Programming Simpson's rule

Author: John A. Gear

Simpson's Rule

For the approximate solution of a definite integral: $\int_a^b f(x)\,dx$. Simpson's rule requires an even number of intervals. We will define the interval distance as $h = \frac{b-a}{/}2n$. Then there are $2n$ intervals. Simpson's rule is:

$$S_{2n} = \frac{1}{3}h\left(f(a) + f(b) + 2\sum_{k=1}^{n-1} f\left(a + 2kh\right) + 4\sum_{i=1}^{n} f\left(a + (2i-1)h\right)\right)$$

A bad program

Lets estimate $I = \int_0^1 \frac{1}{1+x^2}dx$ We will first define values for the end points, the number of intervals, calculate the interval distance h and create a procedure for the integrand. The summations will be performed using loops.
We first define a function in terms of x The statement below creates an anonymous function that finds $y = 1/(1+x^2)$. When you call this function, MATLAB assigns the value you pass in to variable x, and then evaluates the equation:

```
format compact
format long
funct = @(x) 1/(1+x^2);
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `funct`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by the argument in parentheses. Because `funct` is a function handle, you can pass it in an argument list to other functions. To execute the `funct` function defined above, type

```
funct(1)
```

```
ans =
   0.500000000000000
```

The program

```
a=0; b=1; n=5; h=(b-a)/(2*n);
funct = @(x) 1/(1+x^2);
tot=funct(a)+funct(b);
for k=1:n-1
    tot=tot+2.0*funct(a+2*k*h);
end;
for i=1:n
    tot=tot+4.0*funct(a+(2*i-1)*h);
end;
tot=h*tot/3.0
```

```
tot =
   0.785398153484804
```

The exact value of the definite integral is $\frac{\pi}{4}$ which to ten digits is 0.7853981635
The above program has all the code in one group of code which is good. I can copy and paste the code, edit it, maybe increase the number of intervals in order to obtain a more accurate estimate or change the end points or even change the integrand. So if I want to change the integrand, the end points, or the number of intervals I have to edit the program. This is very BAD. A smarter option is to create a Matlab function where the integrand, end points and number of intervals are passed to the function in the calling sequence.

A First Procedure

Again we will have $2n$ intervals, so that we always get an even number of them. We will pass the integrand, end points and n in the calling sequence.

```
type SimpRule1.m
```

```
function output=SimpRule1(funct,a,b,n)
h=(b-a)/(2*n);
tot=funct(a)+funct(b);
for k=1:n-1
    tot=tot+2.0*funct(a+2*k*h);
end;
for i=1:n
    tot=tot+4.0*funct(a+(2*i-1)*h);
end;
output=h*tot/3.0;
end
```

Lets test it out. We will estimate the same integral as before. Notice how the function is entered in call to `SimpRule1`

```
ans=SimpRule1(@(x) 1/(1+x^2), 0,1,5)
```

```
ans =
   0.785398153484804
```

Note the same value we obtained before so it appears to work well. Lets estimate $J = \int_0^{\pi} \sin\left(\frac{\theta}{2}\right) d\theta$

```
ans=SimpRule1(@(th) sin(th/2), 0,pi,5)
```

```
ans =
   2.000006784441801
```

The true value is 2. With 10 intervals Simpson's rule produces an estimate which is correct to four decimal places.

A Better Function

We could improve the function by decreasing the number of multiplications. Every time the integrand is evaluated in one of the loops it is multiplied by 2 or 4 and then added to the accumulated total. If we did the summation first then multiplied by 2 or 4 and then added it to the accumulated total we can significantly decrease the number of multiplications being performed.

```
type SimpRule2.m
```

```
function output=SimpRule2(funct,a,b,n)
h=(b-a)/(2*n);
add1=0.0;
for k=1:n-1
    add1=add1+funct(a+2*k*h);
end;
add2=0.0;
for i=1:n
    add2=add2+funct(a+(2*i-1)*h);
end;
output=h*(funct(a)+funct(b)+2.0*add1+4.0*add2)/3.0;
end
```

Now lets estimate the same integrals as before.

```
ans=SimpRule2(@(x) 1/(1+x^2), 0,1,5)
```

```
ans =
   0.785398153484804
```

```
ans=SimpRule2(@(th) sin(th/2), 0,pi,5)
```

```
ans =
   2.000006784441801
```

Which function is faster? We will run both functions 1000 intervals by 100 times and MatLab `tic` and `toc` will be used measure the elapsed time in both cases.

```
tic
   for ic=1:100
       SimpRule2(@(x) 1/(1+x^2), 0,1,500);
   end;
toc
```

```
Elapsed time is 0.048062 seconds.
```

```
tic
   for ic=1:100
       SimpRule1(@(x) 1/(1+x^2), 0,1,500);
   end;
toc
```

```
Elapsed time is 0.047636 seconds.
```

Surprisingly there is negligible difference in elapsed time.

A Functional Procedure

Instead of using loops to evaluate the summations we will now let the integrand operate on a vector of points. The integrand has to be rewritten to allow point by point division and exponentiation.

```
a=0; b=1; n=5; h=(b-a)/(2*n);
funct2 = @(x) 1./(1+x.^2);
```

Notice the change in how `funct2` is defined. In the first summation the integrand operates on the points `a+2*k*h, k = 1 to n-1`. We can create a list of points, evaluate the integrand at theses points and then add them using the `sum` command.

```
add1=sum(funct2(a+2*[1:n-1]*h));
```

In the second summation the integrand operates on the points `a+(2*i-1)*h, i = 1 to n`

```
add2=sum(funct2(a+(2*[1:n]-1)*h));
tot=h*(funct2(a)+funct2(b)+2*add1+4*add2)/3.0
```

```
tot =
   0.785398153484804
```

As a procedure:

```
type SimpRule3.m
```

```
function output=SimpRule3(fun,a,b,n)
h=(b-a)/(2*n);
output=h*(fun(a)+fun(b)+2*sum(fun(a+2*[1:n-1]*h))...
    +4*sum(fun(a+(2*[1:n]-1)*h)))/3.0;
end
```

Lets time it. Same integral same number of intervals.

```
tic
   for ic=1:100
       SimpRule3(@(x) 1./(1+x.^2), 0,1,500);
   end;
toc
```

```
Elapsed time is 0.018664 seconds.
```

We see that the functional procedure is substantially faster.

Order of Simpson's rule

The error bound formula for the composite Simpson's rule tells us that the error is proportional to the interval distance to the power of four (h^4). That is, the composite Simpson's rule is a fourth order method. So when the number of intervals is doubled, the interval distance is halved and the error should decrease to about one sixteenth of its previous amount. If `I(n)` is an estimate with `n` intervals then an estimate of the error in `I(n)` is `I(2n)-I(n)`. When the number of intervals are continuously doubled, the ratio of error estimates should approach $2^4 = 16$. We will estimate $K = \int_0^{\pi/2} \frac{e^{\cos(\theta)}}{1+\sin^2(\theta)}\, d\theta$. with 2, 4, 8, 16, ..., 1024, intervals. Ten estimates.

```
format long
kmax=10;
s1=zeros(kmax,2);
for ic=1:kmax
    n=2^(ic-1); s1(ic,1)=2*n;
    s1(ic,2)=SimpRule3(@(x) exp(cos(x))./(1+(sin(x)).^2),0,pi/2,n);
end;
disp(['Intervals   Estimate']);
for ic=1:kmax
    disp(sprintf('%4d    %16.12f',s1(ic,:)));
end
```

```
Intervals   Estimate
   2      2.258435574007
   4      2.339910536784
   8      2.342721275239
  16      2.342735122531
  32      2.342735847643
  64      2.342735892993
 128      2.342735895828
 256      2.342735896005
 512      2.342735896016
1024      2.342735896017
```

We now calculate estimates of the error. Note that there are now only 9 values in the table. The error in the estimate with 512 intervals is estimated as the difference between the estimate with 1024 intervals and the estimate with 512 intervals.

```
s2=zeros(kmax-1,3);
for ic=1:kmax-1
    s2(ic,1)=s1(ic,1);
    s2(ic,2)=s1(ic,2);
    s2(ic,3)=abs(s1(ic+1,2)-s1(ic,2));
end;
disp(['Intervals   Estimate           Err. Est.']);
for ic=1:kmax-1
    disp(sprintf('%4d    %16.12f     %13.6e',s2(ic,:)));
end;
```

```
Intervals   Estimate           Err. Est.
   2      2.258435574007      8.147496e-02
   4      2.339910536784      2.810738e-03
   8      2.342721275239      1.384729e-05
  16      2.342735122531      7.251119e-07
  32      2.342735847643      4.535029e-08
  64      2.342735892993      2.834835e-09
 128      2.342735895828      1.771836e-10
 256      2.342735896005      1.107292e-11
 512      2.342735896016      6.901146e-13
```

We now calculate the successive ratio of error estimates. The table now only has 8 rows as we can only calculate 8 ratios from 9 error estimates.

```
s3=zeros(kmax-2,4);
for ic=1:kmax-2
    s3(ic,1)=s2(ic,1);
    s3(ic,2)=s2(ic,2);
    s3(ic,3)=s2(ic,3);
    s3(ic,4)=s2(ic,3)/s2(ic+1,3);
end;
disp(['Intervals   Estimate          Err. Est.     Ratio']);
for ic=1:kmax-2
    disp(sprintf('%4d    %16.12f   %13.6e   %g',s3(ic,:)));
end;
```

```
Intervals   Estimate          Err. Est.     Ratio
   2      2.258435574007    8.147496e-02   28.987
   4      2.339910536784    2.810738e-03   202.981
   8      2.342721275239    1.384729e-05   19.0968
  16      2.342735122531    7.251119e-07   15.9891
  32      2.342735847643    4.535029e-08   15.9975
  64      2.342735892993    2.834835e-09   15.9994
 128      2.342735895828    1.771836e-10   16.0015
 256      2.342735896005    1.107292e-11   16.045
```

The last column of this table shows the ratio of error estimates. The elements in the last column clearly approach 16 as the number of intervals are continuously doubled. This numerically confirms that the composite Simpson's rule is fourth order.