Exercise set 9

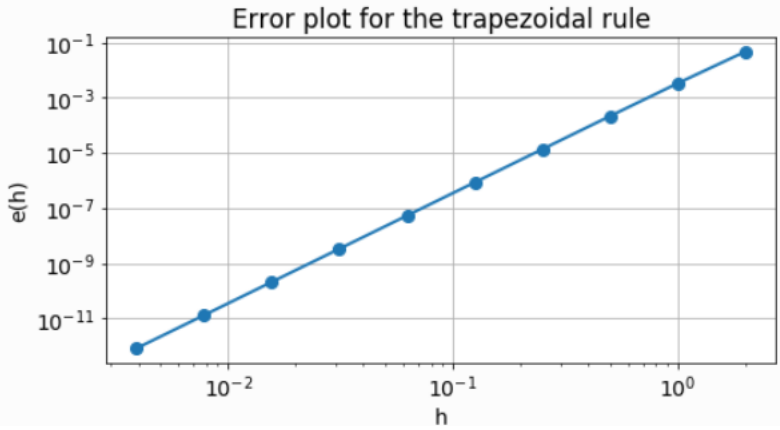
1.

a.

```
In [3]:  %matplotlib inline

         from numpy import *
         from matplotlib.pyplot import *
         from math import factorial
         newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
                      'lines.markersize': 8, 'lines.linewidth': 2,
                      'font.size': 14}
         rcParams.update(newparams)

         def simpson(f, a, b, m=10):
         # Find an approximation to an integral by the composite Simpson's method:
         # Input:
         #   f:    integrand
         #   a, b: integration interval
         #   m:    number of subintervals
         # Output: The approximation to the integral
             n = 2*m
             x_noder = linspace(a, b, n+1)       # equidistributed nodes from a to b
             h = (b-a)/n                         # stepsize
             S1 = f(x_noder[0]) + f(x_noder[n])  # S1 = f(x_0)+f(x_n)
             S2 = sum(f(x_noder[1:n:2]))         # S2 = f(x_1)+f(x_3)+...+f(x_m)
             S3 = sum(f(x_noder[2:n-1:2]))       # S3 = f(x_2)+f(x_4)+...+f(x_{m-1})
             S = h*(S1 + 4*S2 + 2*S3)/3
             return S

         # Code for exercise
         def f(x):                      # Integrand
             return x*e**x

         a, b = -1, 1                   # Integration interval
         exact = 2/e                  # Exact value of the integral (for comparision)
```

```
# Using the error plot from Premliminaries
def compute_plot_error(f, a, b, exact):

    # Find an numerical approximation for different values of h.
    # Store the stepsize h and the error
    n = 1                           # initial stepsize, h=(b-a)
    h = (b-a)/n
    steps = []                      # arrays to store stepsizes and errors
    errors = []
    Nmax = 10
    for k in range(Nmax):
        numres = simpson(f, a, b, n)    # Numerical approximation
        eh = abs(exact - numres)            # Error e(h)
        print('h = {:8.2e},   T(h) = {:10.8f},   e(h) = {:8.2e}'.format(h, numres, eh))
        steps.append(h)                     # Append the step to the array
        errors.append(eh)                   # Append the error to the array
        n = 2*n                             # Reduce the stepsize with a factor 2
        h = (b-a)/n

    # Find the order and the error constant
    print('\nThe order p and the error constant C')
    for k in range(1, Nmax-1):
        p = log(errors[k+1]/errors[k])/log(steps[k+1]/steps[k])
        C = errors[k+1]/steps[k+1]**p
        print('h = {:8.2e},  p = {:4.2f},  C = {:6.4f}'.format(steps[k], p, C))

    # Make an error plot
    clf()
    loglog(steps, errors, 'o-')
    xlabel('h')
    ylabel('e(h)')
    title('Error plot for the trapezoidal rule')
    grid(True)


compute_plot_error(f, a, b, exact)
```

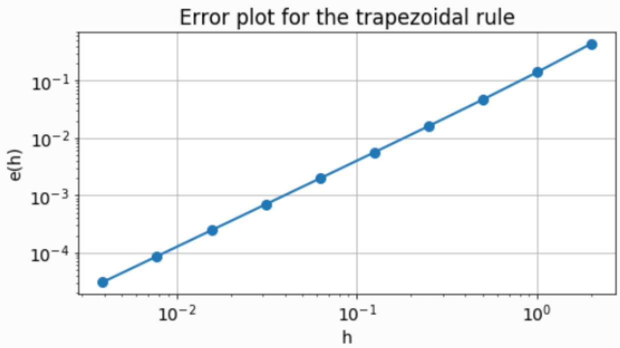
```
h = 2.00e+00,    T(h) = 0.78346746,    e(h) = 4.77e-02
h = 1.00e+00,    T(h) = 0.73913060,    e(h) = 3.37e-03
h = 5.00e-01,    T(h) = 0.73597650,    e(h) = 2.18e-04
h = 2.50e-01,    T(h) = 0.73577259,    e(h) = 1.37e-05
h = 1.25e-01,    T(h) = 0.73575974,    e(h) = 8.59e-07
h = 6.25e-02,    T(h) = 0.73575894,    e(h) = 5.37e-08
h = 3.12e-02,    T(h) = 0.73575889,    e(h) = 3.36e-09
h = 1.56e-02,    T(h) = 0.73575888,    e(h) = 2.10e-10
h = 7.81e-03,    T(h) = 0.73575888,    e(h) = 1.31e-11
h = 3.91e-03,    T(h) = 0.73575888,    e(h) = 8.19e-13

The order p and the error constant C
h = 1.00e+00,   p = 3.95,   C = 0.0034
h = 5.00e-01,   p = 3.99,   C = 0.0035
h = 2.50e-01,   p = 4.00,   C = 0.0035
h = 1.25e-01,   p = 4.00,   C = 0.0035
h = 6.25e-02,   p = 4.00,   C = 0.0035
h = 3.12e-02,   p = 4.00,   C = 0.0035
h = 1.56e-02,   p = 4.00,   C = 0.0035
h = 7.81e-03,   p = 4.00,   C = 0.0035
```



b.

```python
# Code for exercise
def f(x):                          # Integrand
    return sqrt(1-x**2)*e**x

a, b = -1, 1                       # Integration interval
exact = 1.7754996892121809469                      # Exact value of the integral (for comparision)
```

```
h = 2.00e+00,   T(h) = 1.33333333,   e(h) = 4.42e-01
h = 1.00e+00,   T(h) = 1.63540364,   e(h) = 1.40e-01
h = 5.00e-01,   T(h) = 1.72886020,   e(h) = 4.66e-02
h = 2.50e-01,   T(h) = 1.75945841,   e(h) = 1.60e-02
h = 1.25e-01,   T(h) = 1.76989907,   e(h) = 5.60e-03
h = 6.25e-02,   T(h) = 1.77353123,   e(h) = 1.97e-03
h = 3.12e-02,   T(h) = 1.77480571,   e(h) = 6.94e-04
h = 1.56e-02,   T(h) = 1.77525467,   e(h) = 2.45e-04
h = 7.81e-03,   T(h) = 1.77541312,   e(h) = 8.66e-05
h = 3.91e-03,   T(h) = 1.77546909,   e(h) = 3.06e-05

The order p and the error constant C
h = 1.00e+00,  p = 1.59,  C = 0.1401
h = 5.00e-01,  p = 1.54,  C = 0.1356
h = 2.50e-01,  p = 1.52,  C = 0.1316
h = 1.25e-01,  p = 1.51,  C = 0.1290
h = 6.25e-02,  p = 1.50,  C = 0.1274
h = 3.12e-02,  p = 1.50,  C = 0.1265
h = 1.56e-02,  p = 1.50,  C = 0.1260
h = 7.81e-03,  p = 1.50,  C = 0.1257
```



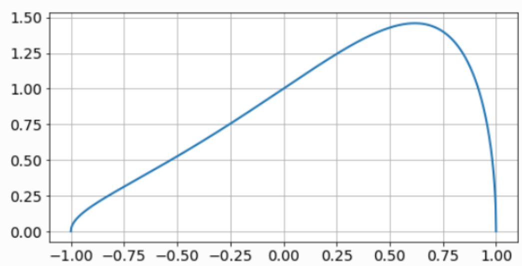
The convergence rate 1.5 is much lower than 4. To see the problem, we'll plot the integrand as the task recommends.

In [5]:

```
%matplotlib inline

from numpy import *
from matplotlib.pyplot import *
from math import factorial
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
rcParams.update(newparams)

x = linspace(-1, 1, 1001)
plot(x, f(x))
```

executed in 946ms, finished 08:26:27 2019-11-05

Out[5]: [<matplotlib.lines.Line2D at 0x7f7320a6d898>]



We can see that the derivative of f is very large in the area around x=1. Since the error depends on the fourth derivative of f, the error when computing the integral near x=1 will be quite large. Thus, the convergence rate would be higher if we chose the integration area to be [-1, 0] instead.

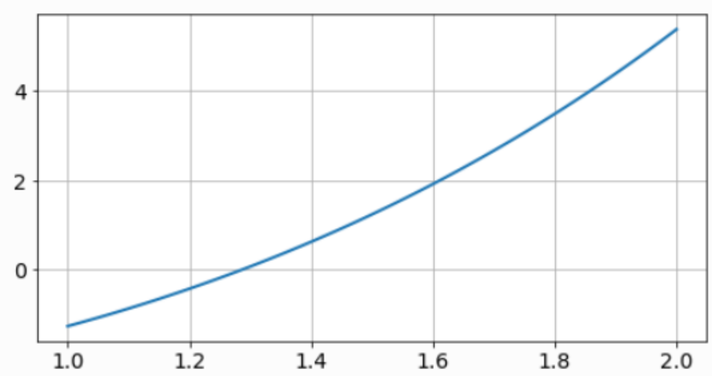
3.

a. Plot of f

```
def f(x):
    return e**x + x**2 - x - 4
x = linspace(1, 2, 101)
plot(x, f(x))
```

executed in 2.06s, finished 08:47:59 2019-11-05

```
[<matplotlib.lines.Line2D at 0x7f8eb31a3550>]
```



Newton's method applied on f:

```
def newton(f, df, x0, tol=1.e-8, max_iter=30):
    # Solve f(x)=0 by Newtons method
    # The output of each iteration is printed
    # Input:
    #   f, df:   The function f and its derivate f'.
    #   x0:  Initial values
    #   tol: The tolerance
    # Output:
    #   The root and the number of iterations
    x = x0
    print('k ={:3d}, x = {:18.15f}, f(x) = {:10.3e}'.format(0, x, f(x)))
    for k in range(max_iter):
        fx = f(x)
        if abs(fx) < tol:           # Accept the solution
            break
        x = x - fx/df(x)            # Newton-iteration
        print('k ={:3d}, x = {:18.15f}, f(x) = {:10.3e}'.format(k+1, x, f(x)))
    return x, k+1
```

```
def f(x):                     # The function f
    return e**x+x**2-x-4

def df(x):                    # The derivative f'
    return e**x+2*x-1

x0 = 1.3                        # Starting value
x, nit = newton(f, df, x0)    # Apply Newton
print('\n\nResult:\nx={}, number of iterations={}'.format(x, nit))
```

executed in 11ms, finished 09:06:05 2019-11-05

```
k =  0, x =  1.300000000000000, f(x) =  5.930e-02
k =  1, x =  1.288746758560088, f(x) =  3.581e-04
k =  2, x =  1.288677969382023, f(x) =  1.332e-08
k =  3, x =  1.288677966823869, f(x) =  0.000e+00


Result:
x=1.2886779668238686, number of iterations=4
```

b.

```python
def fixpoint(g, x0, tol=1.e-8, max_iter=30):
    # Solve x=g(x) by fixed point iterations
    # The output of each iteration is printed
    # Input:
    #   g:   The function g(x)
    #   x0:  Initial values
    #   tol: The tolerance
    # Output:
    #   The root and the number of iterations
    x = x0
    print('k ={:3d}, \tx = {:14.10f}'.format(0, x))
    for k in range(max_iter):
        x_old = x                     # Store old values for error estimation
        x = g(x)                      # The iteration
        err = abs(x-x_old)            # Error estimate
        print('k ={:3d}, \tx = {:14.10f}'.format(k+1, x))
        if err < tol:           # The solution is accepted
            break
    return x, k+1


# Define the functions
def g1(x):
    return log(4 + x - x**2)

def g2(x):
    return sqrt(-e**x + x + 4)

def g3(x):
    return e**x + x**2 - 4

x0 = 1.5
for g in g1, g2, g3:
    print("g = %s:" % g.__name__)
    fixpoint(g, x0, max_iter=4)
```

```
g = g1:
k =  0,         x =   1.5000000000
k =  1,         x =   1.1786549963
k =  2,         x =   1.3322149248
k =  3,         x =   1.2690350905
k =  4,         x =   1.2970764687
g = g2:
k =  0,         x =   1.5000000000
k =  1,         x =   1.0091139329
k =  2,         x =   1.5053054929
k =  3,         x =   0.9998878264
k =  4,         x =   1.5105995169
g = g3:
k =  0,         x =   1.5000000000
k =  1,         x =   2.7316890703
k =  2,         x =  18.8209324059
k =  3,         x = 149220368.2729533017

---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-13-b10819713463> in <module>
     34 for g in g1, g2, g3:
     35     print("g = %s:" % g.__name__)
---> 36     fixpoint(g, x0, max_iter=4)

<ipython-input-13-b10819713463> in fixpoint(g, x0, tol, max_iter)
     12     for k in range(max_iter):
     13         x_old = x                     # Store old values for error estimation
---> 14         x = g(x)                      # The iteration
     15         err = abs(x-x_old)            # Error estimate
     16         print('k ={:3d}, \tx = {:14.10f}'.format(k+1, x))

<ipython-input-13-b10819713463> in g3(x)
     29
     30 def g3(x):
---> 31     return e**x + x**2 - 4
     32
     33 x0 = 1.5

OverflowError: (34, 'Numerical result out of range')
```

As we can see from the results, the first choice of g converges (at least it seems like it does). The second choice oscillates between values of both sides of zero but does not get closer to the correct answer (the interval expands rather than shrink). The third choice is simply just a bad choice, as it diverges quite rapidly and causes an overflow in only four iterations.

4.

c.

```python
def f(x):
    return cos(x)

fixpoint(f, 0.5, max_iter=20)
```

executed in 18ms, finished 10:29:19 2019-11-05

```
k =  0,         x =   0.5000000000
k =  1,         x =   0.8775825619
k =  2,         x =   0.6390124942
k =  3,         x =   0.8026851007
k =  4,         x =   0.6947780268
k =  5,         x =   0.7681958313
k =  6,         x =   0.7191654459
k =  7,         x =   0.7523557594
k =  8,         x =   0.7300810631
k =  9,         x =   0.7451203414
k = 10,         x =   0.7350063090
k = 11,         x =   0.7418265226
k = 12,         x =   0.7372357254
k = 13,         x =   0.7403296519
k = 14,         x =   0.7382462383
k = 15,         x =   0.7396499628
k = 16,         x =   0.7387045394
k = 17,         x =   0.7393414523
k = 18,         x =   0.7389124493
k = 19,         x =   0.7392014441
k = 20,         x =   0.7390067798

(0.7390067797808l297, 20)
```

By checking the result, we can see that the approximation seems reasonable:

```python
print()
print(cos(0.739007))
print(arccos(0.739007))
```

```
0.739137762433
0.739201117255
```

5.

b.

```
set_printoptions(precision=15)          # Output with high accuracy
```

```
def newton_system(f, jac, x0, tol = 1.e-10, max_iter=20):
    x = x0
    print('k ={:3d}, x = '.format(0), x)
    for k in range(max_iter):
        fx = f(x)
        if norm(fx, inf) < tol:          # The solution is accepted.
            break
        Jx = jac(x)
        delta = solve(Jx, -fx)
        x = x + delta
        print('k ={:3d}, x = '.format(k+1), x)
    return x, k

# Example 6

# The vector valued function. Notice the indexing.
def f(x):
    y = array([x[0]**2+x[1]**2-4,
               x[0]*x[1]-1])
    return y

# The Jacobian
def jac(x):
    J = array([[2*x[0], 2*x[1]],
         [x[1],    x[0]]])
    return J

x0 = array([2, 0])          # Starting values
max_iter = 10
newton_system(f, jac, x0, tol = 1.e-10, max_iter = max_iter)  # Apply Newton's method
```

executed in 78ms, finished 12:06:31 2019-11-05

```
k =  0, x =  [2 0]
k =  1, x =  [ 2.   0.5]
k =  2, x =  [ 1.93333333  0.51666667]
k =  3, x =  [ 1.93185274  0.51763705]
k =  4, x =  [ 1.93185165  0.51763809]

(array([ 1.93185165,  0.51763809]), 4)
```

c.

```
# The vector valued function. Notice the indexing.
def f(x):
    y = array([x[0]**2+x[1]**2-2,
               x[0]*x[1]-1])
    return y

# The Jacobian
def jac(x):
    J = array([[2*x[0], 2*x[1]],
         [x[1],    x[0]]])
    return J

x0 = array([2, 0])          # Starting values
```

```
k =  0, x =  [2 0]
k =  1, x =  [ 1.5  0.5]
k =  2, x =  [ 1.25  0.75]
k =  3, x =  [ 1.125  0.875]
k =  4, x =  [ 1.0625  0.9375]
k =  5, x =  [ 1.03125  0.96875]
k =  6, x =  [ 1.015625  0.984375]
k =  7, x =  [ 1.0078125  0.9921875]
k =  8, x =  [ 1.00390625  0.99609375]
k =  9, x =  [ 1.00195312  0.99804688]
k = 10, x =  [ 1.00097656  0.99902344]

Out[29]: (array([ 1.00097656,  0.99902344]), 9)
```