```
%pylab
%load_ext sympyprinting
```

```
Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.
```

```
import time
import pickle
from glob import glob

import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import norm

# symbolic math
import sympy as sym
from sympy.abc import y, B
from sympy import pi

# jit compiled loops
from numba import autojit, jit, double, int_
```

0.1 Determine source term for code verification study

```
Ty = sym.Function('Ty')
Ty = 300 + 200 * sym.sin(3 * pi * y / (2 * B))
Ty
```

```
 2 0 0
 o p e r a t o r n a m e s i n  l e f t (  f r a c 3  2
 f r a c  p i
y  B  r i g h t )
+
3 0 0


     [U+239B]3⋅⋅y[U+239E]
200⋅sin[U+239C][U+2500][U+2500][U+2500][U+2500][U+2500][U+239F] + 300
     [U+239D] 2⋅B [U+23A0]
```

```
dTy = sym.diff(Ty, y, 1)
dTy
```

```
 3 0 0
 f r a c  p i
 o p e r a t o r n a m e c o s  l e f t (  f r a c 3  2
 f r a c  p i
y  B  r i g h t )  B
```

```
          [U+239B]3··y[U+239E]
300··cos[U+239C][U+2500][U+2500][U+2500][U+2500][U+2500][U+239F]
          [U+239D] 2·B [U+23A0]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]
       B
```

```
ddTy = sym.diff(Ty, y, 2)
ddTy
```

1 Common functions

```python
def grid_pts(ngrid, x0=0, x1=1):
    """Return evenly spaced grid points over the given interval.

    Inputs:
        ngrid (int) - number of grid points
        x0 (float, optional) - start of grid
        x1 (float, optional) - end of grid

    Outputs:
        grid (array of floats) - vector of grid points
        dx (float) - grid spacing
    """

    grid = np.linspace(x0, x1, ngrid)
    assert np.allclose(x0, grid[0])
    assert np.allclose(x1, grid[-1])
    dx = np.diff(grid)[0]

    return grid, dx


def ooa(de2, de1, r=2.):
    """Observed order of accuracy.

    Inputs:
        de2 (float) - DE error of coarse mesh
        de1 (float) - DE error of fine mesh
        r (float, default=2.) - grid refinement factor

    Outputs:
        phat (float) - observed order of accuracy
    """

    return np.log(de2 / de1) / np.log(r)


def find_phat(DE, r=2.):
    """Find the observed order of accuracy.

    Inputs:
        DE (array) - discretization error using L2 and Linf norms
        r (float, default=2.) - refinement factor

    Outputs:
        phat (array) - observed order of accuracy
    """

    npts = len(DE) - 1
    phat = np.zeros((npts, 2))
```

```
    for i in range(npts):
        phat[i] = ooa(DE[i, 0], DE[i+1, 0], r=r), ooa(DE[i, 1], DE[i+1, 1], r=r)

    return phat


def find_h(grids):
    """Find the normalized grid spacing ‘h’.

    Inputs:
        grids (array) - possible grid shapes, from smallest to largest

    Outputs:
        h (array, int) - normalized grid spacing.
    """

    dx = np.zeros(grids.shape)
    for i, ngrid in enumerate(grids):
        _, dx[i] = grid_pts(ngrid)
    h = dx / dx[-1]

    return h.astype(np.int)


def find_gci(d2, d1, p, Fs=3, r=2):
    """Grid Convergence Index using Roache (1994) method.

    Inputs:
        x2 (array) - grid of the coarse mesh
        x1 (array) - grid of the fine mesh
        f2 (array) - solution on coarse mesh
        f1 (array) - solution on fine meash
        p (float) - order of accuracy
        FS (default=3) - factor of safety
        r (default=2) - grid refinement factor

    Outputs:
        gci (float) - grid convergence index of fine grid
    """

    x2, f2 = d2[:, 0], d2[:, 1]
    x1, f1 = d1[:, 0], d1[:, 1]

    assert len(f1) > len(f2)
    assert len(x2) == len(f2)
    assert len(x1) == len(x1)

    # common grid points for both meshes
    f1_mask = np.in1d(x1, x2)

    c = np.float(Fs) / (r**p - 1)
```

```python
    gci = c * np.abs((f2 - f1[f1_mask]) / f1[f1_mask])

    return x2, gci
```

2 Perform a code validation study

Peform a code validation study using the steady-state manufactured soluation given by

$$T(x) = 300 + 200 \sin\left(\frac{3\pi x}{2L}\right) K,\ L = 1m$$

```python
def exact_p1(x, L=1):
    """Exact solution for verification."""

    return 300. + 200. * np.sin(3 * np.pi * x / (2 * L))


def source_p1(x, alpha, L=1):
    """Source term for manufactured solution."""

    return 450. * alpha * (np.pi / L) ** 2 * np.sin(3*np.pi*x / (2*L))


def solver_p1(T, f, c, dt, kmax):
    """Solver for 1D unsteady heat equation.

    Inputs:
        T (array) - temperature array at inital value
        f (array) - source term evaluated at grid points
        c (float) - alpha / dx**2
        dt (float) - time step
        kmax (int) - max number of iterations to perfom

    Returns:
        T (array) - numerical solution
        L (array) - L2 and Linf of the relative iterative residuals
    """

    assert len(T) == len(f)
    ngrid = len(f)
    imin, imax = 1, ngrid - 1
    Rtmp = 1.

    R = np.zeros(ngrid)
    L = np.zeros((kmax, 2))

    for k in range(kmax):

        # steady-state iterative residual
        for i in range(imin, imax):
            R[i] = c * (T[i-1] - 2*T[i] + T[i+1]) + f[i]
```

```
        Rtmp = norm(R)
        L[k] = Rtmp, norm(R, np.inf)

        # update the temperature value
        T = T + dt * R
        T[0], T[-1] = 300., 100.

        if Rtmp / L[0, 0] <= 1e-14:
            break

    L /= L[0]

    return T, L[:k]
```

```
kmax = 110000
x0, x1 = 0, 1
alpha = 9.71e-5
poss_grids = np.array([5, 9, 17, 33, 65, 129])

namet = './data/MMS_{0}_{1}.npy'
Tnames, Lnames = [], []
save_files = False
```

```
for ngrid in poss_grids:

    x, dx = grid_pts(ngrid)
    dt = .3 * dx**2 / alpha
    c = alpha / dx**2

    f = source_p1(x, alpha)
    T = 300. * np.ones(ngrid)
    T[0], T[-1] = exact_p1(x0), exact_p1(x1)

    start = time.time()
    T, L = solver_p1(T, f, c, dt, kmax)
    print('{0} grid pts: {1:.2f} sec'.format(ngrid, time.time() - start))

    if save_files:
        Tname = namet.format(ngrid, 'T')
        Lname = namet.format(ngrid, 'L')
        Tnames.append(Tname)
        Lnames.append(Lname)
        np.save(Tname, np.c_[x, exact(x), T])
        np.save(Lname, L)

pickle.dump([Tnames, Lnames], open('./data/MSM_names', 'wb'))
```

2.1 Make plots of solution and relative iterative residual

```
fig1, ax = plt.subplots(6, 1, sharex=True)
fig2, ax2 = plt.subplots()

Tnames, Lnames = pickle.load(open('./data/MSM_names', 'rb'))
markers = ['o', '^', 's', 'x', '*', 'v']
xx = np.linspace(0, 1, 1000)
ex = exact_p1(xx)
DE = np.zeros((len(Tnames), 2))

for i in range(len(Tnames)):

    Tf, Lf = Tnames[i], Lnames[i]
    grid = str(poss_grids[i])

    T, L = np.load(Tf), np.load(Lf)
    de = (T[:, 2] - T[:, 1]) / T[:, 1]
    DE[i] = norm(de, np.inf), norm(de, 2)

    ax[i].plot(T[:, 0], T[:, 2], color='k', ls='None', marker=markers[i], label='N = {0}'
        .format(grid))
    ax[i].plot(xx, ex, '-k', lw=1.5, alpha=.75)
    ax[i].set_yticks(np.arange(100, 600, 100))
    ax[i].set_xlim(-0.05, 1.05)
    ax[i].legend(loc='upper right', frameon=False, numpoints=1)

    ax2.loglog(L[:, 0], 'k', lw=2)
    ax2.loglog(L[:, 1], '--k', lw=2)
    #ax2.semilogy(L[:, 0], 'k', lw=2)
    #ax2.semilogy(L[:, 1], '--k', lw=2)

for axis in ax[:-1]:
    axis.set_yticks([])

ax[-1].set_xlabel('distance, m', fontsize='x-large')
ax[-1].set_ylabel('temp, K', fontsize='x-large')
ax[-1].tick_params(axis='x', which='major', labelsize=12)
fig1.subplots_adjust(bottom=.08, top=.99, hspace=0)

ax2.set_ylabel('relative residual')
ax2.set_xlabel('iteration, k')
leg = ax2.legend((r'$L_2$', r'$L_\infty$'), loc='best', fancybox=True)
leg.get_frame().set_alpha(0.5)

#ax2.annotate('0.25 on data', (1e3, 1e-13), textcoords='data', size=10)

#fig1.savefig('./report/figs/MMS_profiles.eps')
#fig2.savefig('./report/figs/MMS_resid.eps')
#fig2.savefig('./report/figs/MMS_resid.svg')
```

```
# find order of accuracy and normalized grid spacing
phat = find_phat(DE)
h = find_h(poss_grids)


fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

ax1.semilogx(h[1:], phat[:, 0], '-.ks', ms=8, label=r'$L_2$')
ax1.semilogx(h[1:], phat[:, 1], '--ko', ms=8, label=r'$L_\infty$')
ax1.grid(True, which='both')

ax1.set_ylim(0, 3)
ax1.set_ylabel(r'order of accuracy, $\hat{p}$', fontsize='x-large')
ax1.tick_params(axis='both', which='major', labelsize=14)

ax2.loglog(h, DE[:, 0], '-.ks', ms=8, label=r'$L_2$')
ax2.loglog(h, DE[:, 1], '--ko', ms=8, label=r'$L_\infty$')
ax2.grid(True, axis='x', which='both')
ax2.tick_params(axis='both', which='major', labelsize=14)

ax2.set_xlim(.75, 40)
ax2.set_xlabel('grid refinement factor, h', fontsize='x-large')
ax2.legend(loc='lower right', fancybox=True)
ax2.set_ylabel(r'norm of global DE', fontsize='x-large')

fig.tight_layout()

fig.savefig('./report/figs/MMS_ooa.eps')

#hh = np.linspace(.01, 40, 1000)
#offset = -1e-5
#ff = 1 * hh - offset
#ss = 2 * hh - offset
#ax.loglog(hh, ff)
#ax.loglog(hh, ss)
```

3 Part 2

Numerical solution for the heated bar with a source term.

```
def source_p2(x):
    """Source term for part 2."""

    return 100 * (.25 * (.75 - np.abs(x - 2./3))) ** 4



def solver_p2(T, f, c, dt, kmax, BC=1):
    """Solver for 1D unsteady heat equation.

    Inputs:
        T (array) - temperature array at inital value
```

```
        f (array) - source term evaluated at grid points
        c (float) - alpha / dx**2
        dt (float) - time step
        kmax (int) - max number of iterations to perfom

    Returns:
        T (array) - numerical solution
        L (array) - L2 and Linf of the relative iterative residuals
    """

    assert len(T) == len(f)
    ngrid = len(f)
    imin, imax = 1, ngrid - 1
    Rtmp = 1.
    Tbc = -200 # K/m

    R = np.zeros(ngrid)
    L = np.zeros((kmax, 2))

    for k in range(kmax):

        # steady-state iterative residual
        for i in range(imin, imax):
            R[i] = c * (T[i-1] - 2*T[i] + T[i+1]) + f[i]
        Rtmp = norm(R)
        L[k] = Rtmp, norm(R, np.inf)

        # update the temperature value
        T = T + dt * R

        # update the derivative boundary condition
        if BC == 1:
            T[-1] = T[-2] + dx * Tbc
        elif BC == 2:
            T[-1] = 1/3. * (4*T[-2] - T[-3] + 2*dx*Tbc)

        if Rtmp / L[0, 0] <= 1e-12:
            break

    L /= L[0]

    return T, L
```

```
kmax = 400000
x0, x1 = 0, 1
alpha = 9.71e-5
poss_grids = np.array([5, 9, 17, 33, 65, 129])
bcs = np.array([1, 2])

x, dx = grid_pts(ngrid)
dt = .3 * dx**2 / alpha
```

```
c = alpha / dx**2

namet = './data/p2_{0:03d}_{1}_BC{2}.npy'
save_files = False
```

```
for ngrid in poss_grids:
    for BC in bcs:
        x, dx = grid_pts(ngrid)
        dt = .3 * dx**2 / alpha
        c = alpha / dx**2

        f = source_p2(x)
        T = 300. * np.ones(ngrid)

        start = time.time()
        T, L = solver_p2(T, f, c, dt, kmax, BC=BC)
        print('{0} grid pts: {1:.2f} sec'.format(ngrid, time.time() - start))

        if save_files:
            Tname = namet.format(ngrid, 'T', BC)
            Lname = namet.format(ngrid, 'L', BC)
            np.save(Tname, np.c_[x, T])
            np.save(Lname, L)
```

```
129 grid pts: 126.91 sec
129 grid pts: 129.80 sec
```

```
BC1_T = sorted(glob('./data/*T_BC1*'))
BC2_T = sorted(glob('./data/*T_BC2*'))
BC1_L = sorted(glob('./data/*L_BC1*'))
BC2_L = sorted(glob('./data/*L_BC2*'))
nruns = len(BC1_T)

fig1, ax = plt.subplots(1, 2, sharex=True, sharey=True)
fig2, ax2 = plt.subplots()

markers = ['o', '^', 's', 'x', '*', 'v']
colors = ['b', 'g', 'r', 'k', 'y', 'm']
DE_1 = np.zeros((nruns, 2))
DE_2 = DE_1.copy()


for i in range(nruns):

    grid = poss_grids[i]
    t1, t2 = np.load(BC1_T[i]), np.load(BC2_T[i])
    l1, l2 = np.load(BC1_L[i]), np.load(BC2_L[i])
```

```
    ax[0].plot(t1[:, 0], t1[:, 1], ls='None', color=colors[i], ms=6, marker=markers[i],
        alpha=.75, label='{0} noes'.format(grid))
    ax[1].plot(t2[:, 0], t2[:, 1], ls='None', color=colors[i], ms=6, marker=markers[i],
        alpha=.75)

    if True:
        lw = 1.5
        ax2.loglog(l1[:, 0], 'g--', lw=lw)
        ax2.loglog(l2[:, 0], 'r--', lw=lw)
        ax2.loglog(l1[:, 1], 'g', lw=lw)
        ax2.loglog(l2[:, 1], 'r', lw=lw)

ax[0].set_title('First order accurate')
ax[1].set_title('Second order accurate')
ax[0].legend(loc='best', frameon=False)
ax[0].set_xlabel('distance, m')
ax[1].set_xlabel('distance, m')
ax[0].set_ylabel('temperature, K')
ax[0].set_xlim(-0.05, 1.05)
ax[0].tick_params(axis='both', which='major', labelsize=12)

ax2.set_ylabel('relative residual')
ax2.set_xlabel('iteration, k')
ax2.tick_params(axis='both', which='major', labelsize=12)
leg = ax2.legend((r'1st order $L_2$', r'2nd order $L_2$', r'1st order $L_\infty$', r'2nd
     order $L_\infty$'), loc='best', fancybox=True)
leg.get_frame().set_alpha(0.5)

fig1.subplots_adjust(hspace=0)
fig1.tight_layout()
#fig1.savefig('./report/figs/p2_profiles.eps')

fig2.tight_layout()
#fig2.savefig('./report/figs/p2-relresid.eps')
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)
markers = ['^', 'o', 's', '*', 'v']

for i in range(nruns - 1):

    grid_h2, grid_h1 = poss_grids[i], poss_grids[i+1]
    t1_h1, t2_h1 = np.load(BC1_T[i]), np.load(BC2_T[i])
    t1_h2, t2_h2 = np.load(BC1_T[i+1]), np.load(BC2_T[i+1])

    gcix1, gci1 = find_gci(t1_h1, t1_h2, 1)
    gcix2, gci2 = find_gci(t2_h1, t2_h2, 2)

    label = '{0} - {1} grid'.format(grid_h2, grid_h1)
    ax1.semilogy(gcix1, gci1, 'k', ls='None', marker=markers[i], label=label)
    ax2.semilogy(gcix2, gci2, 'k', ls='None', marker=markers[i])
```

```
ax1.legend(loc='lower right', frameon=False, numpoints=2)
ax1.set_xlabel('distance, m')
ax2.set_xlabel('distance, m')
ax1.set_xlim(-.05, 1.05)
ax2.set_xlim(-.05, 1.05)
ax1.set_ylabel('GCI')

fig.tight_layout()

#fig.savefig('./report/figs/p2_gci.eps')
```