Individuālais darbs (Nr10)

Jānis Erdmanis

December 9, 2013

In [27]:
```
! ipython3 nbconvert --to=latex "Individuālais darbs.ipynb"

# \usepackage{xltxtra}
# \usepackage{fontspec}
# -- \usepackage{fontenc} --
```

```
[NbConvertApp] Using existing profile dir:
'/home/akels/.config/ipython/profile_default'
[NbConvertApp] Converting notebook Individuālais darbs.ipynb to latex
[NbConvertApp] Support files will be in Individuālais darbs_files/
[NbConvertApp] Loaded template latex_article.tplx
[NbConvertApp] Writing 82786 bytes to Individuālais darbs.tex
```

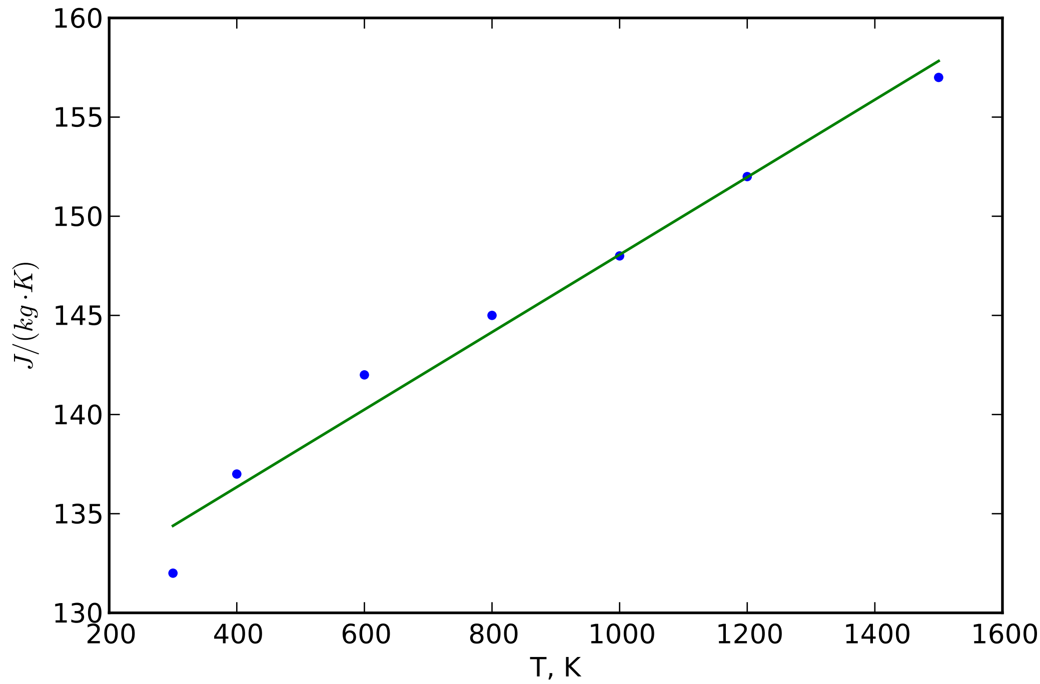
In [1]:
```
%config InlineBackend.figure_format = 'svg'
```

In [2]:
```
from pylab import *
from monochrome import setFigLinesBW

def legend(**kw):
    from pylab import legend as Oldlegend
    setFigLinesBW(gcf())
    Oldlegend(**kw)
```

0.1 Parametri no eksperimentālajiem datiem

In [3]:
```
T = array([300,400,600,800,1000,1200,1500])
cp_ = array([132,137,142,145,148,152,157])

fig = figure()
xlabel('T, K')
ylabel(r'$J/(kg \cdot K)$')
plot(T,cp_,'.')

from scipy.optimize import curve_fit

def cp(T,k,c0): return k*T + c0
par,_ = curve_fit(cp,T,cp_ )
plot(T,cp(T,*par))
print(par)
```

```
[  1.95340050e-02   1.28528967e+02]
```



0.2 Bezdimensionalizācija

Bezdimensionalizēju iepriekšējo funkciju un vienādojumu ar:

$$T = T_F \tilde{T} c_p = c_p(T_F)\tilde{c}_p t = \frac{d\,\rho \tilde{c}_p}{2\sigma T_F^3}\tilde{t}$$

Tādēļ risināmais vienādojums:

$$\frac{d\tilde{T}}{d\tilde{t}} = \frac{1-\tilde{T}^4}{\tilde{T}\frac{d\tilde{c}_p}{d\tilde{T}} + \tilde{c}_p}$$

0.3 Integrētāja prototips

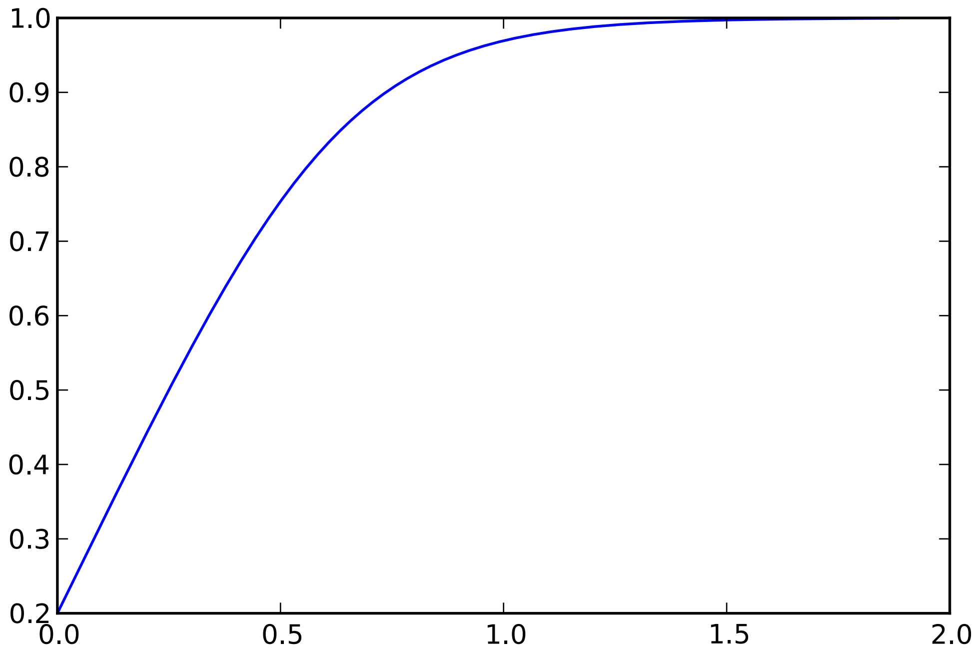
In [4]:
```
class Integrate:
    """
   Izveidota, lai būtu ērti manīt parametrus un veikt automatisku bezdimensionalizāciju.
    """

    defaults = dict(

    T_F = 1500, # K
    cp_r = 157.83, # J/(kg*K), cp(T_F)
    t_r = 15.92, # sek

    h_SI = 1, # sek
    tmax_SI = 30, # sek
    T0_SI = 300 # K
    )

    def __init__(self,specific={}):
```

```
        parametri =Integrate.defaults.copy()
        parametri.update(specific)

        self.__dict__.update(parametri) # ievieto parametrus objektā
        self.bezdim(**parametri)

    def bezdim(self,T0_SI,T_F,tmax_SI,t_r,h_SI,cp_r,**kw):
        """
        Bezdimensionalizē parametrus
        """

        def cp_t(T):
            T_SI = T*T_F
            cp_SI = cp(T,*par)
            return cp_SI/cp_r

        dict_up = dict(

        T0 = T0_SI/T_F,
        tmax = tmax_SI/t_r,
        h = h_SI/t_r,

        dcp = par[0]*t_r/cp_r,
        cp = cp_t
        )

        self.__dict__.update(dict_up)
```

0.4 Atrisinājums ar iebūvēto integrētāju

In [5]:

```
class Ieb_int(Integrate):

    def integr(self):

        cp = self.cp
        dcp = self.dcp
        T0 = self.T0
        tmax = self.tmax

        def f(T,t):

            DT = (1 - T**4)/(T*dcp + cp(T))
            return DT

        from scipy.integrate import odeint

        #T0 = 300/T_F
        t = linspace(0,tmax,1000)
        T = odeint(f,T0,t)

        plot(t,T)

gra = Ieb_int(dict(tmax_SI = 30))
gra.integr()
```

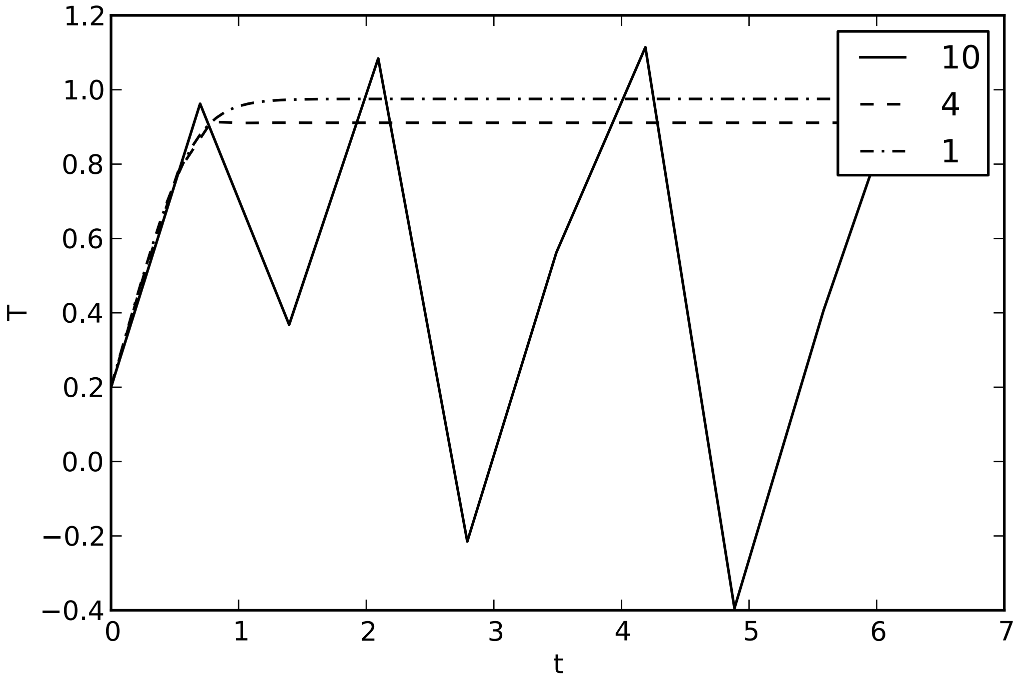


1 (T2) 2. kārtas Teilora metode

In [6]:
```
class Teilora(Integrate):

    def integr(self):

        cp = self.cp
        dcp = self.dcp
        T0 = self.T0
        tmax = self.tmax
        h = self.h

        #try:
        #    N = self.N
        #except:
        N = int(tmax/h)

        tpoints = linspace(0,tmax,N)
        Tpoints = empty(tpoints.shape)

        Tpoints[0] = T0

        for n,t in enumerate(tpoints[:-1]):

            Tn = Tpoints[n]

            DTn =  (1 - Tn**4)/(Tn*dcp + cp(Tn))
            D2Tn = (-2*Tn**4*dcp - 4*Tn**3*cp(Tn) - 2*dcp) / (Tn*dcp + cp(Tn))

            Tpoints[n + 1] = Tn + h*DTn + h**2/2 * D2Tn

        return tpoints,Tpoints
```

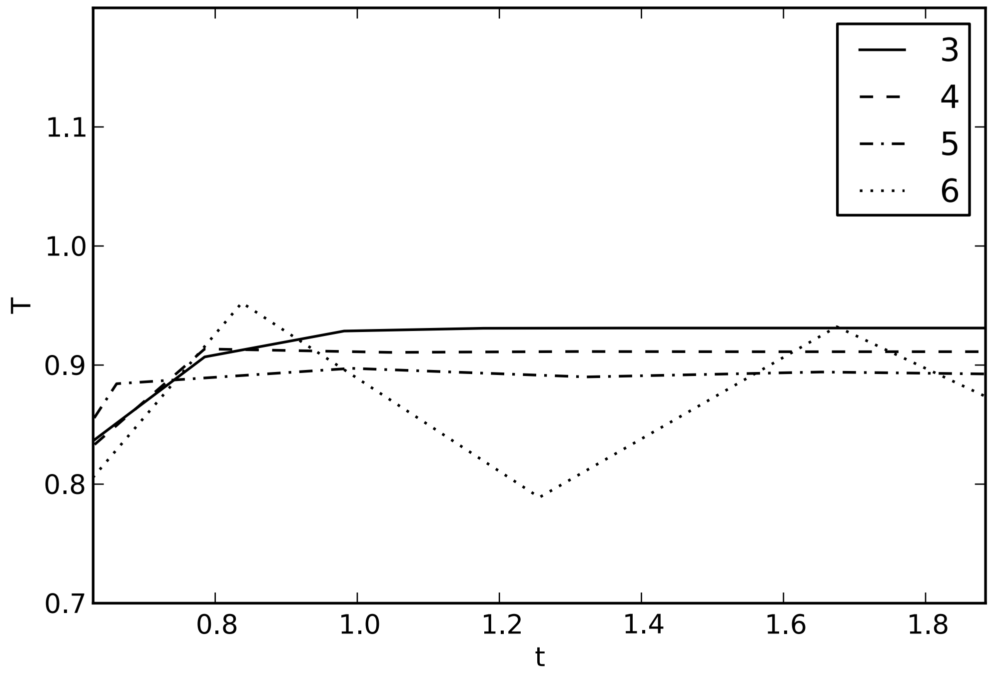
1.1 1.a

In [7]:
```
fig = figure()
#ax = fig.gca()
xlabel('t')
ylabel('T')

for hi in [10,4,1]:
    teil = Teilora(dict(h_SI=hi,tmax_SI=100))
    t,T = teil.integr()
    plot(t,T,label=hi)

legend()
```

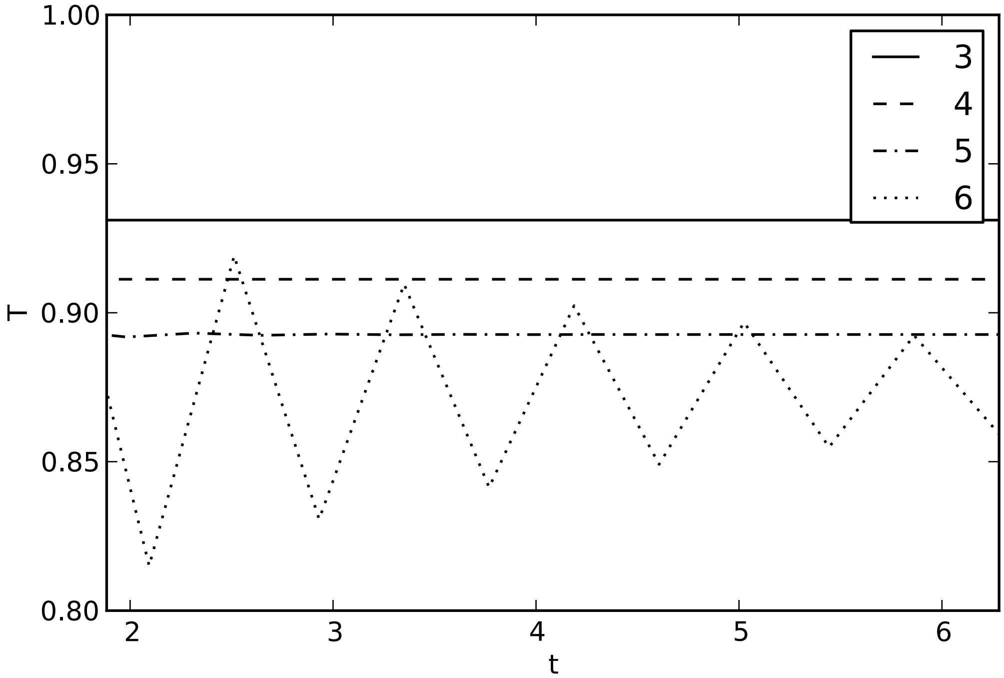


1.2 1.b

In [8]:
```
fig = figure()
#ax = fig.gca()
xlabel('t')
ylabel('T')
t_r  = 15.92
xlim(10/t_r,30/t_r)
ylim(0.7,1.2)

for hi in range(3,7):
    teil = Teilora(dict(h_SI=hi,tmax_SI=100,t_r=t_r))
    t,T = teil.integr()
    plot(t,T,label=hi)

legend()
```



In [9]:
```
fig = figure()
#ax = fig.gca()
xlabel('t')
ylabel('T')
t_r  = 15.92
xlim(30/t_r,100/t_r)
ylim(0.8,1)

for hi in range(3,7):
    teil = Teilora(dict(h_SI=hi,tmax_SI=100,t_r=t_r))
    t,T = teil.integr()
    plot(t,T,label=hi)

legend()
```



1.3 1.c

In [10]:
```
for hi in [0.5,0.8,1,2,3,4,5,10,20][::-1]:

    teil = Teilora(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = teil.integr()

    teil = Teilora(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = teil.integr()

    err = max(abs((T1-T2[::2])/T1))
    print ('hi = {0}  err = {1:.4f}'.format(hi,err))
```

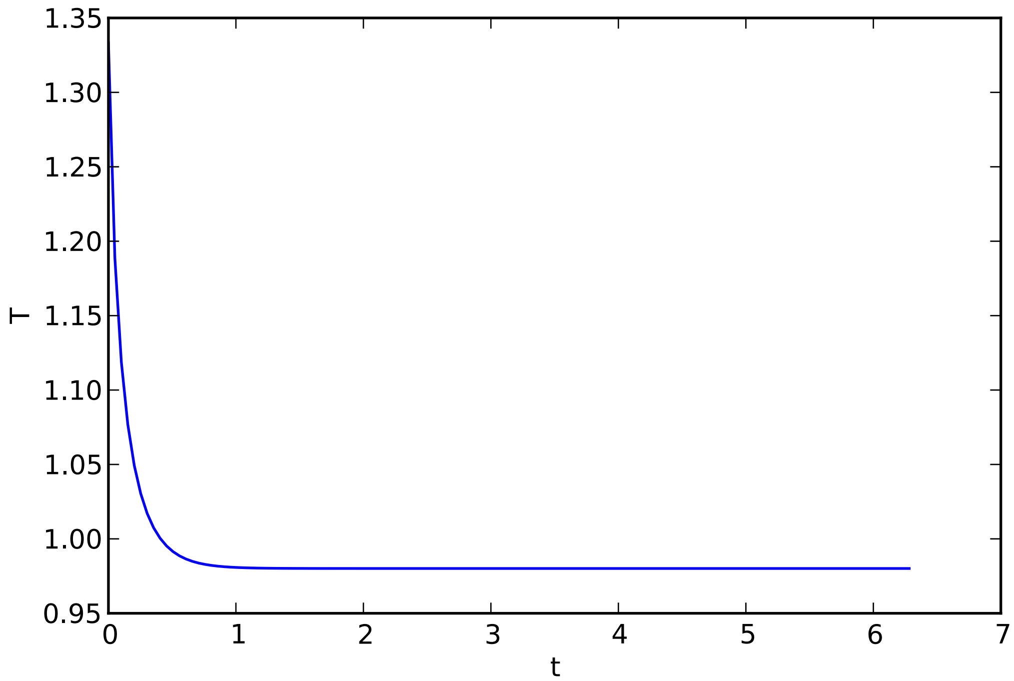
```
hi = 20  err = 1.0000
hi = 10  err = 5.1508
hi = 5  err = 0.0574
hi = 4  err = 0.0452
hi = 3  err = 0.0350
hi = 2  err = 0.0241
hi = 1  err = 0.0124
hi = 0.8  err = 0.0100
hi = 0.5  err = 0.0063
```

Tātad solis h=0.8

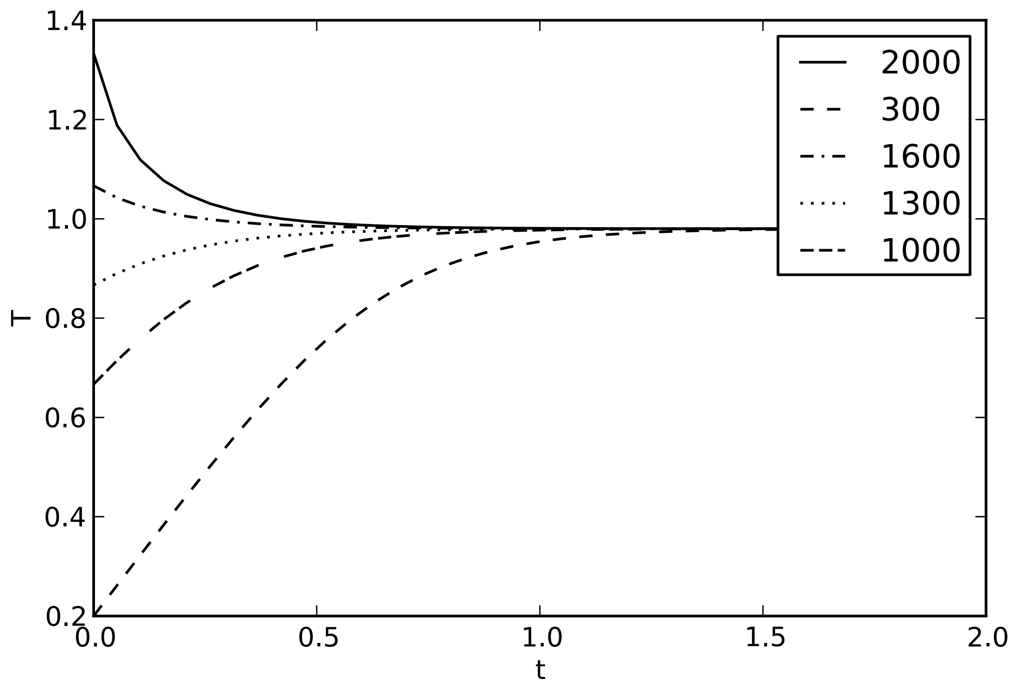
1.4 1.d

In [11]:
```
T0 = 2000 # K

fig = figure()
#ax = fig.gca()
xlabel('t')
ylabel('T')

teil = Teilora(dict(h_SI=0.8,tmax_SI=100,T0_SI = 2000))
t,T = teil.integr()
plot(t,T)
```

Out [11]:
```
[<matplotlib.lines.Line2D at 0xb0b2860c>]
```



In [12]:
```
fig = figure()
xlabel('t')
ylabel('T')


for T0 in [2000,300,1600,1300,1000]:
    teil = Teilora(dict(h_SI=0.8,tmax_SI=30,T0_SI = T0))
    t,T = teil.integr()
    plot(t,T,label=T0)

legend()
```



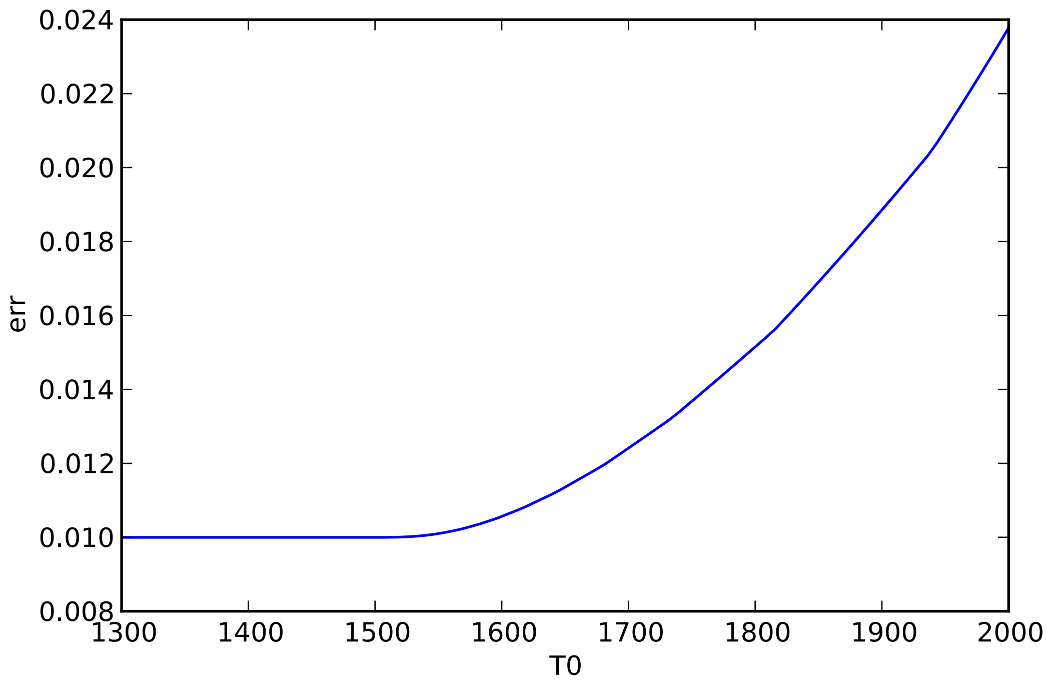
- Ja plāksnes temperatūra ir **mazāka** par apkārtējās vides temperatūru, tad plāksne siltumu **saņem**, līdz iegūst vides temperatūru

- Ja plāksnes temperatūra ir **lielāka** par apkārtējās vides temperatūru, tad plāksne siltumu **zaudē**, līdz iegūst vides temperatūru
- Redzams arī, ka izmantotā laika mērogošana ir noderīga, jo ja $\tilde{t} < 1$, tad plāksnes temperatūra būtiski atšķiras no vides temperatūras

In [13]:
```
err_list = []
T0_list = linspace(1300,2000,100)

for T0 in T0_list:

    teil = Teilora(dict(h_SI=0.8,tmax_SI=100,T0_SI = T0))
    t1,T1 = teil.integr()

    teil = Teilora(dict(h_SI=0.8/2,tmax_SI=100,T0_SI = T0))
    t2,T2 = teil.integr()

    err = max(abs((T1-T2[::2])/T1))
    err_list.append(err)
    #print ('T0 = {0} \t err = {1:.3f}'.format(T0,err))

fig = figure()
xlabel('T0')
ylabel('err')
plot(T0_list,err_list)
```

```
[<matplotlib.lines.Line2D at 0xb06db80c>]
```

Out [13]:



2 (RK2) 2. kārtas Rungas-Kutas metode

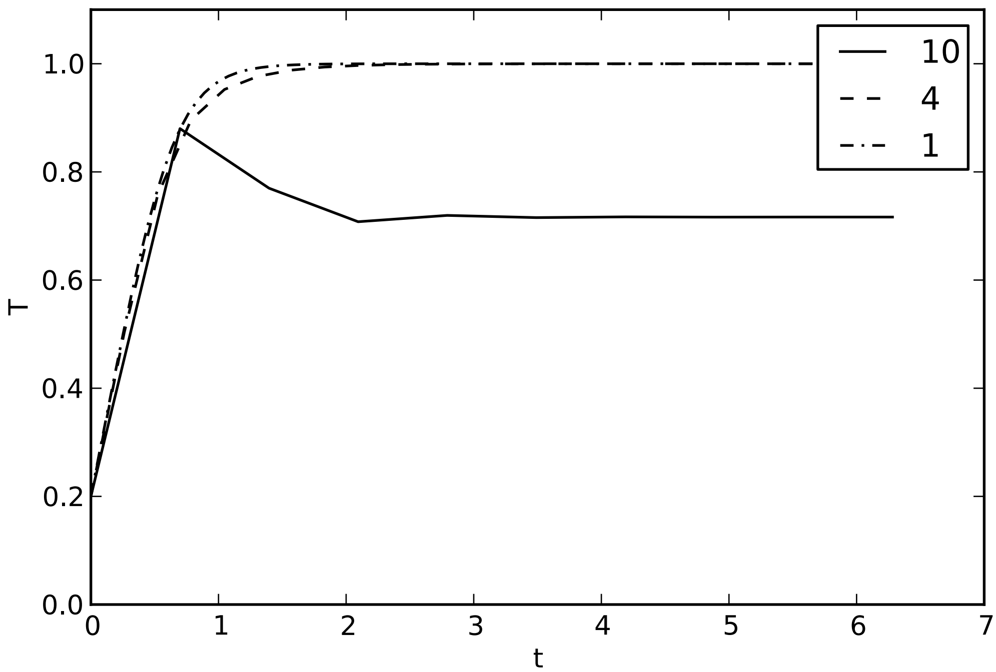
In [14]:
```
class RK2(Integrate):

    def integr(self):

        cp = self.cp
        dcp = self.dcp
        T0 = self.T0
        tmax = self.tmax
        h = self.h
```

```
        N = int(tmax/h)

        tpoints = linspace(0,tmax,N)
        Tpoints = empty(tpoints.shape)

        Tpoints[0] = T0

        def f(Tn,t): return (1 - Tn**4)/(Tn*dcp + cp(Tn))

        for n,t in enumerate(tpoints[:-1]):

            Tn = Tpoints[n]

            k1 = h*f(Tn,t)
            k2 = h*f(Tn + 1/2*k1,t +1/2*h)

            Tpoints[n+1] = Tn + k2

            #k1 = Tn + 2/3*h*f(Tn,t)

            #Tpoints[n + 1] = Tn + h/4*f(Tn,t) + 3/4*h*f(k1,t + 2/3*h)

        return tpoints,Tpoints
```

2.1 2.a

In [15]:

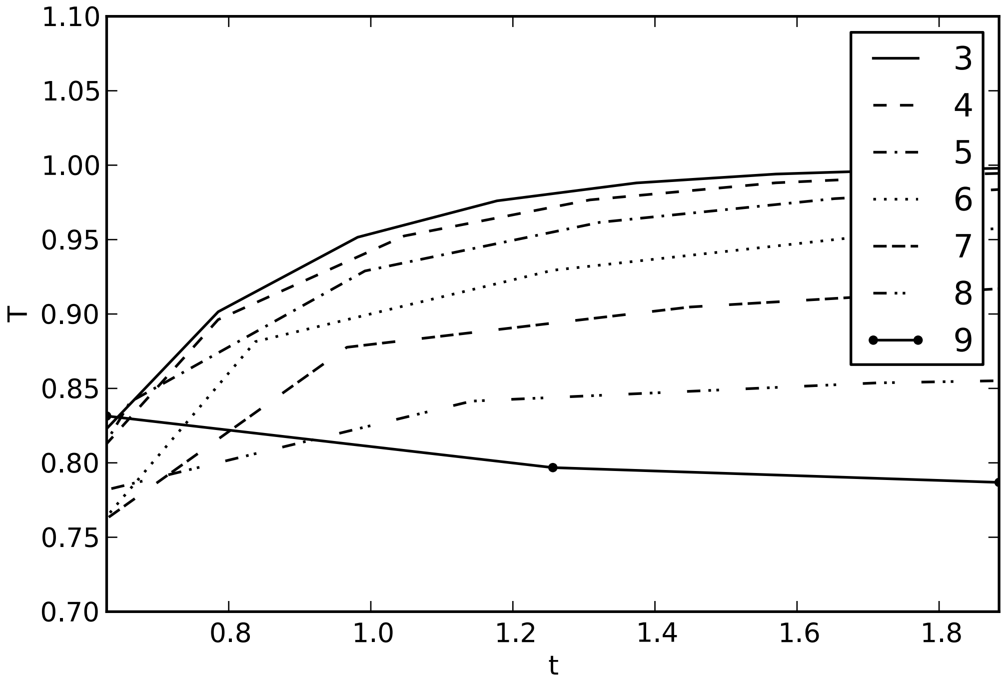
```
fig = figure()
ylim(0,1.1)
xlabel('t')
ylabel('T')

for hi in [10,4,1]:
    runga = RK2(dict(h_SI=hi,tmax_SI=100))
    t,T = runga.integr()
    plot(t,T,label=hi)

legend()
```



2.2 2.b

In [16]:

```
fig = figure()
#ax = fig.gca()
xlabel('t')
ylabel('T')
t_r  = 15.92
xlim(10/t_r,30/t_r)
ylim(0.7,1.1)

for hi in range(3,10):
    runga = RK2(dict(h_SI=hi,tmax_SI=100,t_r=t_r))
    t,T = runga.integr()
    plot(t,T,label=hi)

legend()
```



2.3 2.c

In [17]:

```
for hi in [0.5,0.8,1,2,3,4,5,10,20][::-1]:

    runga = RK2(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = runga.integr()

    runga = RK2(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = runga.integr()

    err = max(abs((T1-T2[::2])/T1))
    print ('hi = {0}  err = {1:.6f}'.format(hi,err))
```

```
hi = 20  err = 12.050398
hi = 10  err = 0.395623
hi = 5  err = 0.027610
hi = 4  err = 0.015290
hi = 3  err = 0.007060
hi = 2  err = 0.002553
hi = 1  err = 0.000539
hi = 0.8  err = 0.000335
```

```
hi = 0.5  err = 0.000124
```

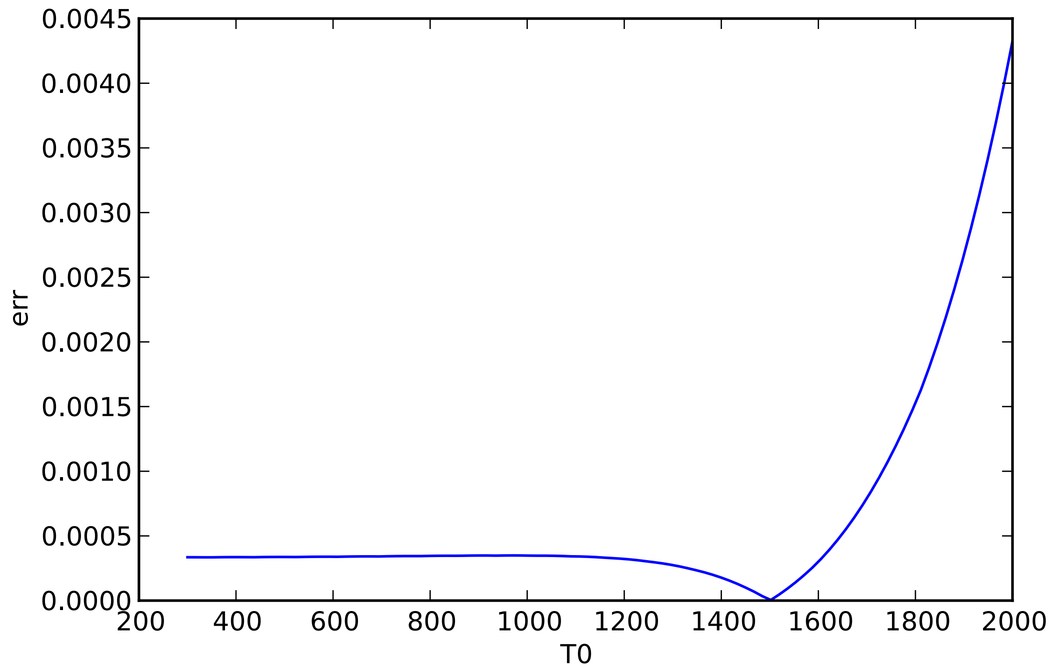
2.4 2.d

In [18]:

```
err_list = []
T0_list = linspace(300,2000,100)

for T0 in T0_list:

    runga = RK2(dict(h_SI=0.8,tmax_SI=100,T0_SI = T0))
    t1,T1 = runga.integr()

    runga = RK2(dict(h_SI=0.8/2,tmax_SI=100,T0_SI = T0))
    t2,T2 = runga.integr()

    err = max(abs((T1-T2[::2])/T1))
    err_list.append(err)
    #print ('T0 = {0} \t err = {1:.3f}'.format(T0,err))

fig = figure()
xlabel('T0')
ylabel('err')
plot(T0_list,err_list)
```

```
[<matplotlib.lines.Line2D at 0xb13ba40c>]
```

Out [18]:



2.5 Kopīgais un atšķirīgais starp RK2 un T2 metodēm

Kopīgs:

- Vienāda lokālā aproksimācijas kārta $O(h^2)$
- Abas ir viensoļu metodes, kurām approksimācijas kļūda T_{n+1} ir atkarīga no T_n, bet ne no $T_{n-1}, T_{n-2}, \ldots$
- Uzreiz var pielietot zinot tikai vienu punktu (t_0, T_0)
- Soļa izmēru katrā solī var mainīt nemainot koeficientu vērtības, kas ir noderīgi adaptīvajām metodēm

Atšķirīgs:

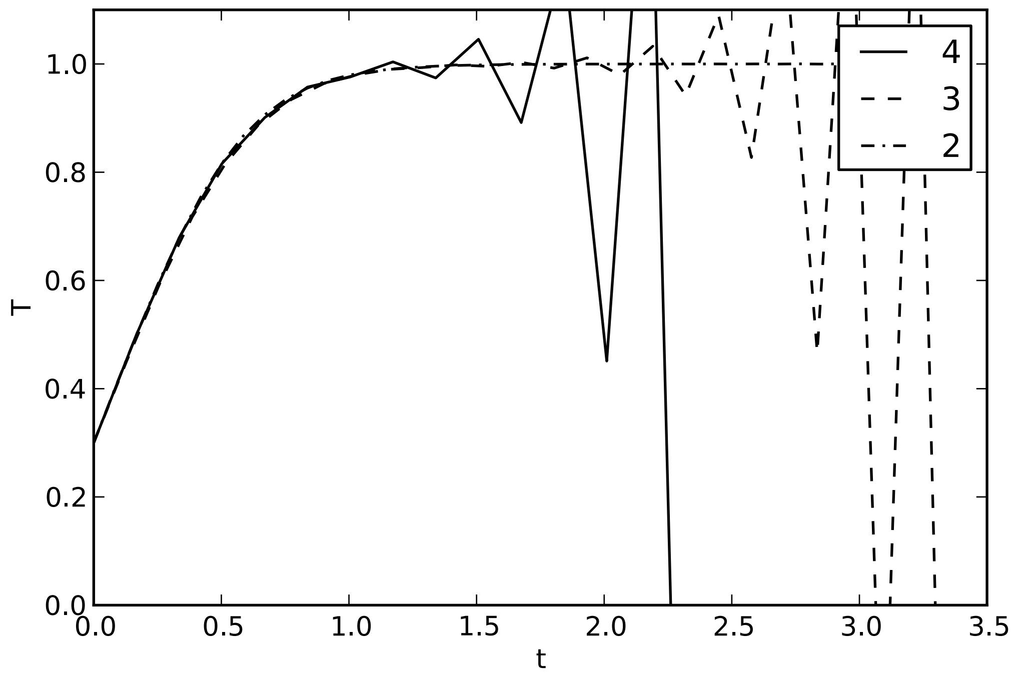
- RK2 metodei nepieciešams, lai integrējamā funkcija $f = f(T, t)$ būtu definēta arī laika brīžos $(t_{n+1} + t_n)/2$
- T2 metodi var pielietot tikai tad, ja $f = f(T, t)$ var vismaz vienreiz atvasināt pēc laika

3 (AB4) Adamsa-Bašforda 4. kārtas metode

In [19]:
```
class AB4(Integrate):

    def integr(self):

        cp = self.cp
        dcp = self.dcp
        T0 = self.T0
        tmax = self.tmax
        h = self.h

        def f(Tn,t): return (1 - Tn**4)/(Tn*dcp + cp(Tn))



        N = int(tmax/h)

        tpoints = linspace(0,tmax,N+1) # N?
        Tpoints = zeros(tpoints.shape)

        Tpoints[0] = T0

        for n,t in enumerate(tpoints[:3]):

           # Tn = Tpoints[n]
            #k1 = Tn + 2/3*h*f(Tn,t)

            #Tpoints[n + 1] = Tn + h/4*f(Tn,t) + 3/4*h*f(k1,t + 2/3*h)

            Tn = Tpoints[n]

            k1 = h*f(Tn,t)
            k2 = h*f(Tn + 1/2*k1,t +1/2*h)

            Tpoints[n+1] = Tn + k2


        for n in range(0,len(tpoints)-4): # [1:-1]

            T0 = Tpoints[n]
            T1 = Tpoints[n+1]
            T2 = Tpoints[n+2]
            T3 = Tpoints[n+3]

            t0 = tpoints[n]
            t1 = tpoints[n+1]
            t2 = tpoints[n+2]
            t3 = tpoints[n+3]

            f0 = f(T0,t0)
            f1 = f(T1,t1)
            f2 = f(T2,t2)
            f3 = f(T3,t3)

            # Indexing is shifted to left since tpoints has striped first value
            # however it coincides with wikipedia notation

            #y = y + h/24*( 55*f(tn(1,i), yn(1,i)) - 59*f(tn(1,i-1), yn(1,i-1))
```

```
            #     + 37*f(tn(1,i-2), yn(1,i-2)) - 9*f(tn(1,i-3), yn(1,i-3)) )

            Tpoints[n+4] = T3 + h/24*( 55*f3 - 59*f2 + 37*f1 - 9*f0 )


        return tpoints,Tpoints
```

In [24]:
```
fig = figure()
ylim(0,1.1)
xlabel('t')
ylabel('T')

for hi in [4,3,2]:
    adams = AB4(dict(h_SI=hi,tmax_SI=80,T_F=1000,cp_r=157.83,t_r=23.88))
    t,T = adams.integr()
    plot(t,T,label=hi)

legend()
```



3.1 3.b

In [25]:
```
for hi in [0.5,0.8,1,2,3,4][::-1]:

    adams = AB4(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = adams.integr()

    adams = AB4(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = adams.integr()

    err = max(abs((T1-T2[::2])/T1))
    print ('hi = {0}  err = {1:.8f}'.format(hi,err))
```

```
hi = 4   err = 1.49861340
hi = 3   err = 1.48846732
hi = 2   err = 11.60643150
hi = 1   err = 0.00015137
hi = 0.8   err = 0.00006659
hi = 0.5   err = 0.00001571
-c:11: RuntimeWarning: overflow encountered in double_scalars
-c:11: RuntimeWarning: invalid value encountered in double_scalars
-c:9: RuntimeWarning: invalid value encountered in true_divide
```

4 (RK4) metode

In [26]:
```
class RK4(Integrate):

    def integr(self):

        cp = self.cp
        dcp = self.dcp
        T0 = self.T0
        tmax = self.tmax
        h = self.h

        N = int(tmax/h)

        tpoints = linspace(0,tmax,N)
        Tpoints = empty(tpoints.shape)

        Tpoints[0] = T0

        def f(Tn,t): return (1 - Tn**4)/(Tn*dcp + cp(Tn))

        for n,t in enumerate(tpoints[:-1]):

            Tn = Tpoints[n]

            k1 = h*f(Tn,t)
            k2 = h*f(Tn + 1/2*k1,t+1/2*h)
            k3 = h*f(Tn + 1/2*k2,t + 1/2*h)
            k4 = h*f(Tn + k3,t+h)


            Tpoints[n + 1] = Tn + 1/6*(k1 + 2*k2 + 2*k3 + k4)

        return tpoints,Tpoints
```

In [27]:
```
for hi in [0.5,0.8,1,2,3,4,10,20][::-1]:

    runga = RK4(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = runga.integr()

    runga = RK4(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = runga.integr()

    err = max(abs((T1-T2[::2])/T1))
    print ('hi = {0}  err = {1:.8f}'.format(hi,err))
```

```
hi = 20   err = 1.75677594
hi = 10   err = 0.30872700
hi = 4   err = 0.00150002
hi = 3   err = 0.00040079
hi = 2   err = 0.00006628
hi = 1   err = 0.00000347
```

```
hi = 0.8  err = 0.00000137
hi = 0.5  err = 0.00000020
-c:18: RuntimeWarning: overflow encountered in double_scalars
-c:18: RuntimeWarning: invalid value encountered in double_scalars
```

5 Visu metožu kļūdu funkcijas $err(h)$ apkopojums

In [28]:

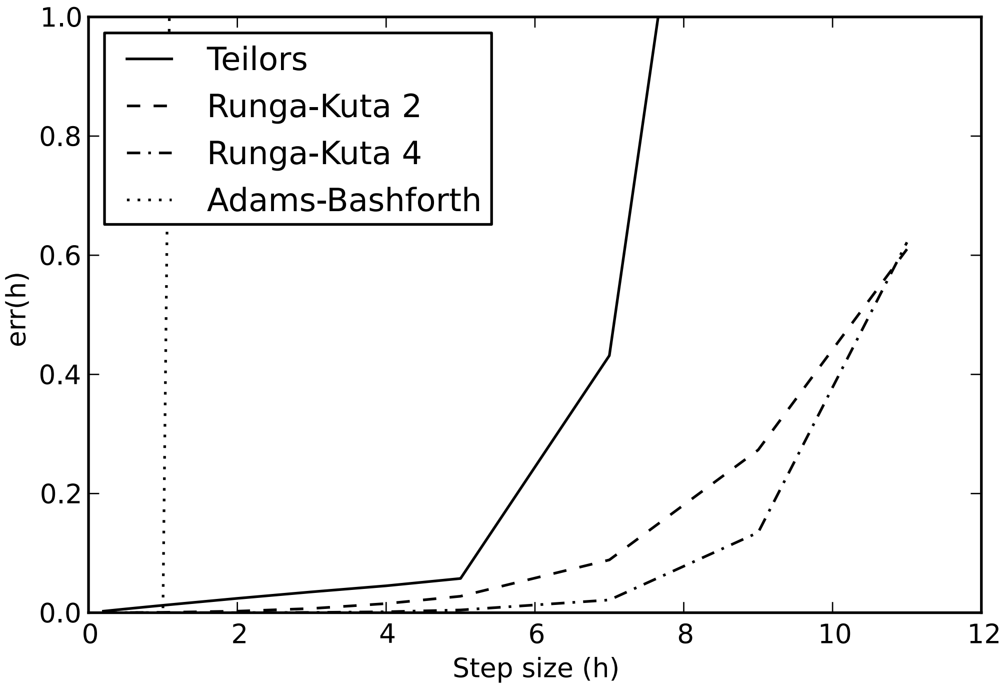
```
hrange = [0.2,0.5,0.8,0.9,1,2,3,4,5,7,9,11]

teilorh = []
for hi in hrange:

    teil = Teilora(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = teil.integr()

    teil = Teilora(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = teil.integr()

    err = max(abs((T1-T2[::2])/T1))
    teilorh.append(err)

rungah = []
for hi in hrange: #[0.5,0.8,1,2,3,4,5,10,20][::-1]:

    runga = RK2(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = runga.integr()

    runga = RK2(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = runga.integr()

    err = max(abs((T1-T2[::2])/T1))
    rungah.append(err)

adamsh = []
for hi in hrange: #[0.5,0.8,1,2,3,4][::-1]:

    adams = AB4(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = adams.integr()

    adams = AB4(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = adams.integr()

    err = max(abs((T1-T2[::2])/T1))
    adamsh.append(err)

runga4h = []
for hi in hrange: #[0.5,0.8,1,2,3,4,10,20][::-1]:

    runga = RK4(dict(h_SI=hi,tmax_SI=100))
    t1,T1 = runga.integr()

    runga = RK4(dict(h_SI=hi/2,tmax_SI=100))
    t2,T2 = runga.integr()

    err = max(abs((T1-T2[::2])/T1))
    runga4h.append(err)
```

```
-c:36: RuntimeWarning: invalid value encountered in true_divide
-c:36: RuntimeWarning: invalid value encountered in subtract
```

In [29]:

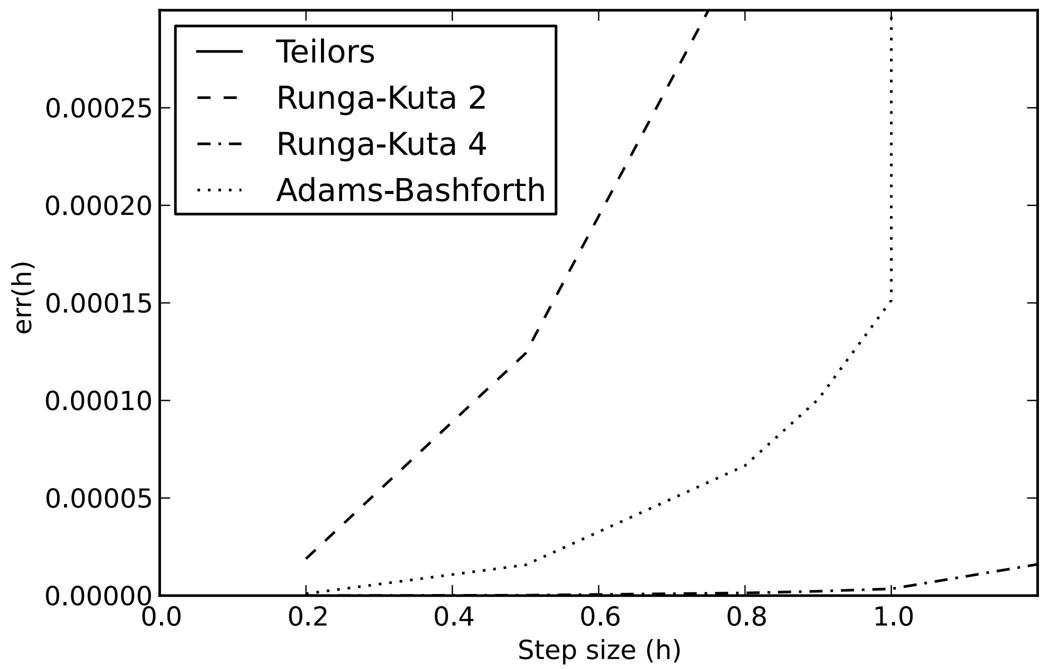
```
fig = figure()
xlabel('Step size (h)')
ylabel('err(h)')
ylim(0,1)

plot(hrange,teilorh,label='Teilors')
```

```
plot(hrange,rungah,label='Runga-Kuta 2')
plot(hrange,runga4h,label='Runga-Kuta 4')
plot(hrange,adamsh,label='Adams-Bashforth')

legend(loc=2)
```



In [26]:

```
fig = figure()
xlabel('Step size (h)')
ylabel('err(h)')
ylim(0,0.0003)
xlim(0,1.2)

plot(hrange,teilorh,label='Teilors')
plot(hrange,rungah,label='Runga-Kuta 2')
plot(hrange,runga4h,label='Runga-Kuta 4')
plot(hrange,adamsh,label='Adams-Bashforth')

legend(loc=2)
```



6 Skaitlisko metožu efektivitātes analīze

- Visefektīvākā metode ir *Runga-Kuta* 4. kārtas metode, kura rada vismazāko kļūdu risinājumā, kas bija sagaidāms, jo tai ir augstākā vienādojuma approksimācijas kārta jeb $O(h^4)$.
- Ja tiek izmantots laika solis, kas ir mazāks par 1 sek, tad labus rezultātus dos *Ādamsa-Bašforda* metode, bet pie $h > 1\,sek$ var novērot, ka shēma ir nestabila.
- Ja sākotnējā plāksnes temperatūra ir liela (šajā gadījumā $T > 2000K$), tad *T2* metode rada mazāku kļūdu nekā *RK2* metode.