Summary of Routines Written in Python

In [1]:

```python
import sys
from scipy import linalg as LA
import numpy as np
from matplotlib import pyplot as plt

sys.path.append('/Users/Lampe/PyScripts')
import blfunc as bl

np.set_printoptions(precision = 2, suppress = True)
```

Matprop

In [2]:

```python
def Matprop(mat_type = 0, elast_parm = 0, visco_parm = 0, elplast_parm = 0):
    """ Function used to load material parameters into "matpropv[0:99]".
    Elastic (orthotropic) parameters will always be calculated and returned.
    If mat_type not equal to zero, additional parameters (in addtional to elastic) will be calculated.

    INPUT
    mat_type: integer - defines how the material will be modeled
        0 = Elastic Material (default)
        1 = Viscoelastic Material
        3 = Elastoplastic Material
        All Other Values = Error - function will be terminated
    elast_parm: array of orthotropic elastic parameters (up to nine unique values)
        elast_parm[0] = Y1
        elast_parm[1] = Y2
        elast_parm[2] = Y3
        elast_parm[3] = nu12
        elast_parm[4] = nu23
        elast_parm[5] = nu31
        elast_parm[6] = G44
        elast_parm[7] = G55
        elast_parm[8] = G66
    visco_parm: not defined yet
    elplast_parm: not defined yet

    OUTPUT
        matpropv: vector[0:99] - stores material parameters (constants) for different constitutive models
        00 - 29: elasticity parameters
        30 - 39: viscoelasticity parameters
        40 - 59: plasticity parameters
        60 - 99: undefined
    """

    if mat_type == 0:
        print "Elastic Material"
    elif mat_type == 1:
        print "Viscoelastic Material"
    elif mat_type == 2:
        print "Elastoplastic Material"
    else:
        sys.exit("BAD VALUE: mat_type must be valued 0 (elastic), 1 (viscoelastic), or 2 (plastic)")


    ##########################################
    # user defined Engineering moduli / constitutive parameters
    ##########################################
    if any(elast_parm) == 0:
        # use default (debugging) values
        print "Zero value in Elastic Parameter array, default elastic parmeters used:"
        print "Y1, Y2, Y3 = 3, 1, 2"
        print "nu12, nu23, nu31 = 0.3, 0.35, 0.4"
        print "G1, G2, G3 = 1, 1, 1"

        Y1, Y2, Y3 = 3, 3, 3 # GPa
        nu12, nu23, nu31 = 0.35, 0.35, 0.35
        G1, G2, G3 = 4, 4, 4 # GPa
    else:
        # use user defined values
        print "user defined elastic parm"
        print elast_parm

        Y1, Y2, Y3 = elast_parm[0], elast_parm[1], elast_parm[2] # GPa
        nu12, nu23, nu31 = elast_parm[3], elast_parm[4], elast_parm[5],
        G1, G2, G3 = elast_parm[6], elast_parm[7], elast_parm[8],  # GPa

    if visco_parm == 0:
        # use default values
        print "defaut viscoelastic parm - current none"
    else:
        # use user defined values
        print "user defined viscoelastic parm"

    if elplast_parm == 0:
        # use default values
        print "default elastoplastic parm - current none"
    else:
        # use user defined values
        print "user defined elastoplastic parm"


    # add additional parameters below

    ############################################
    # determine components of the elastic matrix
    # elastic components are always calculated
    ############################################
    matpropv = np.zeros(100)
    em = bl.Elastic(Y1, Y2, Y3, nu12, nu23, nu31, G1, G2, G3 ) # the elasticicty matrix (6x6)
    matpropv[0:12] = [em[0,0], em[1,1], em[2,2], em[0,1], em[1,2], em[2,0],
                      em[1,0], em[2,1], em[0,2], em[3,3], em[4,4], em[5,5]]

    ############################################
    # determine path specific components for elastic component overrides
    ############################################
    # path specific elastic components (14-1 in notes)
    Eps_31 = em[2,0] / em[2,2] # Plane stress override
    Eps_32 = em[2,1] / em[2,2] # plane stress override
    matpropv[12:14] = [Eps_31, Eps_32]

    # uniaxial stress override calc - use Cramer's rule
    Dux = LA.det(np.array([[em[1,1], em[1,2]], [em[2,1], em[2,2]]]))
    Aux = LA.det(np.array([[em[1,0], em[1,2]], [em[2,0], em[2,2]]]))
    Bux = LA.det(np.array([[em[1,1], em[1,0]], [em[2,1], em[2,0]]]))

    Eux_21 = Aux / Dux # uniaxial stress override
    Eux_31 = Bux / Dux # uniaxial stress override
    matpropv[14:16] = [Eux_21, Eux_31]

    # hydrostatic stress override calc (14-2 in notes)
    a1 = em[0,1] - em[1,1]
    a2 = em[0,2] - em[1,2]
    a3 = em[0,0] - em[2,0]

    b1 = em[0,1] - em[2,1]
    b2 = em[0,2] - em[2,2]
    b3 = em[0,0] - em[2,0]

    Dh = LA.det(np.array([[a1, a2], [b1, b2]]))
    Ah = LA.det(np.array([[a3, a2], [b3, b2]]))
    Bh = LA.det(np.array([[a1, a3], [b1, b3]]))

    Eh_21 = Ah / Dh # hydrostatic stress override
    Eh_31 = Bh / Dh # hydrostatic stress override
    matpropv[16:18] = [Eh_21, Eh_31]

    # add additional material parameters below
    if mat_type == 1:
        # calculate viscoelastic parameters
        print "mat_type = 1, viscoelastic"

    if mat_type == 2:
        # calculate elastoplastic parameters
        print "mat_type = 2, elastoplastic"

#     print "matpropv:"
#     print  matpropv
    print em
    return matpropv
```

Aniso_Elast

In [3]:

```python
def Aniso_Elast(matpropv, estrn_incv, estrnv, strsv, term_type, p_max, p_min, path = 0,):
    """
    The constitutive equation for anisotropic (orthotropic) elasticity.
    Calculate's stress resulting from an elastic strain increment.
    path: integer (0 to 9) - indentifies the type of loading or strain path e.g. the test type
        0 = strain prescribed - the default
        1 = uniaxial stress
        2 = plane stress
        3 = hydrostatic stress
        4 = equal biaxial stress (triaxial test)
```

```
        5 - 9: undefined, may be defined later

    matpropv: vector (0 to 99) - stores material parameters (constants) for different constitutive models
        00 - 29: elasticity parameters
        30 - 39: viscoelasticity parameters
        40 - 59: plasticity parameters
        60 - 99: undefined

    estrn_incv: vector (0 to 5) - stores elastic strain increments
        Note:  sqrt(2) has been omitted on shear components - cannot transform to another basis with this vector
        0 = delta e11 elastic
        1 = delta e22 elastic
        2 = delta e33 elastic
        3 = delta e12 elastic
        4 = delta e23 elastic - currently empty
        5 = delta e31 elastic - currently empty

    estrnv: vector (0 to 5) - stores elastic strain values
        0 = e11 elastic
        1 = e22 elastic
        2 = e33 elastic
        3 = e12 elastic
        4 = e23 elastic - currently empty
        5 = e31 elastic - currently empty

    strsv:  vector (0 to 5) - stores total stress measure
        Note:  sqrt(2) has been omitted on shear components - cannot transform to another basis with this vector
        0 = sigma11
        1 = sigma22
        2 = sigma33
        3 = sigma12
        4 = sigma23
        5 = sigma31
    """

    # elastic matrix components
    E_11 = matpropv[0]
    E_22 = matpropv[1]
    E_33 = matpropv[2]
    E_12 = matpropv[3]
    E_23 = matpropv[4]
    E_31 = matpropv[5]
    E_21 = matpropv[6]
    E_32 = matpropv[7]
    E_13 = matpropv[8]
    E_44 = matpropv[9]
    E_55 = matpropv[10]
    E_66 = matpropv[11]

    # elastic path specific overrides
    Eps_31 = matpropv[12]
    Eps_32 = matpropv[13]
    Eux_21 = matpropv[14]
    Eux_31 = matpropv[15]
    Eh_21  = matpropv[16]
    Eh_31  = matpropv[17]
    Etxe_32  = matpropv[18]
    # Etx_b  = matpropv[19]

    # define componet specific elastic strian increments
    estrn_inc_11 = estrn_incv[0]
    estrn_inc_22 = estrn_incv[1]
    estrn_inc_33 = estrn_incv[2]
    estrn_inc_12 = estrn_incv[3]
    estrn_inc_23 = estrn_incv[4] # currently undefined
    estrn_inc_31 = estrn_incv[5] # currently undefined

    if path == 0:
        # strain prescribed path - no overrides made
        override = 0
#         print "no overrides"
    elif path == 1:
        # uniaxial stress
        print "uniaxial stress override"
        estrn_inc_22 = -Eux_21 * estrn_inc_11
        estrn_inc_33 = -Eux_31 * estrn_inc_11
    elif path == 2:
        # plane stress
        print "plane stress override"
        estrn_inc_33 = -(Eps_31 * estrn_inc_11) - (Eps_32 * estrn_inc_22)
    elif path ==3:
        # hydrostatic stress
        print "hydrostatic stress override"
        estrn_inc_22 = -Eh_21 * estrn_inc_11
        estrn_inc_33 = -Eh_31 * estrn_inc_11
    elif path == 4:
        # biaxial stress (Trx Extension)
        # sigma 1 > sigma 2 = sigma 3,
        print "Triaxial Extension Stress Override"
               # biaxial stress (Trx Compression)
        # sigma 1 = sigma 2 > sigma 3
        if term_type[0] == 7 and term_type[1] == 2:
            p0 = p_min
            D_trxc = LA.det(np.array([[E_22, E_23],
                                      [E_32, E_33]]))
            rhs_22 = p0 - strsv[1] - E_21 * estrn_inc_11
            rhs_33 = p0 - strsv[2] - E_31 * estrn_inc_11
            A_trxc = LA.det(np.array([[rhs_22, E_23],
                                     [rhs_33, E_33]]))
            B_trxc = LA.det(np.array([[E_22, rhs_22],
                                      [E_32, rhs_33]]))
            estrn_inc_22 = A_trxc / D_trxc
            estrn_inc_33 = B_trxc / D_trxc
        else:
            print """For Triaxial Extension:
            First step -  assumed hydrostatic confining pressure is applied.
             Second Step - sigma_11 will be increased (requires a type 2 termination (sigma_11 max).
                          sigma_22 and sigma_33 will be maintained at p_min (pressure minimum during
                          hydrostatic step."""
            sys.exit("Must redfined step sequence or termination limits")
    elif path == 5:
        # biaxial stress (Trx Compression)
        # sigma 1 = sigma 2 > sigma 3
        print "Triaxial Compression Stress Override"
        if term_type[0] == 7 and term_type[1] == 3:
            p0 = p_min
            D_trxc = LA.det(np.array([[E_22, E_23],
                                      [E_32, E_33]]))
            rhs_22 = p0 - strsv[1] - E_21 * estrn_inc_11
            rhs_33 = p0 - strsv[2] - E_31 * estrn_inc_11
            A_trxc = LA.det(np.array([[rhs_22, E_23],
                                     [rhs_33, E_33]]))
            B_trxc = LA.det(np.array([[E_22, rhs_22],
                                      [E_32, rhs_33]]))
            estrn_inc_22 = A_trxc / D_trxc
            estrn_inc_33 = B_trxc / D_trxc
        else:
            print """For Triaxial Compression
            First step -  assumed hydrostatic confining pressure is applied.
            Second Step - sigma_11 will be increased (requires a type 3 termination (sigma_11 min).
                          sigma_22 and sigma_33 will be maintained at p_min (pressure minimum during
                          hydrostatic step."""
            sys.exit("Must redfined step sequence or termination limits")

    else:
        # may add additional paths later
        print "Invalid Path Specification in 'Aniso_Elast'"
        sys.exit('Bad Path Type: exit routine')

    # strain increments have been modified as necessary, now calculate stress increments
    strs_inc_11 = E_11 * estrn_inc_11 + E_12 * estrn_inc_22 + E_13 * estrn_inc_33
    strs_inc_22 = E_21 * estrn_inc_11 + E_22 * estrn_inc_22 + E_23 * estrn_inc_33
    strs_inc_33 = E_31 * estrn_inc_11 + E_32 * estrn_inc_22 + E_33 * estrn_inc_33
    strs_inc_12 = E_44 * estrn_inc_12
    strs_inc_23 = E_55 * estrn_inc_23
    strs_inc_31 = E_66 * estrn_inc_31

    # update total strain vector
    estrnv[0] = np.around(estrnv[0] + estrn_inc_11, 10)
    estrnv[1] = np.around(estrnv[1] + estrn_inc_22, 10)
    estrnv[2] = np.around(estrnv[2] + estrn_inc_33, 10)
    estrnv[3] = np.around(estrnv[3] + estrn_inc_12, 10)
    estrnv[4] = np.around(estrnv[4] + estrn_inc_23, 10)
    estrnv[5] = np.around(estrnv[5] + estrn_inc_31, 10)

    # update total stress vector
    strsv[0] = np.around(strsv[0] + strs_inc_11, 10)
    strsv[1] = np.around(strsv[1] + strs_inc_22, 10)
    strsv[2] = np.around(strsv[2] + strs_inc_33, 10)
    strsv[3] = np.around(strsv[3] + strs_inc_12, 10)
    strsv[4] = np.around(strsv[4] + strs_inc_23, 10)
    strsv[5] = np.around(strsv[5] + strs_inc_31, 10)

    return estrnv, strsv
```

In [4]:
```python
def Constit_Eq(mat_type, leg, matpropv, strn_incm, strnv, estrnv, bstrnv, pstrnv, strsv, path = 0):
    """
    Provides a general structure to be used for any constitutive equation.
    Initially assumes the plastic and elastic strain increments are equal.
    This assumption is then checked, and if not true an adjustment is made.
    Each of the constitutive equation S\E will update their respective strain type,
    and the sum of all specific strain types will be the total strain.

    INPUT
        path: integer (0 to 9) - indentifies the type of loading or strain path e.g. the test type
            0 = strain prescribed - the default
            1 = uniaxial stress
            2 = plane stress
            3 = hydrostatic stress
            4 = equal biaxial stress (triaxial test)
            5 - 9: undefined, may be defined later
        leg: defines which leg, increment size can vary with leg
        mat_type: integer - defines how the material will be modeled
            0 = Elastic Material (default)
            1 = Viscoelastic Material
            3 = Elastoplastic Material
            All Other Values = Error - function will be terminated
        matpropv: vector (0 to 99) - stores material parameters (constants) for different constitutive models
            00 - 29: elasticity parameters
            30 - 39: viscoelasticity parameters
            40 - 59: plasticity parameters
            60 - 99: undefined

    OUTPUT
        strnv:  vector (0 to 5) containing the total strain
        estrnv: vector (0 to 5) containing the total elastic strain
        bstrnv: vector (0 to 5) containing the total back strain
        pstrnv: vector (0 to 5) containing the total plastic strain
    """

    # assign all increments to be elastic increments, then calculate total stress and total strain
    estrn_incv = strn_incm[:, leg]

    # now calls the anisotropic (orthotropic) elasticity constitutive equation
    estrnv, strsv = Aniso_Elast(matpropv, estrn_incv, estrnv, strsv, path)

    # make adjustments for viscous or plastic materials
    if mat_type == 0:
        # assumes all strains and elastic and no adjustments are made
        mat_string = "Elastic"
#         print "fully elastic material"
    elif mat_type == 1:
        # Call viscoelastic constitutive equation
        print "calls Visc"
    elif mat_type == 2:
        # Call elastoplastic constitutive equation
        print "calls elast plast"

    # sum up strains
    for i in xrange(6):
        strnv[i] = estrnv[i] + bstrnv[i] + pstrnv[i]

    return strnv, estrnv, bstrnv, pstrnv, strsv
```

Printing Functions

In [5]:
```python
##################################
# printing functions
##################################
def valprint(string, value):
    """ Inforces uniform formatting of scalar value outputs."""
    print("{0:>15}: {1: .2e}".format(string, value))

def matprint(string, value):
    """ inforces uniform formatting of matrix value outputs."""
    print("{0}:".format(string))
    print(value)
```

Term2 and Limit_Delta

In [6]:
```python
def Term2(irow, limit_deltav, j):
    """
    Determines if the current leg should be terminated.  Termination occurs by ???
    :param irow: index for limit_deltav
    :param limit_deltav: vector of calculated difference between limiting and current values
    :param j: with loop index
    :return:
    """
    cont = 1 # default is to continue
    if j != 1:
        rate_delta = limit_deltav[irow] - limit_deltav[irow - 1]
        if rate_delta > 0:
            # value is not approaching limit
            # stop loop and go to next leg
            cont = 0
        elif rate_delta < 0:
            # value is diverging from limit
            # continue looping
            cont = 1
        elif rate_delta == 0:
            # value parallel to limit
            # currently continue looping also
            cont = 1
    return cont


def Limit_Delta(irow, strnv, strsv, p, term_limv, term_type, leg, limit_deltav):
    """
    Calculates the difference between the termination value (limit) and
    the current value
    """

    if term_type[int(leg)] != 1:
        if term_type[int(leg)] == 2:
            val = strsv[0]
        if term_type[int(leg)] == 3:
            val = strsv[0]
        if term_type[int(leg)] == 4:
            val = strnv[0]
        if term_type[int(leg)] == 5:
            val = strnv[0]
        if term_type[int(leg)] == 6:
            val = p
        if term_type[int(leg)] == 7:
            val = p
    else:
        val = irow

    limit = term_limv[int(term_type[leg] - 1)]
    limit_deltav[irow] = np.abs(limit - val)
    return limit_deltav
```

Storage

In [7]:
```python
def Storage(irow, SM, time_step, strsv, strnv, estrnv, bstrnv = 0, pstrnv = 0, histv = 0):
    """
    Stores values of stress and strain at each time_step.
    All columns in a single row of data will be calculated at a single step,
    and the steps will progress moving down the rows.

    irow: identifies which row of SM to place data on
    SM : storage matrix - details below
    time-step: either the time or step
    strnv: total strain vector
    estrnv: elastic strain vector
    bstrnv: back strainvector
    pstrnv: plastic strain vector
    strsv: stress vector
    histv: unsure

    SM: storage matrix - column identifiers
        00 = irow

        -Stress
        01 = strs_11
        02 = strs_22
        03 = strs_33
        04 = strs_12
        05 = strs_23
        06 = strs_31

        -Various stress measures:
        07 = P1 - max principal stress
```

```
    08 = P2 - intermediate principal stress
    09 = P3 - min pricipal stress
    10 = p  - tensile pressure
    11 = q1
    12 = q2
    13 = q  - mises stress
    14 = I1 - first invariant of the cauchy stress tensor
    15 = J2 - second invariant of deviatoric stress tensor
    16 = J3 - third invariant of the deviatoric stress tensor
    17 = r  - radial Lode coordinate
    18 = z  - axial Load coordinate
    19 = theta - Lode angle
    20 = triax - triaxiality

    21 undef
    22 undef

    -Strain Measures
    23 = strn_11
    24 = strn_22
    25 = strn_33
    26 = strn_12
    27 = strn_23
    28 = strn_31

    29 = strn_vol
    30 = estrn_11
    31 = estrn_22
    32 = estrn_33
    33 = estrn_12
    34 = estrn_23
    35 = estrn_31

    36 = estrn_vol

    37 = bstrn_11
    38 = bstrn_22
    39 = bstrn_33
    40 = bstrn_12
    41 = bstrn_23
    42 = bstrn_31

    43 = pstrn_11
    44 = pstrn_22
    45 = pstrn_33
    46 = pstrn_12
    47 = pstrn_23
    48 = pstrn_31

    49 undef
    50 undef
"""
#define vector of column names for plotting
col_namev = ["Increment",
            "Stress_11", "Stress_22", "Stress_33", "Stress_12", "Stress_23", "Stress_31",
            "Sigma_1", "Sigma_2", "Sigma_3", "Mean Stress (p)", "q1", "q2", "Mises Stress (q)", "I1", "J2", "J3",
            "Lode r", "Lode z", "Lode Angle", "Triaxiality",
            "Undefined", "Undefined",
            "Strain_11", "Strain_22", "Strain_33", "Strain_12", "Strain_23", "Strain_31",
            "Strain_vol",
            "Elastic Strain_11", "Elastic Strain_22", "Elastic Strain_33",
            "Elastic Strain_12", "Elastic Strain_23", "Elastic Strain_31",
            "Plastic Strain_11", "Plastic Strain_22", "Plastic Strain_33",
            "Plastic Strain_12", "Plastic Strain_23", "Plastic Strain_31",
            "Back Strain_11", "Back Strain_22", "Back Strain_33",
            "Back Strain_12", "Back Strain_23", "Back Strain_31",
            "Undefined", "Undefined"]

##################################
# printing functions
##################################
def valprint(string, value):
    """ Inforces uniform formatting of scalar value outputs."""
    print("{0:>15}: {1: .2e}".format(string, value))

def matprint(string, value):
    """ inforces uniform formatting of matrix value outputs."""
    print("{0}:".format(string))
    print(value)

# if irow < 0:
#     return sys.exit("irow poorly defined in Storage")

#load all stresses into a matrix

##################################
# stress measure calculations
##################################
sigma = np.array([[strsv[0], strsv[3], strsv[5]],
                  [strsv[3], strsv[1], strsv[4]],
                  [strsv[5], strsv[4], strsv[2]]])
norm = LA.norm(sigma, 'fro') #L-2 norm

#if the norm is greater than machine accuracy: compute values
#else, the sigma matrix is assumed to be of zero order and all values will
#be assumed to equal zero
if norm > 10 * np.finfo(float).eps:
    sigma_sp = 1.0/3.0 * np.trace(sigma) * np.eye(3) #spherical (isotropic) stress matrix
    sigma_dv = sigma - sigma_sp #deviatoric stress matrix

    #calculate principal stresses
    eigvals = list(LA.eigvalsh(sigma))
    eigvals.sort()
    eigvals.reverse()
    P1 = eigvals[0] #maximum principal stress
    P2 = eigvals[1] #Int principal stress
    P3 = eigvals[2] #min principal stress

    #calculate the principal deviatoric stresses
    eigvals_dv = list(LA.eigvalsh(sigma_dv))
    eigvals_dv.sort()
    eigvals_dv.reverse()

    #calculate the max shear stress (sigma_1 - sigma_3)
    max_shear_stress = (max(eigvals) - min(eigvals)) / 2.0

    #calculate stress invariants
    I1 = np.trace(sigma)
    J2 = 1.0 / 2.0 * np.trace(np.dot(sigma_dv, sigma_dv))
    J3 = 1.0 / 3.0 * np.trace(np.dot(sigma_dv, np.dot(sigma_dv, sigma_dv)))

    #calculate other stress measures
    p = 1.0 / 3.0 * I1 #tensile pressure or mean stress
    q1 = np.sqrt(3.0) / 2 * (eigvals[0] - eigvals[1])
    q2 = -3.0 / 2.0 * (eigvals_dv[0] + eigvals_dv[1])

    # check that terms are equivalent
    mises_strs = np.sqrt(3.0 * J2)
    q = np.sqrt(q1**2 + q2**2)
    sqrt_J2 = np.sqrt(3.0 / 2.0 * (eigvals_dv[0]**2 + eigvals_dv[1]**2 + eigvals_dv[2]**2))

    #Lode angle calcs
    lode_r = np.sqrt(2.0 * J2)
    lode_z = I1 / np.sqrt(3.0) #coordinate is parallel to hydrostatic axis

    # c = 3.0 * np.sqrt(6.0) * LA.det(sigma_dv / lode_r)
    c = 3.0 * np.sqrt(6.0) * J3 / (np.sqrt(J2))
    lode_theta = 1.0 / 3.0 * np.arcsin(c)

    #stress triaxiality
    triax = p / mises_strs
else:
    #print calculated value of norm
    valprint("Row of SM", irow)
    valprint("Norm of Stress Matrix:", norm)
    valprint("Machine Accuracy", np.finfo(float).eps)
    print "Calculated norm of stress matrix is less than machine accuracy"
    print "All calculated values of stress are assumed to equal zero"
    P1 = 0
    P2 = 0
    P3 = 0
    p = 0
    q1 = 0
    q2 = 0
    q = 0
    I1 = 0
    J2 = 0
    J3 = 0
    lode_r = 0
    lode_z = 0
    lode_theta = 0
    triax = 0


##################################
# print check
##################################
# matprint("Input Stress", sigma)
# matprint("Spherical Stress", sigma_sp)
# matprint("Deviatoric Stress", sigma_dv)
# valprint("P1", eigvals[0])
# valprint("P2", eigvals[1])
```

```
    # valprint("P3", eigvals[2])
    # valprint("Tensile Pressure", p)
    # valprint("q1", q1)
    # valprint("q2", q2)
    # valprint("Mises Stress = q", q)
    # valprint("I1", I1)
    # valprint("J2", J2)
    # valprint("J3", J3)
    # print "Lode Coord."
    # valprint("Lode r", lode_r)
    # valprint("Lode z", lode_z)
    # valprint("Lode Angle (rad)", lode_theta)
    # valprint("Triaxiality", triax)

    ##################################
    # strain calc
    ##################################

    # need to define volumetric strain
    # should use log strains6

    ##################################
    # allocate values to SM
    ##################################
    SM[irow, 0]     = irow
    SM[irow, 1:7]   = strsv[0], strsv[1], strsv[2], strsv[3], strsv[4], strsv[5]
    SM[irow, 7:21]  = P1, P2, P3, p, q1, q2, q, I1, J2, J3, lode_r, lode_z, lode_theta, triax
    SM[irow, 21:23] = -999, -999 #undefined
    SM[irow, 23:29] = strnv[0], strnv[1], strnv[2], strnv[3], strnv[4], strnv[5]
    SM[irow, 29:30] = -999 #volumetric strain
    SM[irow, 30:36] = estrnv[0], estrnv[1], estrnv[2], estrnv[3], estrnv[4], estrnv[5]
    SM[irow, 36:42] = bstrnv[0], bstrnv[1], bstrnv[2], bstrnv[3], bstrnv[4], bstrnv[5]
    SM[irow, 42:48] = pstrnv[0], pstrnv[1], pstrnv[2], pstrnv[3], pstrnv[4], pstrnv[5]
    SM[irow, 48:50] = -999, -999 # undefined
    return SM, col_namev
```

Plot_Setup

In [8]:
```
def Plot_Setup(SM, col_namev, out_dir, out_name, irow,
               sub_plot = 1, path = 0, x1 = 0, x2 = 0, y1 = 1, y2 = 23, fmt = "pdf"):
    """ Produces Plots of data and saves to output path.  Default output format is .pdf
    SM: storage matrix
    col_namev: vector of column names from SM
    sub_plot: plotting option
        0 = plot with single variable(y1(x))
        1 = plot with two variables (y1(x) and y2(x)) (default)
    path_name: name of modeled path
        0 = strain prescribed - the default
        1 = uniaxial stress
        2 = plane stress
        3 = hydrostatic stress
        4 = equal biaxial stress (triaxial test)
        5 - 9: undefined, may be defined later
    out_dir: output directory where plot will be saved
    out_name: name of output file
    x1: plots along the horizontal axis
    y1: plots along vertical axis against x1
    y2: plots along vertical axis also against x1
    fmt: identifies the file type (pdf, png, jpeg, etc.)
    """
    t_delta = 2

    path_name = ['Strain Prescribed', 'Uniaxial Stress', 'Plane Stress',
                 'Hydrostatic Stress', 'Triaxial Extension', 'Triaxial Compression', 'Undefined']

    time = SM[:,0] * t_delta

    plt.close('all')
    fig = plt.figure()

    if sub_plot == 0:
        # single plot
        dat_x1 = SM[0:irow, x1]
        dat_y1 = SM[0:irow, y1]
        ax_x1 = col_namev[x1]
        ax_y1 = col_namev[y1]

        ax1 = fig.add_subplot(2, 1, 1)
        ax1.plot(dat_x1, dat_y1, 'bo-')
        ax1.set_ylabel(ax_y1)
        ax1.grid(True)

        ax1.set_title(path_name[path])
        plt.show()

    elif sub_plot == 1:
        # two plots
        dat_x1 = SM[0:irow, x1]
        dat_y1 = SM[0:irow, y1]
        dat_x2 = SM[0:irow, x2]
        dat_y2 = SM[0:irow, y2]

        ax_x1 = col_namev[x1]
        ax_y1 = col_namev[y1]
        ax_x2 = col_namev[x2]
        ax_y2 = col_namev[y2]

        ax1 = fig.add_subplot(2, 1, 1)
        ax1.plot(dat_x1, dat_y1, 'bo-')
        ax1.set_ylabel(ax_y1)
        ax1.grid(True)
        ax1.set_title(path_name[int(path)])

        # if x1 and x2 are the same, the axis title will be shared
        if x1 == x2:
            ax2 = fig.add_subplot(2, 1, 2, sharex = ax1)
        else:
            ax2 = fig.add_subplot(2, 1, 2)
            fig.tight_layout()
            ax1.set_xlabel(ax_x1)

        ax2.plot(dat_x2, dat_y2, 'yo-')
        ax2.set_ylabel(ax_y2)
        ax2.grid(True)
        ax2.set_xlabel(ax_x2)

    # save figure to out_dir
    fname = out_dir + "/" + out_name + "." + fmt

    fig.savefig(fname, dpi=None, facecolor='w', edgecolor='w',
                orientation='Landscape', papertype=None, format=None,
                transparent=True, bbox_inches='tight', pad_inches=0.1,
                frameon=None)

    # plt.show()
    return "Plotting Complete"
```

Driver

In [9]:
```
def Driver(run_title, n_leg, path_type, term_type,
           Y1, Y2, Y3, nu12, nu23, nu31, G44, G55, G66,
           strn_11, strn_22 = 0, strn_33 = 0, strn_12 = 0, mat_type = 0, t_inc = 0,
           n_max = 100,
           inc_max = 50, strs_max = 1, strs_min = -1, strn_max = 1, strn_min = -1, p_max = 0.5, p_min = -0.5):

    """
    Driver Program for Constitutive Modeling

    path: integer (0 to 9) - indentifies the type of loading or strain path e.g. the test type
        0 = strain prescribed - the default
        1 = uniaxial stress
        2 = plane stress
        3 = hydrostatic stress
        4 = Triaxial Extension
        5 = Triaxial Compression
        6 - 9: undefined, may be defined later
    nleg: number of legs for the load path that is being modeled
    term_type: defines how each of the nleg will be terminated
        1 = n_max, maximium number of steps
        2 = strs_max, maximium stress value
        3 = strs_min, minimium stress value
        4 = strn_max, maximium strain value
        5 = strn_min, minimium strain value
        6 = p_max, maximium pressure (mean stress)
        7 = p_min, minimium pressure (mean stress)

    SM: storage matrix
    """

    #### create empty (zero) arrays ####
    # test loading path
    path = np.ones(n_leg) * path_type

    # material parameters used in the constitutive quations
    elastic_parm = np.zeros(9)
    visco_parm = np.zeros(10)
    elplast_parm = np.zeros(10)
    strn_incm = np.zeros((6,  n_leg))
```

```
#initialize stress vector
strsv = np.zeros(6)

# initialize strain vectors
estrnv = np.zeros(6)
bstrnv = np.zeros(6)
pstrnv = np.zeros(6)
strnv = np.zeros(6)

#initialize storage matrix (SM)
SM = np.zeros((n_max, 50))

#### define values for empty (zero) arrays ####
#define elastic parameters - default values give a nice Elasticity matrix
elastic_parm[0], elastic_parm[1], elastic_parm[2] = Y1, Y2, Y3 # youngs
elastic_parm[3], elastic_parm[4], elastic_parm[5] = nu12, nu23, nu31 # poissons
elastic_parm[6], elastic_parm[7], elastic_parm[8] = G44, G55, G66 # shear
#define viscoelastic parameters
#define elastoplastic parameters

#define total strain increments
#check for proper entry of strain increments
# def inc_check():

# KEEP THE MAGNITUDE OF STRAIN INCREMENTS CONSISTENt BETWEEN LEGS
strn_incm[0,0:n_leg] = np.ones(n_leg) * strn_11 # strn_inc_11
strn_incm[1,0:n_leg] = np.ones(n_leg) * strn_22 # strn_inc_22
strn_incm[2,0:n_leg] = np.ones(n_leg) * strn_33 # strn_inc_33
strn_incm[3,0:n_leg] = np.ones(n_leg) * strn_12 # strn_inc_12
strn_incm[4,0:n_leg] = np.ones(n_leg) * strn_12 # strn_inc_23
strn_incm[5,0:n_leg] = np.ones(n_leg) * strn_12 # strn_inc_31

############################
# develop termination criteria
###########################
term_limv = [inc_max, strs_max, strs_min, strn_max, strn_min, p_max, p_min]

#define matrix to store differences between limiting and measured values
limit_deltav= np.zeros(n_max)
limit_deltav[0] = np.abs(term_limv[int(term_type[0] - 1)]) #initial delta is the total difference

matpropv = Matprop(mat_type, elastic_parm) # vector of material properties

irow = 1
j = 0
for i in range(n_leg):
    leg = i
    if j > n_max:
        print "n_max reached - global break"
        break
    j = 1 # counter for while loop
    while j <= inc_max:
        # call the consitutive equations
        strnv, estrnv, bstrnv, pstrnv, strsv = Constit_Eq(mat_type, leg, matpropv, strn_incm, strnv,
                                                          estrnv, bstrnv, pstrnv, strsv,
                                                          term_type, p_max, p_min, path = int(path[leg]))
        p = 1.0 / 3.0 * (strsv[0] + strsv[1] + strsv[2])

        if term_type[int(leg)] == 1:
            # continuation will be handled by "while loop"
            cont = 1
        else:
            # specify condition to continue
            limit_deltav = Limit_Delta(irow, strnv, strsv, p, term_limv, term_type, leg, limit_deltav)
            cont = Term2(irow, limit_deltav, j)

        if cont == 1:
            if mat_type != 1:
                time_step = (irow - 1) * t_inc
            SM, col_namev = Storage(irow, SM, time_step, strsv, strnv, estrnv, bstrnv, pstrnv)
            j = j + 1
            irow = irow + 1
            if j > n_max:
                print "n_max reached - global break"
                break
            continue
        else:
            break

print run_title
print "Run Completed"
valprint("Number of Legs", n_leg)
valprint("Number of Calculations", irow)

return SM, col_namev, irow
```

Call Driver and Plot_Setup

In [10]:

```
import sys
import numpy as np
from matplotlib import pyplot as plt

sys.path.append('/Users/Lampe/PyScripts')
import constit_mod

np.set_printoptions(precision = 2, suppress = True)

"""
Driver Program for Constitutive Modeling

path: integer (0 to 9) - indentifies the type of loading or strain path e.g. the test type
    0 = strain prescribed - the default
    1 = uniaxial stress
    2 = plane stress
    3 = hydrostatic stress
    4 = Triaxial Extension
    5 = Triaxial Compression
    6 - 9: undefined, may be defined later
nleg: number of legs for the load path that is being modeled
term_type: defines how each of the nleg will be terminated
    1 = n_max, maximium number of steps
    2 = strs_max, maximium stress value
    3 = strs_min, minimium stress value
    4 = strn_max, maximium strain value
    5 = strn_min, minimium strain value
    6 = p_max, maximium pressure (mean stress)
    7 = p_min, minimium pressure (mean stress)

SM: storage matrix
"""
run_title = "Prob3_Txce3"

#### Model Parameters ###
n_leg = 2 # number of legs / changes in strain increment
path_type = [3, 4] # Loading path default is zero
t_inc = 1 # for viscoelastic analysis

#user defined strain increments
#define an increment value for each leg if needed

strn_11 = [-0.001, 0.005]
strn_22 = [-0.005, 0]
strn_33 = [0.0]
strn_12 = [0.0]

# termination type
term_type = np.ones(n_leg) * [7, 2]

#define termination values
n_max = 100 #global termination on number of loops

# local terminations dependent on term_type
inc_max = 50 # per leg
strs_max = -0.15
strs_min = -1.0
strn_max = 1
strn_min = -1
p_max = 0.5
p_min = -0.5

# mat_type: integer - defines how the material will be modeled
#     0 = Elastic Material (default)
#     1 = Viscoelastic Material
#     3 = Elastoplastic Material
mat_type = 0

### Material Parameters ###
Y1, Y2, Y3 = 28.0/3, 28.0/3, 28.0/3 # youngs
nu12, nu23, nu31 = 1.0/6, 1.0/6, 1.0/6 # poissons
G44, G55, G66 = 0.5, 0.5, 0.5 # shear

# call driver
SM, col_namev, irow = constit_mod.Driver(run_title, n_leg, path_type, term_type,
           Y1, Y2, Y3, nu12, nu23, nu31, G44, G55, G66,
           strn_11 = strn_11, strn_22 = strn_22, strn_33 = strn_33, strn_12 = strn_12, mat_type = mat_type,
           t_inc = t_inc,
           n_max = n_max,
           inc_max = inc_max, strs_max = strs_max, strs_min = strs_min, strn_max = strn_max, strn_min = strn_min,
           p_max = p_max, p_min = p_min)
```

```
out_dir = "/Users/Lampe/Documents/UNM_Courses/ME-562_ConstitutiveModelingAndAssociatedAlogorithms_SCHREYER/HW03"
out_fig_name = run_title
out_fig_fmt = "pdf"

x1 = 23
y1 = 24

x2 = 23
y2 = 25

# call plotting device
constit_mod.Plot_Setup(SM, col_namev, out_dir, out_name = out_fig_name, irow = irow,
                       sub_plot = 1,path = path_type[len(path_type) -1],
                       x1 = x1, x2 = x2, y1 = y1, y2 = y2, fmt = out_fig_fmt)
```

```
Elastic Material
user defined elastic parm
[ 9.33  9.33  9.33  0.17  0.17  0.17  0.5   0.5   0.5 ]
defaut viscoelastic parm - current none
default elastoplastic parm - current none
[[ 10.   2.   2.   0.   0.  -0.]
 [  2.  10.   2.   0.   0.  -0.]
 [  2.   2.  10.   0.   0.  -0.]
 [  0.   0.   0.   1.   0.  -0.]
 [  0.   0.   0.   0.   1.  -0.]
 [  0.   0.   0.   0.   0.   1.]]
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
hydrostatic stress override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Triaxial Extension Stress Override
Prob3_Txce3
Run Completed
 Number of Legs:  2.00e+00
Number of Calculations:  4.50e+01

constit_mod.py:640: RuntimeWarning: invalid value encountered in double_scalars
  c = 3.0 * np.sqrt(6.0) * J3 / (np.sqrt(J2))
constit_mod.py:644: RuntimeWarning: divide by zero encountered in double_scalars
  triax = p / mises_strs
```

```
Out[10]: 'SM: storage matrix - column identifiers\n        00 = irow\n\n        -Stress\n        01 = strs_11\n        02 = strs_22\n
03 = strs_33\n        04 = strs_12\n        05 = strs_23\n        06 = strs_31\n\n        -Various stress measures:\n        0
= P1 - max principal stress\n        08 = P2 - intermediate principal stress\n        09 = P3 - min pricipal stress\n        1
= p  - tensile pressure\n        11 = q1\n        12 = q2\n        13 = q  - mises stress\n        14 = I1 - first invariant o
the cauchy stress tensor\n        15 = J2 - second invariant of deviatoric stress tensor\n        16 = J3 - third invariant of
the deviatoric stress tensor\n        17 = r  - radial Lode coordinate\n        18 = z  - axial Load coordinate\n        19 =
heta - Lode angle\n        20 = triax - triaxiality\n\n        21 undef\n        22 undef\n\n        -Strain Measures\n
23 = strn_11\n        24 = strn_22\n        25 = strn_33\n        26 = strn_12\n        27 = strn_23\n        28 = strn_31\n\n
29 = strn_vol\n        30 = estrn_11\n        31 = estrn_22\n        32 = estrn_33\n        33 = estrn_12\n        34 = estrn_
3\n        35 = estrn_31\n\n        36 = estrn_vol\n\n        37 = bstrn_11\n        38 = bstrn_22\n        39 = bstrn_33\n
40 = bstrn_12\n        41 = bstrn_23\n        42 = bstrn_31\n\n        43 = pstrn_11\n        44 = pstrn_22\n        45 = pstr
_33\n        46 = pstrn_12\n        47 = pstrn_23\n        48 = pstrn_31\n\n        49 undef\n        50 undef\n        '

In [ ]:
```