linspace(h,1.,n)

    err[k]=abs(sum(f(x)*h)-ex    )
    errT[k]=abs(trap(f,n-1)-ex    )
end



loglog(ns,err)       # blue curve
loglog(ns,errT)      # green curve

loglog(ns,1./ns)     # red curve
loglog(ns,1./ns.^2)  # aqua curve;
```

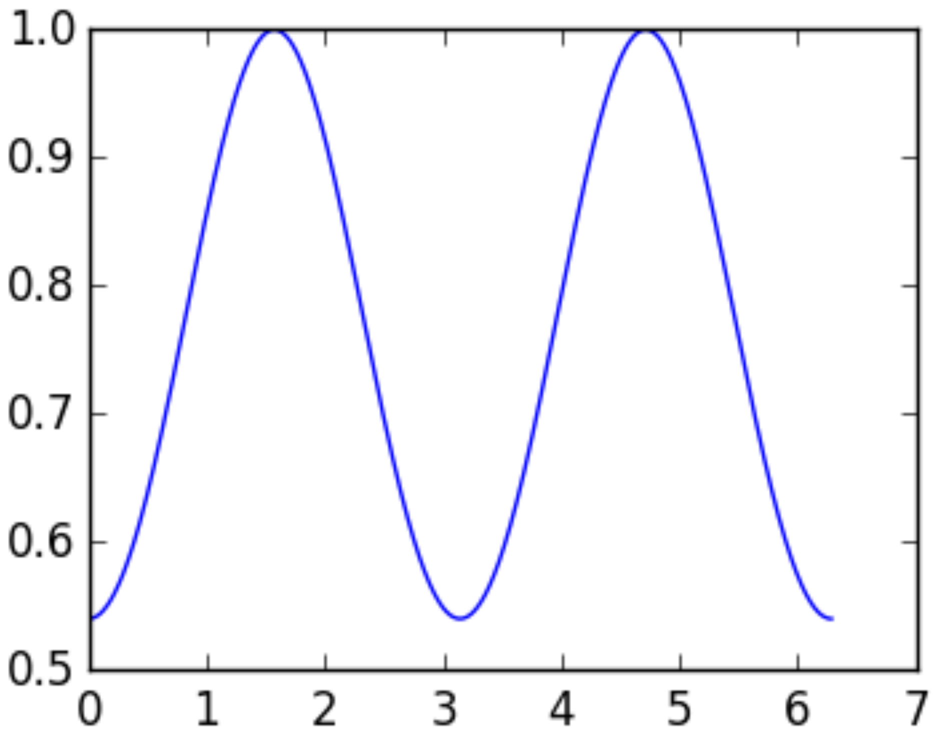


Periodic functions

One last thing, which should come as a suprise. Consider now a periodic function (on $[0, 2\pi)$):

In [29]:

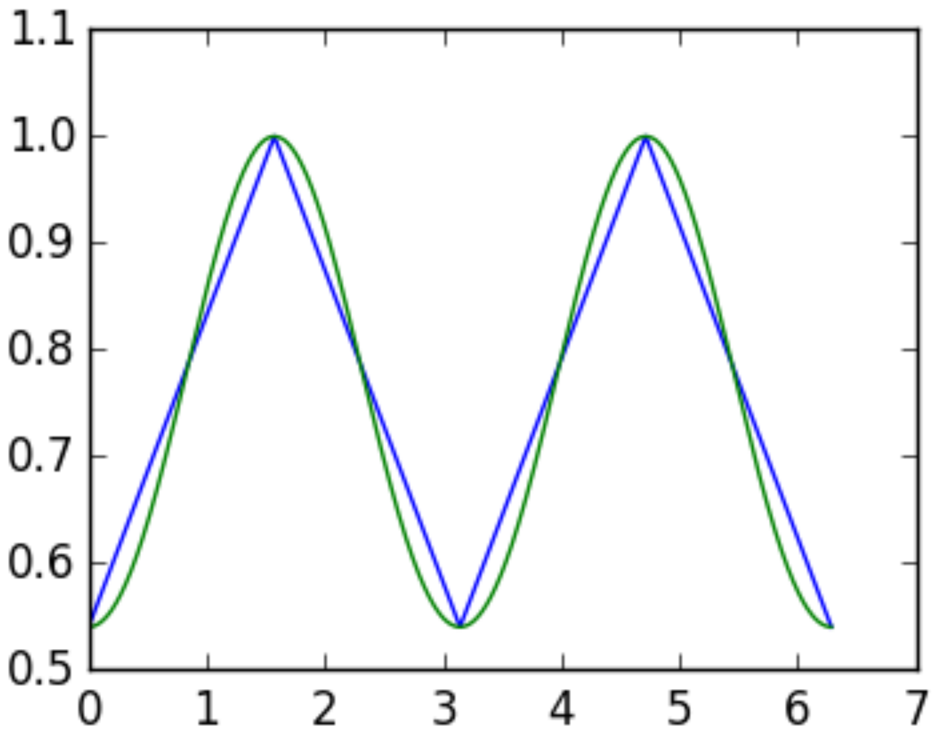
```
f=θ -> cos(cos(θ))

g=linspace(0.,2π,1000)

plot(g,f(g));
```



Approximating the functions by trapeziums is still very inaccurate:

In [24]:

```
n=4
θ=linspace(0.,2π,n+1)

plot(θ,f(θ))
plot(g,f(g));
```



The `trapθ` function gives the trapezium rule on `[0,2π)`, which is just the original trapezium rule scaled by `2π`.

In [33]:

```
function trapθ(f,n)
    h=2π/n
    x=linspace(0.,2π,n+1)

    v=f(x)
    h/2*v[1]+sum(v[2:end-1]*h)+h/2*v[end]
end;
```

Amazingly, the error is tiny!

In [34]:

```
ex=quadgk(f,0.,2π)[1]
trapθ(f,16)-ex
```

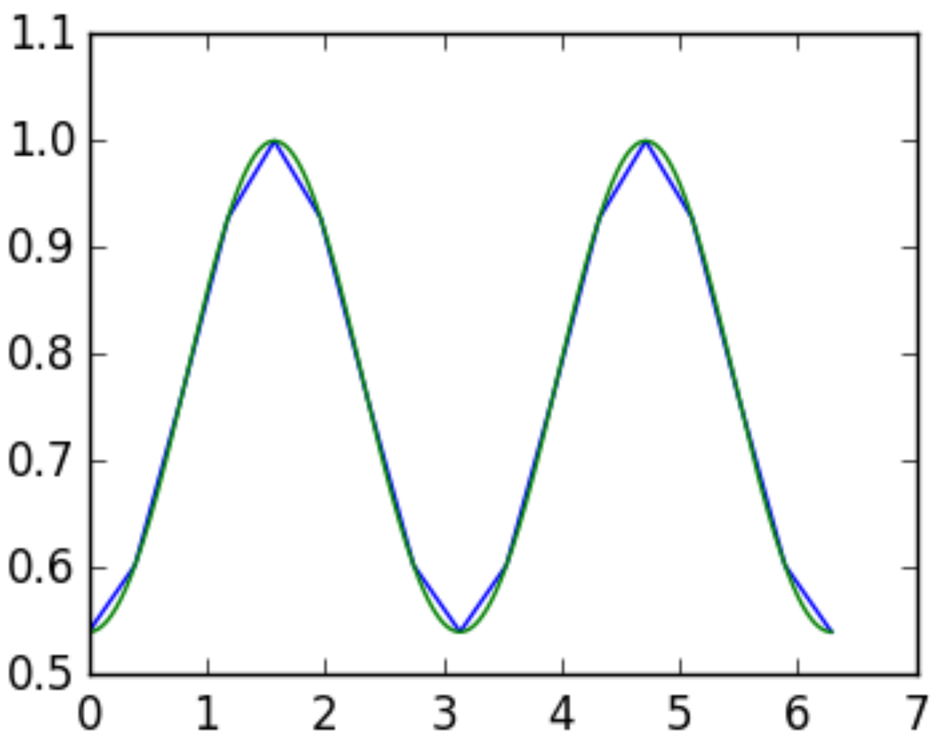
Out[34]:

```
-8.881784197001252e-16
```

This is despite the fact that the trapezoids are still no where near the curve:

In [36]:

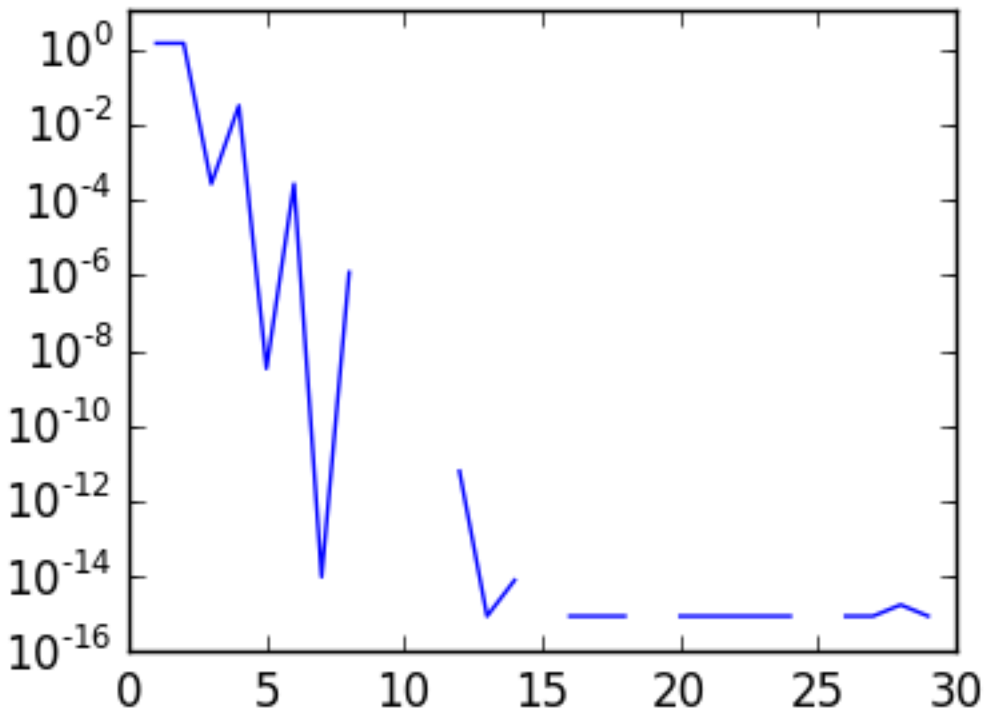
```
n=16
θ=linspace(0.,2π,n+1)

plot(θ,f(θ))
plot(g,f(g));
```



We can plot the error with `semilogy`, to see that the error is actually going down exponentially fast:

```
In [38]:
```

```
ns=1:30

errT=zeros(length(ns))

for k=1:length(ns)
    errT[k]=abs(trapθ(f,k)-ex     )
end

semilogy(ns,errT);
```



10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
0
5
10
15
20
25
30