Numerical Solutions for DEs HW3

YANG, Ze (5131209043)

April 1, 2017

A Note to TA:
Hi, this is the senior student from Antai College who did not register for this course. I would like to do all the assignments for practice, but feel free to just skip my homework if you don't have time. Thank you again for allowing me to access the assignments and other class material! :)
- Ze

Problem 1. (Iserles 4.4) Determine all values of θ such that the theta method (1.13) is absolutely stable.

Proof. The theta method reads

$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + h[\theta \boldsymbol{f}(t_n, \boldsymbol{y}_n) + (1-\theta)\boldsymbol{f}(t_{n+1}, \boldsymbol{y}_{n+1})], \qquad n = 0, 1, ...$$

To find the a-stability region ($\mathcal{D}$) we apply the method to scalar ode,

$$\begin{aligned} y_{n+1} &= y_n + h[\theta\lambda y_n + (1-\theta)\lambda y_{n+1}] \\ y_{n+1} &= \frac{1+h\lambda\theta}{1-h\lambda(1-\theta)} y_n = \frac{1+z\theta}{1-z(1-\theta)} y_n \end{aligned} \tag{1}$$

Therefore, we want to find θ, such that $\mathbb{C}^- = \{z \in \mathbb{C} : \mathrm{Re}(z) < 0\} \subseteq \{z \in \mathbb{C} : \left|\frac{1+z\theta}{1-z(1-\theta)}\right| < 1\} = \mathcal{D}$ (By the definition of a-stability). I.e. we want $|1 + z\theta| < |1 - z(1-\theta)|$ $(*)$ holds for all z that $\mathrm{Re}(z) < 0$.

Write $z = x + yi$:

- $\theta = 1/2$, the method is a-stable, since it's the trapezoid method.
- $\frac{1}{2} < \theta \leq 1$, we write $\theta = \frac{1}{2} + \delta$. Let $z = -1/\delta + 0i$. Then $LHS = 1/2\delta$, $RHS = |1 + 1/\delta(-1/2 - \delta)| = 1/2\delta$. Then the equation $(*)$ does not hold, so in this case the method is not a-stable.
- $0 \leq \theta < \frac{1}{2}$, we want to show

  $$|1 + \theta x + \theta y i| < |1 - (1-\theta)x - (1-\theta)yi|$$

  for all $(x, y) \in \mathbb{C}^-$. Denote the complex number inside absolute value as r (right hand side) and l (left hand side). $\mathrm{Im}(l) = \theta y$, $\mathrm{Im}(l) = (1-\theta)y$, since $0 \leq \theta < \frac{1}{2} \Rightarrow 1 - \theta > \theta > 0$. So $\mathrm{Im}(l) > \mathrm{Im}(r) > 0$, Hence $|\mathrm{Im}(l)| > |\mathrm{Im}(r)|$.
  For the real parts, $\mathrm{Re}(l) = 1 + x\theta$, $\mathrm{Re}(r) = 1 - (1-\theta)x$.

  1. $1 + x\theta \leq 0$: since we must have $x < 0$. $(2\theta - 1) < 0 \Rightarrow (2\theta - 1)x > 0$, $2 + (2\theta - 1)x > 0 \Rightarrow 1 - (-\theta + 1)x > -1 - \theta x \geq 0$, i.e. $\mathrm{Re}(r) > -\mathrm{Re}(l) \geq 0$.
  2. $1 + x\theta > 0$: $\theta x < -(1-\theta)x \Rightarrow 0 < 1 + \theta x < 1 - (1-\theta)x \Rightarrow \mathrm{Re}(r) > \mathrm{Re}(l) \geq 0$.

  In either case we have $|\mathrm{Re}(l)| > |\mathrm{Re}(r)|$. So finally, we have $|r| < |l|$ for all $z \in \mathbb{C}^-$. And we conclude that for $\theta \in [0, 1/2]$, the method is a-stable.

□

Problem 2. (Iserles 4.9) The two-step method

$$\boldsymbol{y}_{n+2} - \boldsymbol{y}_n = 2h\boldsymbol{f}(t_{n+1}, \boldsymbol{y}_{n+1}), \quad n = 0, 1, ...$$

is called the explicit midpoint rule.

a. Denote by $w_1(z)$ and $w_2(z)$ the zeroes of the underlying function $\eta(z, \cdot)$, show that $w_1(z)w_2(z) \equiv -1$ for all $z \in \mathbb{C}$.
b. Show that $\mathcal{D} = \emptyset$.
c. We say that $\tilde{\mathcal{D}}$ is a *weak linear stability domain* of a numerical method if, when applied to the scalar linear equation, it produces a uniformly bounded solution sequence. (It is easy to see that $\tilde{\mathcal{D}} = \text{cl}\mathcal{D}$ for most methods of interest.) Determine explicitly $\tilde{\mathcal{D}}$ for the method.

Proof. (a.) We have $\rho(w) = w^2 - 1$, $\sigma(w) = 2w$. So let $z = \lambda h$, $\eta(w, z) = \rho(w) - z\sigma(w) = w^2 - 2zw - 1$. Suppose $w_1(z)$ and $w_2(z)$ solves $\eta(\cdot, z) = 0$, then by Vieta's formula, we have $w_1(z)w_2(z) = \frac{-1}{1} = -1$. □

(b.) We have $w_1(z) = \frac{2z+\sqrt{4z^2+4}}{2} = z + \sqrt{z^2+1}$. $w_2(z) = z - \sqrt{z^2+1}$. So $\mathcal{D} = \{z \in \mathbb{C} : |w_1(z)| < 1, |w_2(z)| < 1\}$. But

$$|w_1(z)| \cdot |w_2(z)| = |w_1(z)w_2(z)| = |-1| = 1$$

implies that $|w_1(z)| = 1/|w_2(z)|$. Hence either (1) $|w_1| = |w_2| = 1$, or (2) one of the two roots has modulus strictly smaller than 1, another one strictly greater than 1. And for all $z \in \mathbb{C}$, one of the two cases must be true. $\Rightarrow \forall z \in \mathbb{C}$, $z \notin \mathcal{D}$. $\mathcal{D} = \emptyset$.

(c) Apply to scalar linear equation $y' = \lambda y$,

$$y_{n+2} - y_n = 2h\lambda y_{n+1}$$

□

Problem 3. (Iserles 5.8) Show that the *Hénon-Heiles system* (using the definition from wikipedia)

$$\begin{aligned} q_1' &= p_1 \\ p_1' &= -q_1 - 2\lambda q_1 q_2 \\ q_2' &= p_2 \\ p_2' &= -q_2 - \lambda(q_1^2 - q_2^2) \end{aligned} \tag{2}$$

is Hamiltonian and identify explicitly the hamiltonian energy.

Proof. Let $\boldsymbol{q} = (q_1, q_2)$, $\boldsymbol{p} = (p_1, p_2)$. Assume the system is Hamiltonian, then

$$\begin{aligned} \nabla_{\boldsymbol{p}} H &= \dot{\boldsymbol{q}} = \begin{pmatrix} p_1 & p_2 \end{pmatrix} \\ \nabla_{\boldsymbol{q}} H &= -\dot{\boldsymbol{p}} = \begin{pmatrix} q_1 + 2\lambda q_1 q_2 & q_2 + \lambda(q_1^2 - q_2^2) \end{pmatrix} \end{aligned} \tag{3}$$

Hence

$$H(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{2}(p_1^2 + p_2^2) + f(\boldsymbol{q})$$

$$H(\boldsymbol{p}, \boldsymbol{q}) = \int (q_1 + 2\lambda q_1 q_2) dq_1 + g(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{2}q_1^2 + \lambda q_2 q_1^2 + g(\boldsymbol{p}, \boldsymbol{q})$$

$$H(\boldsymbol{p}, \boldsymbol{q}) = \int [q_2 + \lambda(q_1^2 - q_2^2)] dq_2 + h(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{2}q_2^2 + \lambda q_1^2 q_2 - \frac{1}{3}\lambda q_2^3 + h(\boldsymbol{p}, \boldsymbol{q})$$

Combine the first integrals above we have:

$$H(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{2}(p_1^2 + p_2^2) + \frac{1}{2}(q_1^2 + q_2^2) + \lambda\left(q_1^2 q_2 - \frac{1}{3}q_2^3\right)$$

Check the partial derivatives of H, we find that it gives the system indeed. Hence the system is Hamiltonian, and the hamiltonian energy is given by $H(\boldsymbol{p}, \boldsymbol{q})$.

□

Problem 4. The symplectic Euler method for the Hamiltonian system reads

$$\begin{cases} \boldsymbol{p}_{n+1} = \boldsymbol{p}_n - h\frac{\partial H(\boldsymbol{p}_{n+1}, \boldsymbol{q}_n)}{\partial \boldsymbol{q}} \\ \boldsymbol{q}_{n+1} = \boldsymbol{q}_n + h\frac{\partial H(\boldsymbol{p}_{n+1}, \boldsymbol{q}_n)}{\partial \boldsymbol{p}} \end{cases} \tag{4}$$

a. Show that this is a first order method.

b. Prove from basic principles that, as implied by its name, the method is indeed symplectic. (Hint: let $G = \nabla^2 H$, and write $G = [G_{11}, G_{12}; G_{21}, G_{22}]$).

c. Assuming that the Hamiltonian is separable, $H(\boldsymbol{p}, \boldsymbol{q}) = T(\boldsymbol{p}) + V(\boldsymbol{q})$, where T and V correspond to kinetic and potential energy respectively. Show that the method can be implemented explicitly.

d. Use symplectic Euler and explicit Euler to solve the problem of non-linear pendulum. The Hamiltonian is $H(p, q) = \frac{1}{2}p^2 - \cos(q)$, and the Hamiltonian equations are

$$\dot{p} = -\sin q, \qquad \dot{q} = p$$

with initial condition $p(0) = 0$, $q(0) = 1$. Plot the error of the numerical methods in the Hamiltonian H.

Proof. (a.) We examine the truncation error $\boldsymbol{T}_n^{[1,2]}/h$ for $\boldsymbol{q}$ and $\boldsymbol{p}$ respectively. Denote $\boldsymbol{f} = \dot{\boldsymbol{p}} = -\frac{\partial H}{\partial \boldsymbol{q}}$, $\boldsymbol{g} = \dot{\boldsymbol{q}} = \frac{\partial H}{\partial \boldsymbol{p}}$.

$$\begin{aligned} \boldsymbol{T}_n^{[1]}/h &= \frac{\boldsymbol{p}(t_{n+1}) - \boldsymbol{p}(t_n)}{h} - \boldsymbol{f}(\boldsymbol{p}(t_{n+1}), \boldsymbol{q}(t_n)) \\ &= \tfrac{1}{h}[\dot{\boldsymbol{p}}(t_n)h + \tfrac{1}{2}\ddot{\boldsymbol{p}}(t_n)h^2 + O(h^3)] - (\boldsymbol{f}(t_n) + \boldsymbol{f}'(t_n)h + O(h^2)) \\ &= -\tfrac{1}{2}\boldsymbol{f}'(t_n)h + O(h^2) \end{aligned} \tag{5}$$

The calculation is same for $\boldsymbol{T}_n^{[2]}/h$, we obtain $\boldsymbol{T}_n^{[2]}/h = -\frac{1}{2}\boldsymbol{g}'(t_n)h + O(h^2)$. Hence we conclude that the method is of order 1. □

(b.) Let

$$\boldsymbol{G} = \nabla^2 H = \begin{pmatrix} \frac{\partial^2 H}{\partial \boldsymbol{p}^2} & \frac{\partial^2 H}{\partial \boldsymbol{p}\partial \boldsymbol{q}} \\ \frac{\partial^2 H}{\partial \boldsymbol{q}\partial \boldsymbol{p}} & \frac{\partial^2 H}{\partial \boldsymbol{q}^2} \end{pmatrix} =: \begin{pmatrix} \boldsymbol{G}_{pp} & \boldsymbol{G}_{qp}^\top \\ \boldsymbol{G}_{qp} & \boldsymbol{G}_{qq} \end{pmatrix}$$

Then calculate the partial derivatives from the method's formula:

$$\begin{cases} \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} &= \boldsymbol{I} - h\boldsymbol{G}_{qp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} \\ \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} &= -h\boldsymbol{G}_{qq} - h\boldsymbol{G}_{qp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} \\ \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{p}_n} &= h\boldsymbol{G}_{pp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} \\ \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{q}_n} &= \boldsymbol{I} + h\boldsymbol{G}_{pq} + h\boldsymbol{G}_{pp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} \end{cases} \tag{6}$$

I.e.

$$\begin{cases} (\boldsymbol{I} + h\boldsymbol{G}_{qp})\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} = \boldsymbol{I} \\ (\boldsymbol{I} + h\boldsymbol{G}_{qp})\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} = -h\boldsymbol{G}_{qq} \\ -h\boldsymbol{G}_{pp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} + \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{p}_n} = \boldsymbol{O} \\ -h\boldsymbol{G}_{pp}\frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} + \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{q}_n} = \boldsymbol{I} + h\boldsymbol{G}_{pq} \end{cases} \tag{7}$$

rewrite in matrix form:

$$\begin{pmatrix} \boldsymbol{I} + h\boldsymbol{G}_{qp} & \boldsymbol{O} \\ -h\boldsymbol{G}_{pp} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} & \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} \\ \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{p}_n} & \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{q}_n} \end{pmatrix} = \begin{pmatrix} \boldsymbol{I} & -h\boldsymbol{G}_{qq} \\ \boldsymbol{O} & \boldsymbol{I} + h\boldsymbol{G}_{pq} \end{pmatrix} \tag{8}$$

And solve this linear system:

$$\boldsymbol{\Phi} = \begin{pmatrix} \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{p}_n} & \frac{\partial \boldsymbol{p}_{n+1}}{\partial \boldsymbol{q}_n} \\ \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{p}_n} & \frac{\partial \boldsymbol{q}_{n+1}}{\partial \boldsymbol{q}_n} \end{pmatrix} = \begin{pmatrix} (\boldsymbol{I} + h\boldsymbol{G}_{qp})^{-1} & -h\boldsymbol{G}_{qq}(\boldsymbol{I} + h\boldsymbol{G}_{qp})^{-1} \\ h\boldsymbol{G}_{pp}(\boldsymbol{I} + h\boldsymbol{G}_{qp})^{-1} & \boldsymbol{I} + h\boldsymbol{G}_{pq} - h^2\boldsymbol{G}_{pp}\boldsymbol{G}_{qq}(\boldsymbol{I} + h\boldsymbol{G}_{qp})^{-1} \end{pmatrix} \tag{9}$$

So:

$$\begin{aligned}\boldsymbol{\Phi}^\top \boldsymbol{J} &= \begin{pmatrix} (\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} & h\boldsymbol{G}_{pp}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} \\ -h\boldsymbol{G}_{qq}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} & \boldsymbol{I}+h\boldsymbol{G}_{qp}-h^2\boldsymbol{G}_{pp}\boldsymbol{G}_{qq}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} \end{pmatrix} \begin{pmatrix} \boldsymbol{O} & \boldsymbol{I} \\ -\boldsymbol{I} & \boldsymbol{O} \end{pmatrix} \\ &= \begin{pmatrix} -h\boldsymbol{G}_{pp}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} & (\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} \\ -\boldsymbol{I}-h\boldsymbol{G}_{qp}+h^2\boldsymbol{G}_{pp}\boldsymbol{G}_{qq}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} & -h\boldsymbol{G}_{qq}(\boldsymbol{I}+h\boldsymbol{G}_{pq})^{-1} \end{pmatrix}\end{aligned} \tag{10}$$

And then it is clear that $(\boldsymbol{\Phi}^\top \boldsymbol{J})\boldsymbol{\Phi} = \boldsymbol{J}$. So the method is indeed symplectic. □

(c) If the method is separable, $H(\boldsymbol{p},\boldsymbol{q}) = T(\boldsymbol{p}) + V(\boldsymbol{q})$, then it becomes:

$$\begin{cases} \boldsymbol{p}_{n+1} = \boldsymbol{p}_n - h\frac{\partial V(\boldsymbol{q}_n)}{\partial \boldsymbol{q}} \\ \boldsymbol{q}_{n+1} = \boldsymbol{q}_n + h\frac{\partial T(\boldsymbol{p}_{n+1})}{\partial \boldsymbol{p}} \end{cases} \tag{11}$$

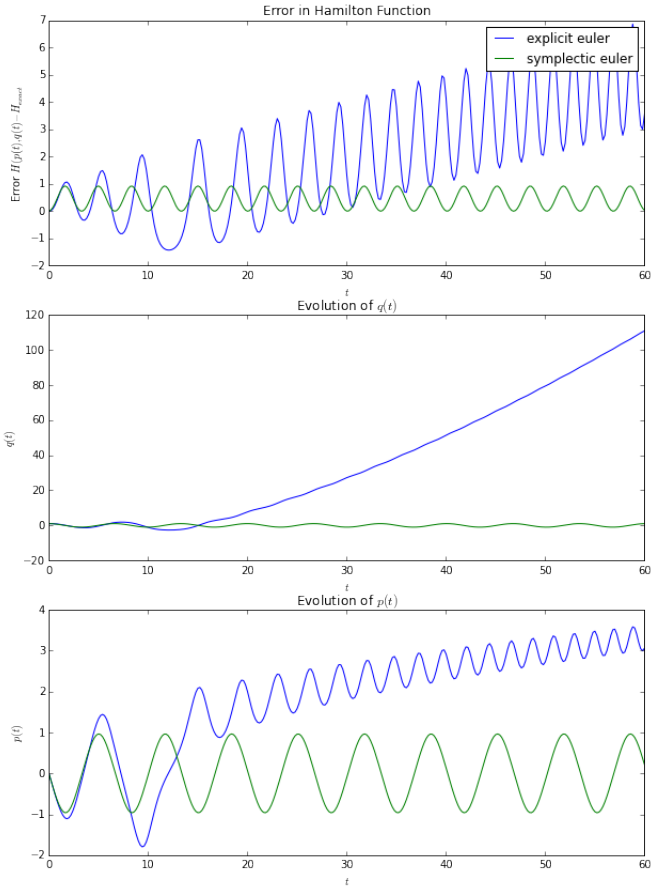
is an explicit method. Denote $\nabla_{\boldsymbol{q}} V = \boldsymbol{f}$ and $\nabla_{\boldsymbol{p}} T = \boldsymbol{g}$, and suppose we know them explicitly. The method is implemented as follows:

Given $\boldsymbol{q}_0, \boldsymbol{p}_0$.
`For` $k = 1$ to n:

$\boldsymbol{p}_k \leftarrow \boldsymbol{p}_{k-1} - h\boldsymbol{f}(\boldsymbol{q}_k)$
$\boldsymbol{q}_k \leftarrow \boldsymbol{q}_{k-1} + h\boldsymbol{g}(\boldsymbol{p}_k) = \boldsymbol{q}_{k-1} + h\boldsymbol{g}(\boldsymbol{p}_{k-1} - h\boldsymbol{f}(\boldsymbol{q}_k))$

which is explicit, and resembles the Runge-Kutta method. □

(d) The plots below gives the numerical solution of this problem choosing $n = 300$, $h = 0.2$. I.e. solve the ode for $t = 0.2, 0.4, ..., 59.8, 60$ with both explicit euler and symplectic euler method.

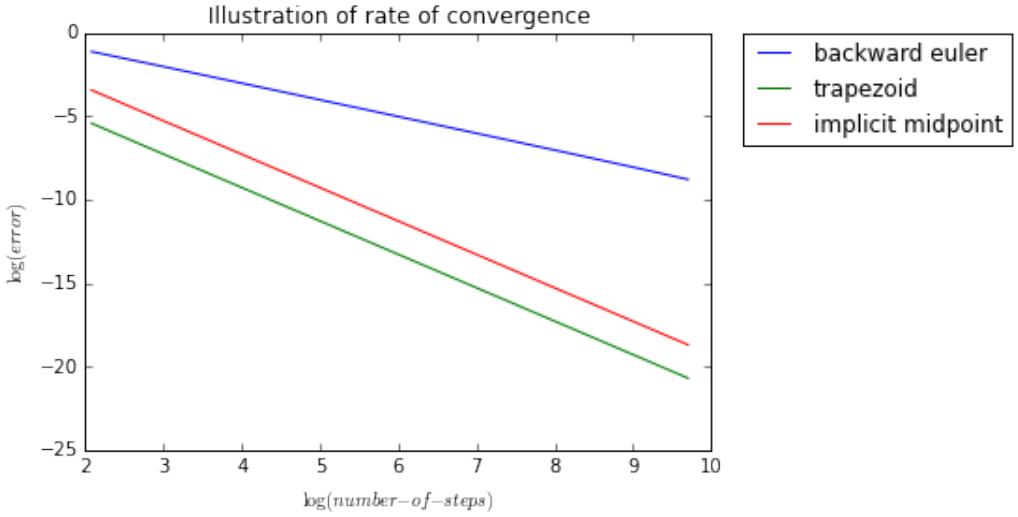


It is clear that for explicit euler method, H gradually grows with t as well as $q(t)$, and finally the object

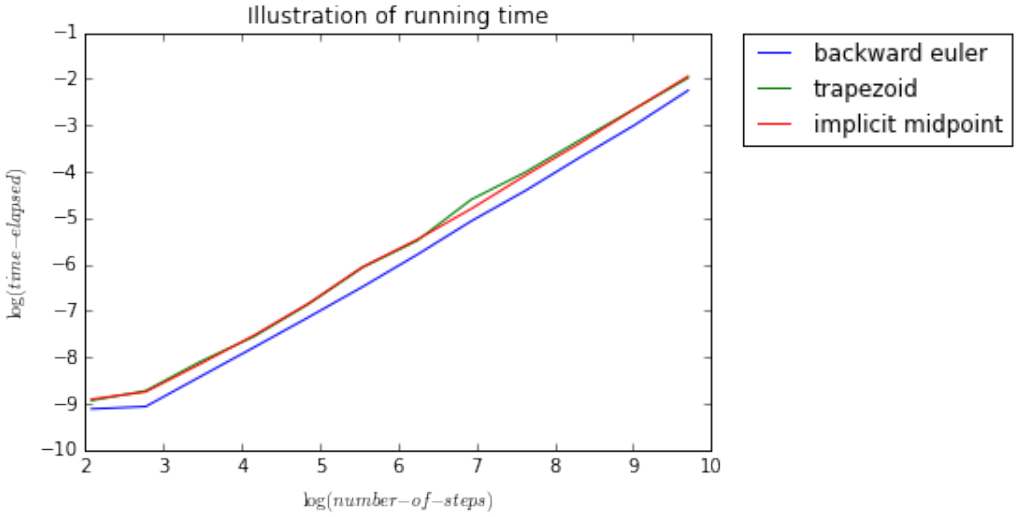
will go to infinitely faraway. For symplectic method H is bounded in a range, and the motion of the pendulum has a regular behavior. □

Problem 5. Implement implicit Euler, implicit midpoint method and trapezoid method, compare their rates of convergence and execution time.

Solution. We plot $\log(n)$ against $\log(error)$ to check the rate of convergence. Theoretically, for a method of order k, we expect to get a line with slope $-k$.

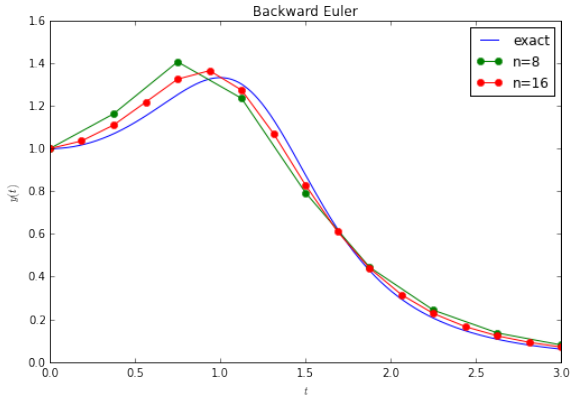


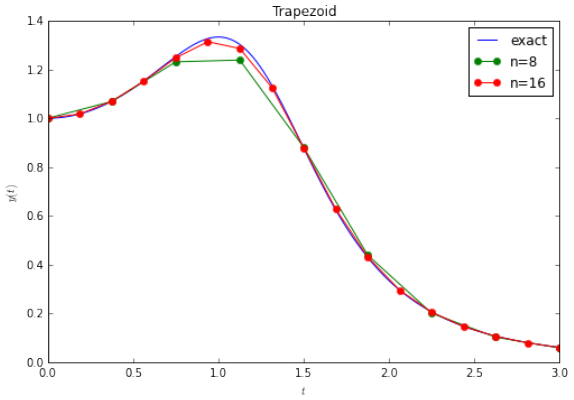
Which is indeed the case. We see that backward euler converges in order 1, other 2 methods have order 2.

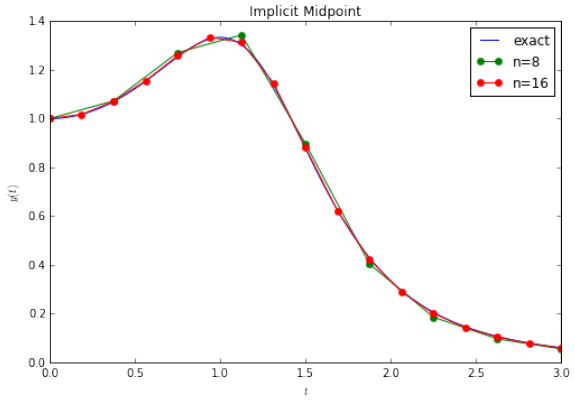


We used the *Secant Method* find roots for these implicit methods. The execution time are all $O(nm)$. Where m is the average number of iterations it takes to find the zeros of implicit updating rules.

Finally we applied these methods to the testcase 5 in last homework. And plotted the numerical solution as below.







Please also see the attached code for details.

HW3 Code

April 5, 2017

1 Numerical Solutions for DEs HW3

YANG, Ze (5131209043)

1.1 Problem 4: The non-linear pendulum.

In [1]:
```
%matplotlib inline
from __future__ import division
import time
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt

f_s = lambda q: -np.sin(q)
g_s = lambda p: p
H_1 = lambda p,q: 0.5*p*p + np.cos(q)

def explicit_euler_solve(f, g, n, t0, t1, p0, q0):
    """
    explicit euler method, solve separable Hamiltonian system
    p'(t)=-D_q H, q'(t)=D_p H, p(t_0)=p_0, q(t_0)=q_0 by

    p_{n+1} <- p_{n} + hf(p_n, q_n).
    q_{n+1} <- q_{n} + hg(p_n, q_n).
    @param f: a function of p and q, which is the - gradient of H wrt q,
        and the derivative of p.
    @param g: a function of p and q, which is the gradient of H wrt p,
        and the derivative of q.
    @param n: the number of steps.
    @param t0, p0, q0: the initial value.
    @param t1: the other end to which we generate numerical solution

    @return t: the np.array {t_k}_1^n
    @return y: the np.array {y_k}_1^n
    """
    h = (t1 - t0) / n
    t = np.linspace(t0, t1, n+1)
    p, q = np.zeros(n+1), np.zeros(n+1)
    p[0], q[0] = p0, q0
    for k in range(n):
        p[k+1] = p[k] + h*f(q[k])
        q[k+1] = q[k] + h*g(p[k])
    return t, p, q
```

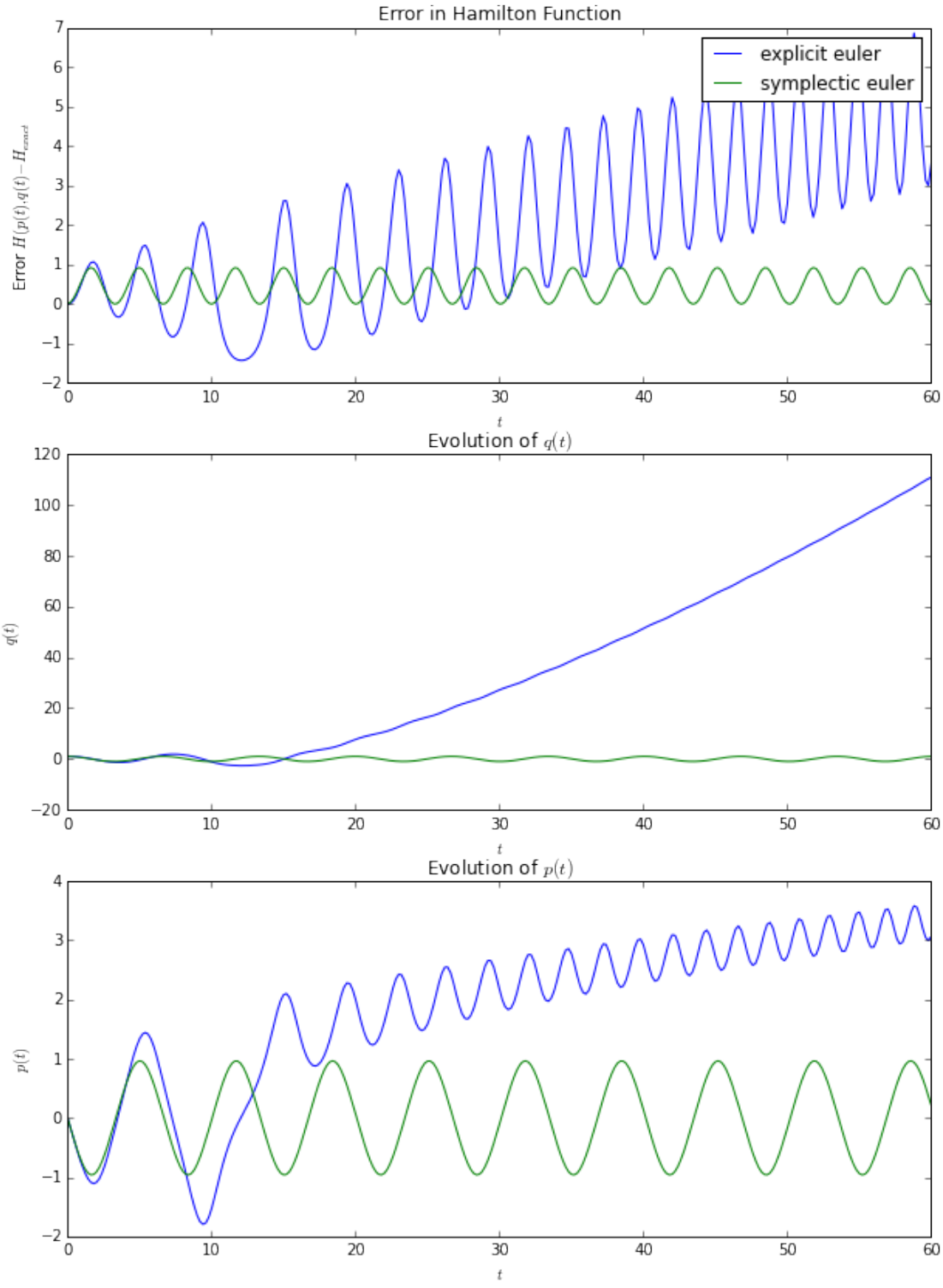
```
        def symplectic_euler_solve(f, g, n, t0, t1, p0, q0):
            """
            semi-implicit euler method, solve separable Hamiltonian system
            p'(t)=-D_q H, q'(t)=D_p H, p(t_0)=p_0, q(t_0)=q_0,
            in which H(p, q) = T(p) + V(q), by

            p_{n+1} <- p_{n} + hf(p_{q_n).
            q_{n+1} <- q_{n} + hg(p_{n+1}).
            @param f: a function of q, which is the - gradient of H wrt q,
                and the derivative of p.
            @param g: a function of p, which is the gradient of H wrt p,
                and the derivative of q.
            @param n: the number of steps.
            @param t0, p0, q0: the initial value.
            @param t1: the other end to which we generate numerical solution

            @return t: the np.array {t_k}_1^n
            @return y: the np.array {y_k}_1^n
            """
            h = (t1 - t0) / n
            t = np.linspace(t0, t1, n+1)
            p, q = np.zeros(n+1), np.zeros(n+1)
            p[0], q[0] = p0, q0
            for k in range(n):
                p[k+1] = p[k] + h*f(q[k])
                q[k+1] = q[k] + h*g(p[k+1])
            return t, p, q


        def hamilton_error(H_func, result, H_exact):
            """
            calcuate the error in Hamilton function.
            @param H_func: the hamilton function H = H(p, q)
            @param result: the result of numerical calculation y = (p, q)
            @param H_exact: the exact value of H, which is a constant.
            """
            n = len(result[0])
            err = []
            for k in range(n):
                pk, qk = result[0][k], result[1][k]
                err.append(H_func(pk, qk) - H_exact)
            return err
```

In [2]:
```
        if __name__ == '__main__':
            p0, q0 = 0, 1
            n, h = 300, 1/5
            t1, p1, q1 = explicit_euler_solve(f_s, g_s, n, 0, n*h, 0, 1)
            err_1 = hamilton_error(H_1, (p1, q1), H_1(p0, q0))

            t2, p2, q2 = symplectic_euler_solve(f_s, g_s, n, 0, n*h, 0, 1)
            err_2 = hamilton_error(H_1, (p2, q2), H_1(p0, q0))
```

In [3]:
```
        fig = plt.figure(figsize=(10,14))
```

```
ax1 = fig.add_subplot(311)
ax2 = fig.add_subplot(312)
ax3 = fig.add_subplot(313)

ax1.plot(t1, err_1)
ax1.plot(t2, err_2)
ax1.set_title('Error in Hamilton Function')
ax1.set_xlabel('$t$')
ax1.set_ylabel('Error $H(p(t), q(t)-H_{exact}$')
ax1.legend(['explicit euler', 'symplectic euler'])

ax2.plot(t1, q1)
ax2.plot(t2, q2)
ax2.set_title('Evolution of $q(t)$')
ax2.set_xlabel('$t$')
ax2.set_ylabel('$q(t)$')

ax3.plot(t1, p1)
ax3.plot(t2, p2)
ax3.set_title('Evolution of $p(t)$')
ax3.set_xlabel('$t$')
ax3.set_ylabel('$p(t)$')
```

Out[3]: <matplotlib.text.Text at 0x106b84710>


Error in Hamilton Function
explicit euler
symplectic euler
Error $H(p(t),q(t)) - H_{exact}$
t
Evolution of $q(t)$
$q(t)$
t
Evolution of $p(t)$
$p(t)$
t

1.2 Problem 5: Implicit Methods

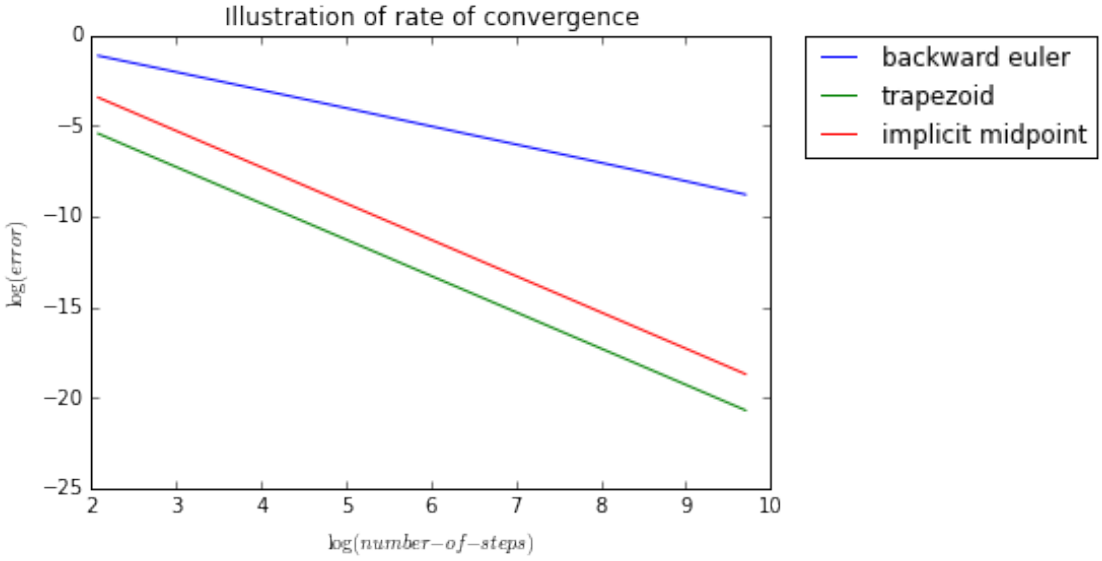
Implement implicit Euler, implicit midpoint and trapezoid method. Compare the rate of convergence and their execution time.

In [4]:
```
# test cases
test_cases = {
    1: lambda t,y: np.pi * np.cos(np.pi * t),
    2: lambda t,y: t + y,
    3: lambda t,y: np.exp(t),
    4: lambda t,y: -20*y + 20* t + 1,
    5: lambda t,y: (y**2) * (t - t**3),
    6: lambda t,y: t**2 - y
}

# initial conditions
t0, t1, y0 = 0.0, 3.0, 1.0

# the library we used in last hw.
def err_est(method, f, n_sample, t0, t1, y0):
    """
    estimate the error of an numerical solution relative
    to an exact solution, which is obtained by using a very small h.

    @param method: function, the numerical method being used.
    @param f: a function of t and y, which is the derivative of y.
    @param n_sample: # of different choices of h that are tested.
    """
    nks, errs, t_elapsed = [], [], []
    # calculate "exact" solution using a very small h
    n_large = 2**18
    t_ex, y_ex = method(f, n_large, t0, t1, y0)

    for k in range(n_sample):
        n_k = 2**(k+3)
        s_k = int(n_large/n_k)
        start_time = time.time()
        # do the numerical procudure with given choice of h,n
        t_path, path = method(f, n_k, t0, t1, y0)
        t_elapsed.append(time.time() - start_time)
        # calc errors
        path_matched = [y_ex[s_k*i] for i in range(len(path))]
        err = max(np.abs(path - path_matched))
        errs.append(err)
        nks.append(n_k)
    return nks, errs, t_elapsed

def implicit_euler_solve(f, n, t0, t1, y0):
    """
    implicit method, solve IVP y'(t)=f(t,y(t)), y(0)=y_0 by
    y_{n+1} <- y_{n} + hf(t_{n+1}, y_{n+1}).
    @param f: a function of t and y, which is the derivative of y.
    @param n: the number of steps.
    @param t0, y0: the initial value.
    @param t1: the other end to which we generate numerical solution
```

```
    @return t_path: the array {t_k}_1^n
    @return path: the array {y_k}_1^n
    """
    h = (t1 - t0) / n
    t, y = np.linspace(t0, t1, n+1), np.zeros(n+1)
    y[0] = y0
    implicit = lambda y_next, k : (
        y_next - y[k] - h * f(t[k+1], y_next)
    )
    for k in range(n):
        y[k+1] = scipy.optimize.newton(
            func=implicit, x0=y[k],
            fprime=None, args=(k,), maxiter=50)
    return t, y


def trapezoid_solve(f, n, t0, t1, y0):
    """
    trapezoid method, solve IVP y'(t)=f(t,y(t)), y(0)=y_0 by
    y_{n+1} <- y_{n} + h/2(f(t_n, y_{n}) + f(t_{n+1}, y_{n+1})).
    @param f: a function of t and y, which is the derivative of y.
    @param n: the number of steps.
    @param t0, y0: the initial value.
    @param t1: the other end to which we generate numerical solution

    @return t_path: the array {t_k}_1^n
    @return path: the array {y_k}_1^n
    """
    h = (t1 - t0) / n
    t, y = np.linspace(t0, t1, n+1), np.zeros(n+1)
    y[0] = y0
    implicit = lambda y_next, k : (
        y_next - y[k] - (h/2) * (f(t[k], y[k]) + f(t[k+1], y_next))
    )
    for k in range(n):
        y[k+1] = scipy.optimize.newton(
            func=implicit, x0=y[k],
            fprime=None, args=(k,), maxiter=50)
    return t, y


def implicit_midpoint(f, n, t0, t1, y0):
    """
    implicit midpoint method, solve IVP y'(t)=f(t,y(t)), y(0)=y_0 by
    y_{n+1} <- y_{n} + h f((t_n + t_{n+1})/2, (y_n + y_{n+1})/2).
    @param f: a function of t and y, which is the derivative of y.
    @param n: the number of steps.
    @param t0, y0: the initial value.
    @param t1: the other end to which we generate numerical solution

    @return t_path: the array {t_k}_1^n
    @return path: the array {y_k}_1^n
    """
```

```
    h = (t1 - t0) / n
    t, y = np.linspace(t0, t1, n+1), np.zeros(n+1)
    y[0] = y0
    implicit = lambda y_next, k : (
        y_next - y[k] - h * f((t[k]+t[k+1])/2, (y[k]+y_next)/2)
    )
    for k in range(n):
        y[k+1] = scipy.optimize.newton(
            func=implicit, x0=y[k],
            fprime=None, args=(k,), maxiter=50)
    return t, y
```

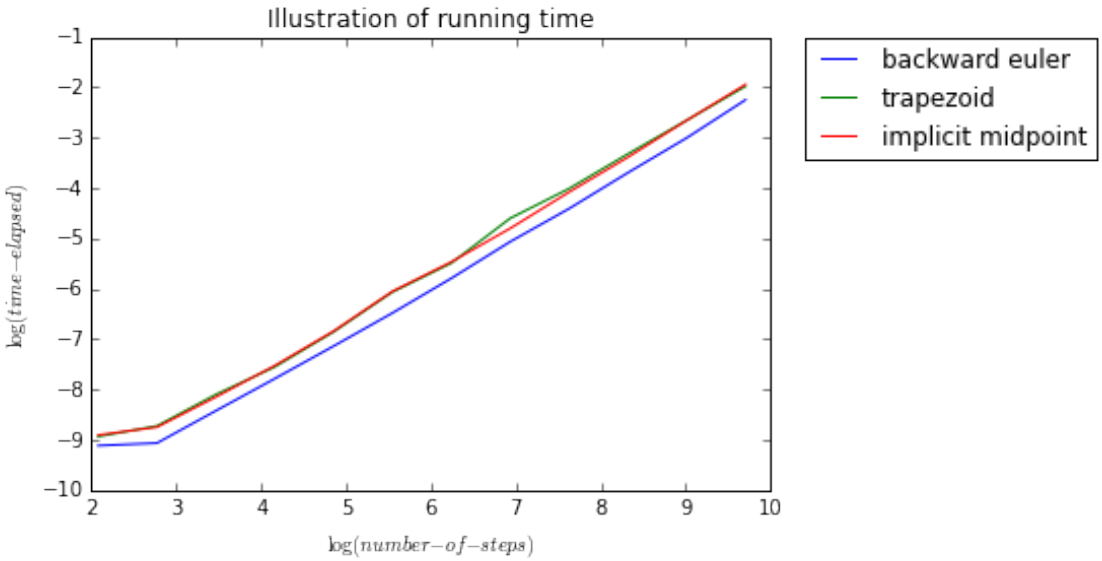
In [5]:
```
if __name__ == '__main__':
    nks ,errs, t_elapsed = err_est(implicit_euler_solve, test_cases[6], 12, t0, t1, y0)
    nks2 ,errs2, t_elapsed2 = err_est(trapezoid_solve, test_cases[6], 12, t0, t1, y0)
    nks3 ,errs3, t_elapsed3 = err_est(implicit_midpoint, test_cases[6], 12, t0, t1, y0)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.log(nks) ,np.log(errs))
    ax.plot(np.log(nks2) ,np.log(errs2))
    ax.plot(np.log(nks3) ,np.log(errs3))

    ax.set_title('Illustration of rate of convergence')
    ax.set_xlabel('$\log(number-of-steps)$')
    ax.set_ylabel('$\log(error)$')
    ax.legend(['backward euler', 'trapezoid', 'implicit midpoint'],
              bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```



In [6]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.log(nks), np.log(t_elapsed))
ax.plot(np.log(nks2), np.log(t_elapsed2))
```

```
ax.plot(np.log(nks3), np.log(t_elapsed3))
ax.set_title('Illustration of running time')
ax.set_xlabel('$\log(number-of-steps)$')
ax.set_ylabel('$\log(time-elapsed)$')
ax.legend(['backward euler', 'trapezoid', 'implicit midpoint'],
          bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

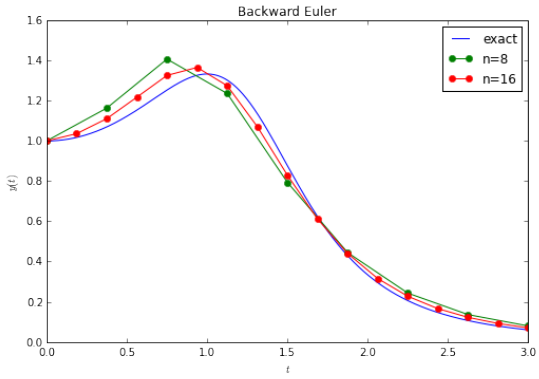
Out[6]: <matplotlib.legend.Legend at 0x10847e550>

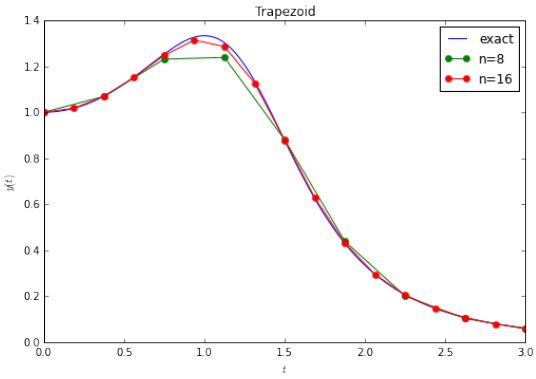


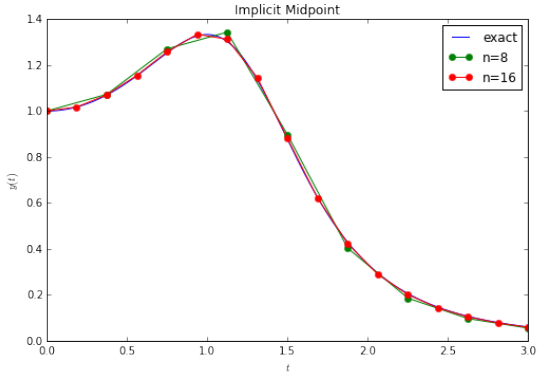
In [7]:
```
# initial conditions
fig = plt.figure(figsize=(18,12))
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)

t_ex, y_ex = implicit_euler_solve(test_cases[5], 2**16, 0.0, 3.0, 1.00)
t1, y1 = implicit_euler_solve(test_cases[5], 8, 0.0, 3.0, 1.0)
t2, y2 = implicit_euler_solve(test_cases[5], 16, 0.0, 3.0, 1.0)
ax1.plot(t_ex, y_ex)
ax1.plot(t1, y1, '.-', markersize=12)
ax1.plot(t2, y2, '.-', markersize=12)
ax1.set_title('Backward Euler')
ax1.set_xlabel('$t$')
ax1.set_ylabel('$y(t)$')
ax1.legend(['exact', 'n=8', 'n=16'])

t_ex, y_ex = trapezoid_solve(test_cases[5], 2**16, 0.0, 3.0, 1.00)
t1, y1 = trapezoid_solve(test_cases[5], 8, 0.0, 3.0, 1.0)
t2, y2 = trapezoid_solve(test_cases[5], 16, 0.0, 3.0, 1.0)
ax2.plot(t_ex, y_ex)
ax2.plot(t1, y1, '.-', markersize=12)
ax2.plot(t2, y2, '.-', markersize=12)
ax2.set_title('Trapezoid')
```

```
ax2.set_xlabel('$t$')
ax2.set_ylabel('$y(t)$')
ax2.legend(['exact', 'n=8', 'n=16'])

t_ex, y_ex = implicit_midpoint(test_cases[5], 2**16, 0.0, 3.0, 1.00)
t1, y1 = implicit_midpoint(test_cases[5], 8, 0.0, 3.0, 1.0)
t2, y2 = implicit_midpoint(test_cases[5], 16, 0.0, 3.0, 1.0)
ax3.plot(t_ex, y_ex)
ax3.plot(t1, y1, '.-', markersize=12)
ax3.plot(t2, y2, '.-', markersize=12)
ax3.set_title('Implicit Midpoint')
ax3.set_xlabel('$t$')
ax3.set_ylabel('$y(t)$')
ax3.legend(['exact', 'n=8', 'n=16'])
```

Out[7]: <matplotlib.legend.Legend at 0x108918b90>







In [ ]: