Modeling and simulations of the Homogenized Material

First, some preliminary data elastic values are imported from mathilde.mat file (external)

The C_values_mathilde must be in your directory in which this notebook file is working, in a folder called "Files_mat"

In [1]:
```
# Obtain the experimental data from .mat file
import scipy.io as sio
import numpy as np
C_mathilde = sio.loadmat('Files_mat/C_values_mathilde.mat')
# Define the constants
# The stiffness constants in [GPa] --> [g/mm(\mu sec)^2]
# are given by the mathilde .mat (transverse isotropic)
C22_m = np.reshape(C_mathilde['C11'], (30,))#*1E-3
C12_m = np.reshape(C_mathilde['C12'], (30,))
C23_m = np.reshape(C_mathilde['C13'], (30,))#*1E-3
C33_m = np.reshape(C_mathilde['C33'], (30,))#*1E-3
C55_m = np.reshape(C_mathilde['C55'], (30,))#*1E-3
C66_m = np.reshape(C_mathilde['C66'], (30,))
# Obtain the density
d = np.reshape(C_mathilde['d'], (30,))#*1E-3
C_mathilde.keys()
```

Out[1]:
```
dict_keys(['__header__', '__version__', '__globals__', 'C11', 'C12', 'C13', 'C3
3', 'C55', 'C66', 'VL_axial', 'VL_normal', 'VT', 'V_plaque', 'd', 'nu'])
```

Solution for the 2D cell problem (modified exacly to be as in P&G) by FEM

In this section it is defined a class to treat the periodic boundary domain, and different functions to treat the elastic coefficients for the variational formulation of the "cell problem"

In [2]:
```
#%%writefile FunctionsCellProblems.py
# Main libraries to import
from dolfin import *
import ufl as ufl
from mshr import *
import matplotlib.pyplot as plt
plt.style.use("ggplot")
%matplotlib inline

# First, define important classefs for the PDE problem
# Define a class for periodic boundary condition
# over the square mesh
class PeriodicBoundary(SubDomain):
    # Obtain the boundaries of the cube mesh
    def inside(self, x, on_boundary):
        height = near(x[1], a) or near(x[1], -a)
        length = near(x[0], a) or near(x[0], -a)
        bdry = height or length
        return bdry and on_boundary

    # Define mapping for periodicity
    def map(self, x, y):
        Top, Bottom = near(x[1], a), near(x[1], -a)
        Right, Left = near(x[0], a), near(x[0], -a)
        # Define periodicity for the Right--> Left boundary
        if Top:
            y[0] = x[0]
            y[1] = x[1] - 2*a
        # Define periodicity for the Up--> Down boundary
        elif Right:
            y[0] = x[0] - 2*a
            y[1] = x[1]
        # Map anyother point outside the boundary
        else:
            y[0] = 0.
            y[1] = 0.

# Define stiffness array C_{i,j,k,l} for
# the transverse isotropic case
def VoigtToArray(A):
    # Upper diagonal
    A11, A12, A13, A14, A15, A16 = A[0,0], A[0,1], A[0,2], A[0,3], A[0,4], A[0,5]
    A22, A23, A24, A25, A26 = A[1,1], A[1,2], A[1,3], A[1,4], A[1,5]
    A33, A34, A35, A36 = A[2,2], A[2,3], A[2,4], A[2,5]
    A44, A45, A46 = A[3,3], A[3,4], A[3,5]
    A55, A56 = A[4,4], A[4,5]
    A66 = A[5,5]
    # Lower diagonal part (symmetric)
    A21 = A12
    A31, A32 = A13, A23
    A41, A42, A43 = A14, A24, A34
    A51, A52, A53, A54 = A15, A25, A35, A45
    A61, A62, A63, A64, A65 = A16, A26, A36, A46, A56

    return np.array([\
           [\
```

```
            [ [A11, A16, A15], [A16, A12, A14], [A15, A14, A13] ] ,\
            [ [A61, A66, A65], [A66, A62, A64], [A65, A64, A64] ] ,\
            [ [A51, A56, A55], [A56, A52, A54], [A55, A54, A53] ] \
           ], \
           [
            [ [A61, A66, A65], [A66, A62, A64], [A65, A64, A63] ] ,\
            [ [A21, A36, A25], [A26, A22, A24], [A25, A24, A23] ] ,\
            [ [A41, A46, A45], [A46, A42, A44], [A45, A44, A43] ] \
           ], \
           [
            [ [A51, A56, A55], [A56, A52, A54], [A55, A54, A53] ] ,\
            [ [A41, A46, A45], [A46, A42, A44], [A45, A44, A43] ] ,\
            [ [A31, A36, A35], [A36, A32, A34], [A35, A34, A33] ] \
           ] \
                     ])

# Define stiffness tensor C_{i,j,k,l} transverse isotropic
def VoigtToTensor(A):
    # Upper diagonal
    A11, A12, A13, A14, A15, A16 = A[0,0], A[0,1], A[0,2], A[0,3], A[0,4], A[0,5]
    A22, A23, A24, A25, A26 = A[1,1], A[1,2], A[1,3], A[1,4], A[1,5]
    A33, A34, A35, A36 = A[2,2], A[2,3], A[2,4], A[2,5]
    A44, A45, A46 = A[3,3], A[3,4], A[3,5]
    A55, A56 = A[4,4], A[4,5]
    A66 = A[5,5]
    # Lower diagonal part (symmetric)
    A21 = A12
    A31, A32 = A13, A23
    A41, A42, A43 = A14, A24, A34
    A51, A52, A53, A54 = A15, A25, A35, A45
    A61, A62, A63, A64, A65 = A16, A26, A36, A46, A56

    return as_tensor([\
           [\
            [ [A11, A16, A15], [A16, A12, A14], [A15, A14, A13] ] ,\
            [ [A61, A66, A65], [A66, A62, A64], [A65, A64, A64] ] ,\
            [ [A51, A56, A55], [A56, A52, A54], [A55, A54, A53] ] \
           ], \
           [
            [ [A61, A66, A65], [A66, A62, A64], [A65, A64, A63] ] ,\
            [ [A21, A36, A25], [A26, A22, A24], [A25, A24, A23] ] ,\
            [ [A41, A46, A45], [A46, A42, A44], [A45, A44, A43] ] \
           ], \
           [
            [ [A51, A56, A55], [A56, A52, A54], [A55, A54, A53] ] ,\
            [ [A41, A46, A45], [A46, A42, A44], [A45, A44, A43] ] ,\
            [ [A31, A36, A35], [A36, A32, A34], [A35, A34, A33] ] \
           ] \
                     ])


# Define stiffness tensor C_{i,j,r,s}
# where the indices r,s are fixed
def VoigtToTensorContract2idx(A, r, s):
    # Obtain all 3x3 array elements at index r,s fixed
    # From VoigtToArray representation
    A11, A12, A13 = A[0,0,r,s], A[0,1,r,s], A[0,2,r,s]
```

```
    A21, A22, A23 = A[1,0,r,s], A[1,1,r,s], A[1,2,r,s]
    A31, A32, A33 = A[2,0,r,s], A[2,1,r,s], A[2,2,r,s]

    return as_tensor([\
                      [A11, A12, A13], \
                      [A21, A22, A23], \
                      [A31, A32, A33] \
                     ])
# Define useful constant
Zero = Constant(0.0)
# Define the strain tensor
def strain(N):
    E11, E12, E13 = N[0].dx(0), 0.5*(N[0].dx(1)+N[1].dx(0)), Zero
    E21, E22, E23 = 0.5*(N[1].dx(0)+N[0].dx(1)), N[1].dx(1), Zero
    E31, E32, E33 = 0.5*(N[2].dx(0)), 0.5*(N[2].dx(1)), Zero
    return as_tensor([\
                      [E11, E12, E13],\
                      [E21, E22, E23],\
                      [E31, E32, E33]\
                     ])
# Define 2D->3D partial derivatives tensor
def deriv3d(w):
    S11, S12, S13 = w[0].dx(0), w[0].dx(1), Zero
    S21, S22, S23 = w[1].dx(0), w[1].dx(1), Zero
    S31, S32, S33 = w[2].dx(0), w[2].dx(1), Zero
    return as_tensor([\
                      [S11, S12, S13],\
                      [S21, S22, S23],\
                      [S31, S32, S33]\
                     ])
# Define indices
i,j,k,l = ufl.indices(4)
# Define stress tensor
def sigma(N, R):
    """
    Input: N --> Solution to the cell problem and
           R --> Tensor array being considered.
    """
    return as_tensor(R[i,j,k,l]*strain(N)[k,l], (i,j))
```

In this part, the CellProblemSol function is defined, which solves the cell problem given certain parameters.

In [5]:
```
#%%writefile MainCellProblems.py
def CellProblemSol(mesh, C_matrix, C_marrow, porosity, structure, indexes,
                   epsilon, omega, save=False):
    """
    CellProblemSol computes the solution of the Cell problem by
    FEM given as input a particular porosity and structure.

    Input parameters
    mesh: A mesh structure where the cell problem is defined.
    C_matrix: Voigt tensor material of bone matrix.
    C_marrow: Voigt tensor material of bone marrow.
    porosity: Value of porosity on the interval [0, 0.3].
    structure: 'Circular' or 'Rectangular'.
    indexes: Indexes to compute the homogenization procedure.
    epsilon: parameters for attenuation as dictionary (viscosity)
    omega: frequency associated to the model being considered.
    Output parameters ----COMPLETE!
    """
    global dx, ds, C, D
    # Define polynomial degrees
    pdim = 1
    # Define mixed function space and boundary conditions
    V = VectorElement("CG", mesh.ufl_cell(), degree=pdim, dim=3)
    V_elem = MixedElement([V, V])
    W = FunctionSpace(mesh, V_elem)
    # Define trial and test functions
    (N_R, N_I) = TrialFunctions(W)
    (w_R, w_I) = TestFunctions(W)

    # Define normal direction of mesh
    #n = FacetNormal(mesh)
    # Obtain the boundries
    bdries = MeshFunction("size_t", mesh,
                          mesh.topology().dim()-1)
    # Define new measure for boundaries
    dx = dx(domain=mesh)
    ds = ds(domain=mesh, subdomain_data=bdries)

    # Define parameters for cell inclusion parameterization
    l = 0.0
    # Obtain the type of structure:
    if structure == 'Circular':
        # Here l is the radius^2 of the cilinder for cell
        l = porosity/np.pi
        C_jit = "(pow(x[0],2)+pow(x[1],2)<l)? Mw: Mb"
    elif structure == 'Rectangular':
        # If its a rectangular inclusion
        # Here l is the half length of inclusion for cell
        l = np.sqrt(porosity)/2
        C_jit = "(std::abs(x[0])<l && std::abs(x[1])<l) \
                 ? Mw: Mb"
    else:
        raise ValueError('Just two type of structures suppported.')

    # Assign attenuation values at each component
    eps_b = epsilon['epsilon_b'] # Attenuation on matrix
```

```
eps_w = epsilon['epsilon_w'] # Attenuation on marrow

# Create expressions for voigt C_{ij} elastic coeffs.
C11 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[0,0], Mw=C_marrow[0,0], l=l)
C12 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[0,1], Mw=C_marrow[0,1], l=l)
C22 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[1,1], Mw=C_marrow[1,1], l=l)
C13 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[0,2], Mw=C_marrow[0,2], l=l)
C23 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[1,2], Mw=C_marrow[1,2], l=l)
C33 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[2,2], Mw=C_marrow[2,2], l=l)
C44 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[3,3], Mw=C_marrow[3,3], l=l)
C55 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[4,4], Mw=C_marrow[4,4], l=l)
C66 = Expression(C_jit, degree=pdim,
                 Mb=C_matrix[5,5], Mw=C_marrow[5,5], l=l)

# Create expressions for voigt viscous D_{ij} coeffs.
D11 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[0,0], Mw=eps_w*C_marrow[0,0], l=l)
D12 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[0,1], Mw=eps_w*C_marrow[0,1], l=l)
D22 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[1,1], Mw=eps_w*C_marrow[1,1], l=l)
D13 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[0,2], Mw=eps_w*C_marrow[0,2], l=l)
D23 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[1,2], Mw=eps_w*C_marrow[1,2], l=l)
D33 = Expression(C_jit, degree=pdim,
                 Mb=eps_b*C_matrix[2,2], Mw=eps_w*C_marrow[2,2], l=l)
D44 = Expression(C_jit, degree=pdim,
                 Mb=2*eps_b*C_matrix[3,3], Mw=eps_w*C_marrow[3,3], l=l)
D55 = Expression(C_jit, degree=pdim,
                 Mb=2*eps_b*C_matrix[4,4], Mw=eps_w*C_marrow[4,4], l=l)
D66 = Expression(C_jit, degree=pdim,
                 Mb=2*eps_b*C_matrix[5,5], Mw=eps_w*C_marrow[5,5], l=l)
# Construct the voigt matrix representation
# For the elasticity tensor
C_voigt = np.array([\
                [C11, C12, C13, 0, 0, 0], \
                [C12, C22, C23, 0, 0, 0], \
                [C13, C23, C33, 0, 0, 0], \
                [0, 0, 0, C44, 0, 0], \
                [0, 0, 0, 0, C55, 0], \
                [0, 0, 0, 0, 0, C66] \
                   ])
# For the viscosity tensor
D_voigt = np.array([\
                [D11, D12, D13, 0, 0, 0], \
                [D12, D22, D23, 0, 0, 0], \
                [D13, D23, D33, 0, 0, 0], \
                [0, 0, 0, D44, 0, 0], \
```

```
                [0, 0, 0, 0, D55, 0], \
                [0, 0, 0, 0, 0, D66] \
                   ])
# Obtain the C and D in tensor representation
C = VoigtToTensor(C_voigt)
D = VoigtToTensor(D_voigt)
# The rhs block A_block is defined for fixed indexes r,s
def A_block(N, w, R):
    # Receives R as a fourth rank tensor
    return (sigma(N, R)[i,j]*w[i].dx(j))*dx
# In the B_block, we fix indexes r,s
def B_block(w, r, s, R_voigt):
    # Obtain array for indexes r,s
    R_array = VoigtToArray(R_voigt)
    # Compute tensor fixing r,s indexes
    R_ct = VoigtToTensorContract2idx(R_array,r,s)
    return -(R_ct[i,j]*deriv3d(w)[i,j])*dx

# Define boundary conditions
jit_ext = "(near(x[0], -0.5) || near(x[0], 0.5) || \
           near(x[1], -0.5) || near(x[1], 0.5)) && on_boundary"
#jit_ext = "(near(x[0], 0.0) && near(x[1], 0.0)) || on_boundary"
bc_ext_C = DirichletBC(W.sub(0), Constant(3*(0.,)), jit_ext)#"on_boundary")
bc_ext_D = DirichletBC(W.sub(1), Constant(3*(0.,)), jit_ext)#"on_boundary")
# List boundary conditions
bcs = [bc_ext_C, bc_ext_D]

# Create Krylov solver
solver = PETScKrylovSolver("gmres", "ilu")
solver.parameters["absolute_tolerance"] = 1E-7
solver.parameters["maximum_iterations"] = 2000
solver.parameters["monitor_convergence"] = False

# Define solution for the variational formulation
N_sol = Function(W)
# Save coeffs of tensor as 4d array
C_array = np.zeros((3,3,3,3))
D_array = np.zeros((3,3,3,3))
# Save coeffs of Q-factor decomposition
IQ_lin = np.zeros_like(C_array)
IQ_nlin = np.zeros_like(C_array)
# Iterate over indexes
for id_i, id_j, r, s in indexes:
    # Variational forms associated to the real and imaginary parts
    A_lhs_R = A_block(N_R, w_R, C) - \
              Constant(omega)*A_block(N_I, w_R, D)
    A_lhs_I = A_block(N_I, w_I, C) + \
              Constant(omega)*A_block(N_R, w_I, D)
    # Compute rhs associated to both (real and imag parts)
    b_rhs_R = B_block(w_R, r, s, C_voigt)
    b_rhs_I = Constant(omega)*B_block(w_I, r, s, D_voigt)

    # Assemble of matrices
    A = assemble(A_lhs_R+A_lhs_I)
    b = assemble(b_rhs_R+b_rhs_I)
    # Apply boundary conditions for both problems
    [bc.apply(A, b) for bc in bcs]
```

```
        # Create vector that spans the null space
        null_vec = Vector(N_sol.vector())
        W.dofmap().set(null_vec, 1.0)
        null_vec *= 1.0/null_vec.norm("l2")
        # Create null space basis object and atach
        # to PERTSc matrix
        null_space = VectorSpaceBasis([null_vec])
        as_backend_type(A).set_nullspace(null_space)
        # Ortogonalize b with respect to the null space
        null_space.orthogonalize(b)
        # Add A matrix representation to Krylov solver
        solver.set_operator(A)
        # Solve the variational problem
        solver.solve(N_sol.vector(), b)

        # Split solution in real and imag parts
        (sol_R, sol_I) = N_sol.split(True)

        # If selected, save solutions
        # ADD SAVING FOR VISCOUS SOLUTIONS!!!!
        #if save and (r,s) == (0,0):
            #str_idx = str(id_i)+str(id_j)+str(r)+str(s)
            #file_to_save ="Results/CellProblems/CellProblemsol3D_idx"+\
            #              str_idx+"Por"+str(porosity)+".pvd"
            #File(file_to_save) << N_sol


        # Compute homogenized coefficient
        C_elem = assemble(C[id_i,id_j,r,s]*dx)
        D_elem = assemble(D[id_i,id_j,r,s]*dx)
        # Obs: We are considering the decomposition given by
        # C^hom + i \omega D^hom
        C_sgm = assemble(sigma(sol_R, C)[id_i,id_j]*dx) - \
                assemble(sigma(sol_I, D)[id_i,id_j]*dx)*omega
        D_sgm = assemble(sigma(sol_I, C)[id_i,id_j]*dx)*pow(omega,-1) + \
                assemble(sigma(sol_R, D)[id_i,id_j]*dx)
        C_array[id_i,id_j,r,s] = C_elem+C_sgm
        D_array[id_i,id_j,r,s] = D_elem+D_sgm
        # Define inverse of Q-factor coefficient (Re/Im part)
        # by computing its linear and linear parts
        # First, its linear part!
        IQ_lin[id_i,id_j,r,s] = D_elem/C_elem
        # Now, the nonlinear one!
        aux = C_elem*(C_elem+C_sgm)
        IQ_nlin[id_i,id_j,r,s] = (C_elem*(D_elem+D_sgm)-D_elem*(C_elem+C_sgm))/au

        # FOR TESTING!!
        print("""
              At indexes: ({0},{1},{2},{3})
              The hom. coeff. C: {4:.4f}
              The hom. coeff. D: {5:.4f}
              IQ (Linear, NonLinear) --> {6:.4f}, {7:.4f}
              """.format(id_i,id_j,r,s,
                         C_array[id_i,id_j,r,s],
                         D_array[id_i,id_j,r,s],
                         IQ_lin[id_i,id_j,r,s],
```

```
                        IQ_nlin[id_i,id_j,r,s]))

    # Return the array
    return C_array, D_array, IQ_lin, IQ_nlin
```

Compute homogenized coeffs. for the rectangular inclusion

For the stabble coefficients C_{22}, C_{33}, C_{23}

In [32]:
```
# Problem solution for testing
part = 50 # Number of partitions
# Creates a square as microstructure
box_mesh = RectangleMesh(Point(-0.5,-0.5),Point(0.5,0.5),
                         part, part)
"""
# Refine box mesh twice
for _ in range(1):
    # First mark all cells
    cell_markers = MeshFunction("bool", box_mesh, 3)
    cell_markers.set_all(False)
    # assign a different mark for the some cells
    for cell in cells(box_mesh):
        p = cell.midpoint()
        if np.abs(p.x()) < 0.15 and np.abs(p.y()) < 0.15:
            cell_markers[cell] = True

    # Refine over the interest domain
    box_mesh = refine(box_mesh, cell_markers, redistribute=True)
"""

# Define parameters of length "a" for box mesh
a = 0.5 # Fixed!
print("""
      Minimum height of element [mm]: {0:.5f};
      Number Cells; {1}, Vertice: {2}
      """.format(box_mesh.hmin(),
                 box_mesh.num_cells(),
                 box_mesh.num_vertices()))
# Test for PETSc
if not has_linear_algebra_backend("PETSc"):
    print(" Not configured PETSc compiling ")

# Define the Voigt matrix stiffness representation
"""
# for material Mb: bone
Mb = np.array([\
               [26.8, 15.2, 15.3, 0, 0, 0], \
               [15.2, 26.8, 15.3, 0, 0, 0], \
               [15.3, 15.3, 35.1, 0, 0, 0], \
               [0, 0, 0, 7.3, 0, 0], \
               [0, 0, 0, 0, 7.3, 0], \
               [0, 0, 0, 0, 0, 5.8]
              ])
"""
# Input elastic coeffs. proposed by Minonzio from SB Data
# Define the Voigt matrix stiffness representation
Mb = np.array([\
               [18.7, 8.84, 10.1, 0, 0, 0], \
               [8.84, 18.7, 10.1, 0, 0, 0], \
               [10.1, 10.1, 31.0, 0, 0, 0], \
               [0, 0, 0, 6.98, 0, 0], \
               [0, 0, 0, 0, 6.98, 0], \
               [0, 0, 0, 0, 0, 4.93]
              ])
# for material Mw: water
```

```
Mw = np.array([\
               [2.0537, 1.9732, 1.9732, 0, 0, 0], \
               [1.9732, 2.0537, 1.9732, 0, 0, 0], \
               [1.9732, 1.9732, 2.0537, 0, 0, 0], \
               [0, 0, 0, 0.0468, 0, 0], \
               [0, 0, 0, 0, 0.0468, 0], \
               [0, 0, 0, 0, 0, 0.0468], \
              ])
# Define an array of porosities
porosities = np.arange(0.01, 0.48, step=0.01)
Npor = len(porosities)
## Let us check for the case Rectangular
structure = 'Circular'
#structure = 'Rectangular'
# Define the array of homogenized coeffs
C_hom = np.zeros((3,3,3,3,Npor))
D_hom = np.zeros_like(C_hom)
IQ_lin_hom = np.zeros_like(C_hom)
IQ_nlin_hom = np.zeros_like(C_hom)
# Indexes will be given only main coeffs
indexes = [(0,0,0,0), (0,0,1,1), (0,1,0,1),
           (2,2,2,2), (1,1,2,2), (1,2,1,2)]
saved = False
# Amount of attenuation for our model
# bone matrix att. at 1E-2 and marrow att at 1E-3 standard.
#epsilon = {'epsilon_b': 1.4E-3, 'epsilon_w': 6E-3}
# epsilon_w = 1E-4 1E-3 1E-2 1E-1
epsilon = {'epsilon_b': 1E-2, 'epsilon_w': 1E-1}
# Radial frequency associated to the simulation
omega = 0.5 # in [Mhz], 0.5 standard
# Iterate over the porosities
for phi in range(Npor):
    try:
        if porosities[phi] in [0.05, 0.15, 0.2]:
            saved = False
        else:
            saved = False
        # Solve cell problems
        print("At porosity: ", porosities[phi])
        C_array, D_array, IQ_lin_array, IQ_nlin_array = CellProblemSol(box_mesh,
                                                                       porosities[phi
                                                                       epsilon, omega

        C_hom[:,:,:,:,phi] = C_array
        D_hom[:,:,:,:,phi] = D_array
        IQ_lin_hom[:,:,:,:,phi] = IQ_lin_array
        IQ_nlin_hom[:,:,:,:,phi] = IQ_nlin_array

    except RuntimeError:
        print('Failed at porosity: ', porosities[phi])
        break
```

```
At indexes: (0,0,1,1)
The hom. coeff. C: 3.6168
The hom. coeff. D: 0.2244
IQ (Linear, NonLinear) --> 0.0251, 0.0370
```

```
At indexes: (0,1,0,1)
The hom. coeff. C: 1.7990
The hom. coeff. D: 0.0353
IQ (Linear, NonLinear) --> 0.0207, -0.0011


At indexes: (2,2,2,2)
The hom. coeff. C: 16.0383
The hom. coeff. D: 0.3149
IQ (Linear, NonLinear) --> 0.0151, 0.0046


At indexes: (1,1,2,2)
The hom. coeff. C: 4.4522
```

Saving the data obtained.

Up to now I give names to the output files manually, since here we are I'm doing prototyping with different cases.
The output files will be located on folder called "DataPICKLE" (which must already exists before on the directory), where the name of such file is given below as "output".

In [9]:
```
# Save data in pickle format
import pickle, io
# Define data to save
data_save = {'structure': structure,
             'MinHeigth': box_mesh.hmin(),
             'NumCells': box_mesh.num_cells(),
             'NumVerts': box_mesh.num_vertices(),
             'C22': list(C_hom[0,0,0,0,:]),# Elastic Coeffs
             'C66': list(C_hom[0,1,0,1,:]),
             'C33': list(C_hom[2,2,2,2,:]),
             'C23': list(C_hom[1,1,2,2,:]),
             'C55': list(C_hom[1,2,1,2,:]),
             'C12': list(C_hom[0,0,1,1,:]),
             'D22': list(D_hom[0,0,0,0,:]),# Viscous Coeffs
             'D66': list(D_hom[0,1,0,1,:]),
             'D33': list(D_hom[2,2,2,2,:]),
             'D23': list(D_hom[1,1,2,2,:]),
             'D55': list(D_hom[1,2,1,2,:]),
             'D12': list(D_hom[0,0,1,1,:]),
             'IQ_lin_22': list(IQ_lin_hom[0,0,0,0,:]),# Linear Part Q^{-1}
             'IQ_lin_66': list(IQ_lin_hom[0,1,0,1,:]),
             'IQ_lin_33': list(IQ_lin_hom[2,2,2,2,:]),
             'IQ_lin_23': list(IQ_lin_hom[1,1,2,2,:]),
             'IQ_lin_55': list(IQ_lin_hom[1,2,1,2,:]),
             'IQ_lin_12': list(IQ_lin_hom[0,0,1,1,:]),
             'IQ_nlin_22': list(IQ_nlin_hom[0,0,0,0,:]),# Nonlinear Part Q^{-1}
             'IQ_nlin_66': list(IQ_nlin_hom[0,1,0,1,:]),
             'IQ_nlin_33': list(IQ_nlin_hom[2,2,2,2,:]),
             'IQ_nlin_23': list(IQ_nlin_hom[1,1,2,2,:]),
             'IQ_nlin_55': list(IQ_nlin_hom[1,2,1,2,:]),
             'IQ_nlin_12': list(IQ_nlin_hom[0,0,1,1,:])
            }
# Define output file
output = 'DataPICKLE/Visc_'+str(structure)+'2DPart'+ \
         str(part)+'m14-3_f6-3_Freq'+str(omega)+'_Tibia.pickle'
with io.open(output, 'wb') as handle:
    pickle.dump(data_save, handle,
                protocol=pickle.HIGHEST_PROTOCOL)
```

In [33]:
```
import numpy as np
mi_matriz = np.column_stack([ list(C_hom[0,0,0,0,:]),
               list(C_hom[0,1,0,1,:]),
               list(C_hom[2,2,2,2,:]),
               list(C_hom[1,1,2,2,:]),
               list(C_hom[1,2,1,2,:]),
               list(C_hom[0,0,1,1,:]),
              list(D_hom[0,0,0,0,:]),
               list(D_hom[0,1,0,1,:]),
               list(D_hom[2,2,2,2,:]),
               list(D_hom[1,1,2,2,:]),
               list(D_hom[1,2,1,2,:]),
               list(D_hom[0,0,1,1,:]) ])
np.savetxt("1E-1.csv", mi_matriz, delimiter=",")
```

Plot relations between C_{ijkl} and Q^{-1}_{ijkl}

Here the plot of various cases are done, in particular the ones related to the paper.

First, just to check the data from Grimal, we make a basic plot.

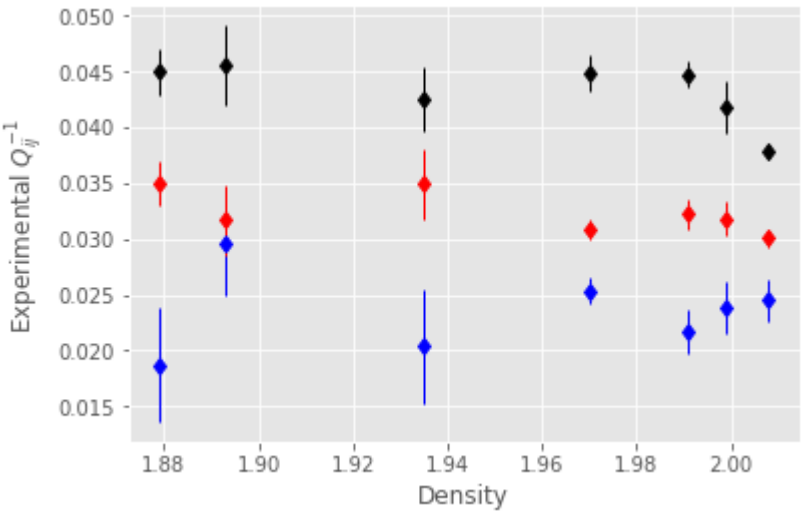
The files imported will be "GrimalDataV1p3.mat and "Q_aniso_Grimal.mat", both of them must be available on the folder "Files_mat"

In [6]:
```
# Obtain the experimental data from .mat file
import scipy.io as sio
import numpy as np
C_tibia_PG = sio.loadmat('Files_mat/GrimalDataV1p3.mat')
# Define the constants
# The stiffness constants have units in [GPa]
C22_pg = C_tibia_PG['C11'].squeeze()
C12_pg = C_tibia_PG['C12'].squeeze()
C13_pg = C_tibia_PG['C13'].squeeze()
C22_pg = C_tibia_PG['C22'].squeeze()
C33_pg = C_tibia_PG['C33'].squeeze()
C44_pg = C_tibia_PG['C44'].squeeze()
C66_pg = C_tibia_PG['C66'].squeeze()
# Define porosity function to plot Tibia PG predictions
porosity_pg = np.arange(0.01, 0.49, step=0.01)
# Obtain density function
density_pg = 2.03*(1-porosity_pg) + porosity_pg

# Import Q values from Grimal data
Q_grimal_data = sio.loadmat('Files_mat/Q_aniso_Grimal.mat')
# Extract density, Q^{-1} values and errors
d_grimal = Q_grimal_data['Q_aniso'][0, 0][0]
Q_inv_grimal = Q_grimal_data['Q_aniso'][0, 0][1]
Q_inv_error_grimal = Q_grimal_data['Q_aniso'][0, 0][2]
Q_grimal_data.keys()
```

Out[6]:
```
dict_keys(['__header__', '__version__', '__globals__', 'Q_aniso'])
```

In [7]:
```
import pickle, io
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[0, :], \
             yerr=Q_inv_error_grimal[0, :], fmt='db', linewidth=1)
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[1, :], \
             yerr=Q_inv_error_grimal[1, :], fmt='dk', linewidth=1)
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[2, :], \
             yerr=Q_inv_error_grimal[2, :], fmt='dr', linewidth=1)
plt.xlabel('Density')
plt.ylabel('Experimental $Q_{ij}^{-1}$ ')
```

Out[7]:
```
Text(0, 0.5, 'Experimental $Q_{ij}^{-1}$ ')
```



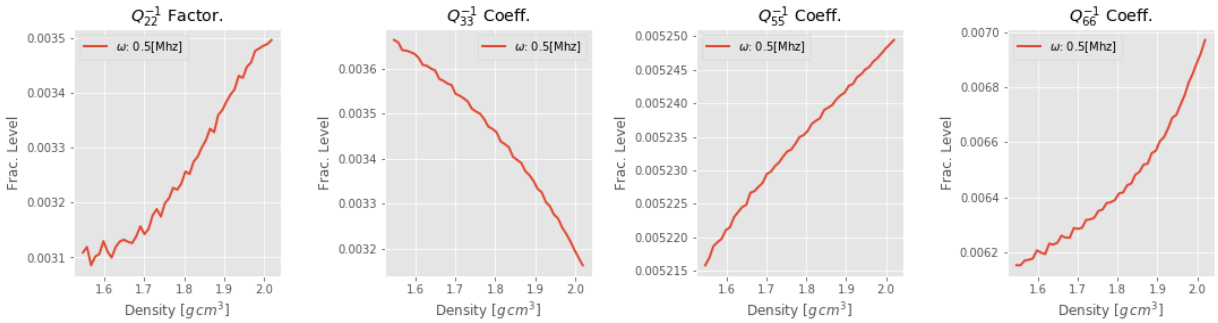
Now, a small relabeling of data is done, just for convenience.

In [8]:
```
# Import Tibia Data from Grimal file
tibia_data = sio.loadmat('Files_mat/Data_Tibia_Grimal.mat')
# Define variables associated to each measurement
Q44_tibia = tibia_data['DataTibia'][0, 0][0].squeeze()
Q44_sens_tibia = tibia_data['DataTibia'][0, 0][1].squeeze()
QE3_tibia = tibia_data['DataTibia'][0, 0][2].squeeze()
QE1_tibia = tibia_data['DataTibia'][0, 0][3].squeeze()
Qshear_tibia = tibia_data['DataTibia'][0, 0][4].squeeze()
density_tibia = tibia_data['DataTibia'][0, 0][5].squeeze()
C11_tibia = tibia_data['DataTibia'][0, 0][6].squeeze()
C33_tibia = tibia_data['DataTibia'][0, 0][7].squeeze()
C13_tibia = tibia_data['DataTibia'][0, 0][8].squeeze()
C44_tibia = tibia_data['DataTibia'][0, 0][9].squeeze()
C66_tibia = tibia_data['DataTibia'][0, 0][10].squeeze()
#tibia_data['DataTibia']
```

Now, Q-factor coefficients are plotted against the density parameter.

In [9]:
```
import pickle, io
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
# First, Obtain the structure
structure = 'Circular'
#structure = 'Rectangular'
# Obtain porosity array
porosities_sim = np.arange(0.01, 0.48, step=0.01)
# Density function, derived from sb data
density_sim = 2.03*(1-porosities_sim) + porosities_sim
# Radial frequency associated to the simulation --> omega [Mhz]
# Now, create the plot figure
fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(20,4))
fig.subplots_adjust(left=0.2, wspace=0.5)
# Define legend label
legend_name = []
# Obtain the mesh partition, fixed!
part = 50
for omega in [0.5]: #[0.1, 0.2, 0.5]
    filename = 'DataPICKLE/Visc_'+str(structure)+'2DPart'+\
                str(part)+'m14-3_f6-3_Freq'+str(omega)+'_Tibia.pickle'
    legend_name.append('$\omega$: '+str(omega)+'[Mhz]')
    with open(filename, 'rb') as handle:
        data_saved = pickle.load(handle)
    # Now, plot the case of Inverse Q_22
    IQ_22 = np.array(data_saved['D22'])/\
            np.array(data_saved['C22'])
    ax[0].plot(density_sim, omega*IQ_22, linewidth=2)
    # plot the case of Inverse Q_33
    IQ_33 = np.array(data_saved['D33'])/\
            np.array(data_saved['C33'])
    ax[1].plot(density_sim, omega*IQ_33, linewidth=2)
    # Now, for the case inverse Q_44 = Q_55
    IQ_55 = np.array(data_saved['D55'])/\
            np.array(data_saved['C55'])
    ax[2].plot(density_sim, omega*IQ_55, linewidth=2)
    # Finally, case of inverse Q_66
    IQ_66 = np.array(data_saved['D66'])/\
            np.array(data_saved['C66'])
    ax[3].plot(density_sim, omega*IQ_66, linewidth=2)

# Add more formating to the plot.
for n in range(0,4):
    if n == 0:
        ax[n].set_title('$Q^{-1}_{22}$ Factor.')
    elif n == 1:
        ax[n].set_title('$Q^{-1}_{33}$ Coeff.')
    elif n == 2:
        ax[n].set_title('$Q^{-1}_{55}$ Coeff.')
    else:
        ax[n].set_title('$Q^{-1}_{66}$ Coeff.')

    # set ylabel at each case
    ax[n].set_ylabel('Frac. Level')
```

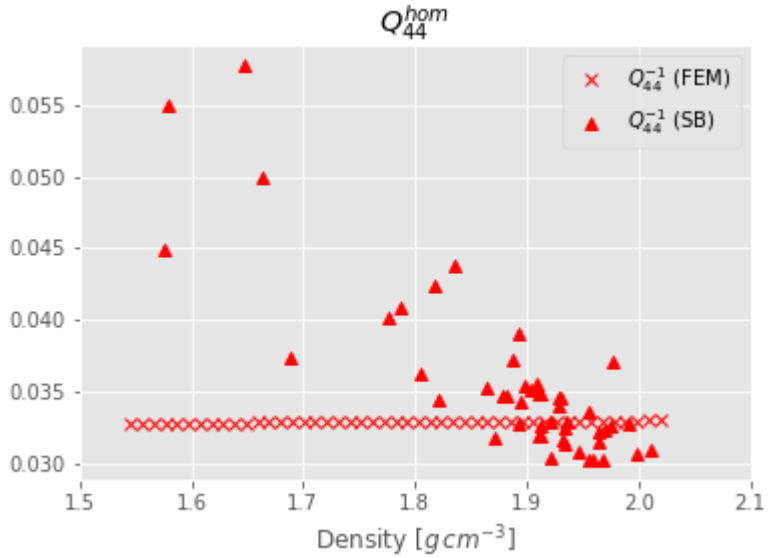
```
    # Mark the other options for the figure
    ax[n].set_xlabel('Density $[g\, cm^{3}]$')
    #ax[n].set_xlim([0.0, 0.3])
    # Add legend
    ax[n].legend(legend_name, loc='best')
    #ax[n].grid()
# save it!
#filename = 'Plots/CellProb_Qfactor'+str(structure)+\
#           'E-3.png'
#plt.savefig(filename, dpi=150, bbox_inches='tight')
# show plot
plt.show()
```



Moreover, here it's saved the comparison between Q_{44} factor simulated and reference.

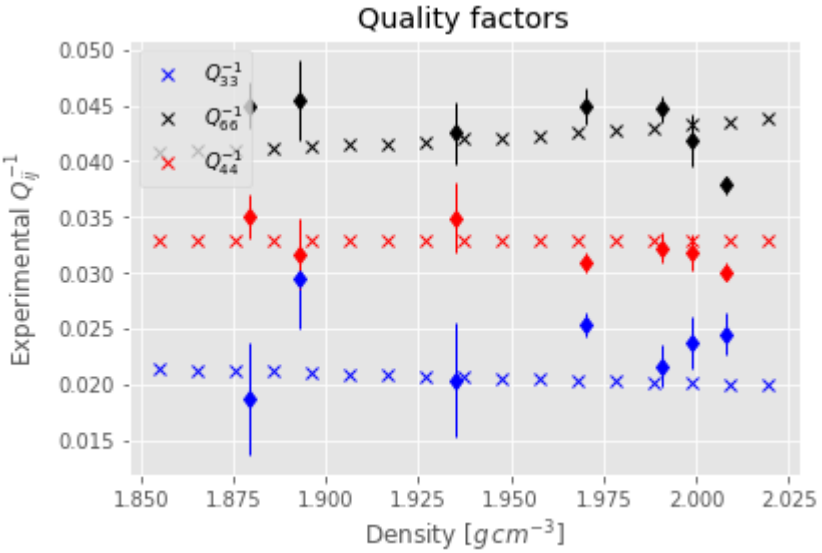
Such file is saved in a folder within the actual directory, called "Plots" that must exits already.

In [10]:
```
# Plot the Q_{44} quality factor from SB data
plt.title("$Q_{44}^{hom}$")
aux_55 = np.array(data_saved['D55'])/np.array(data_saved['C55'])
plt.plot(density_sim, 2*np.pi*0.5*aux_55, 'rx', linewidth=2)

Q44_indx = Q44_sens_tibia >= 0.8
plt.plot(density_tibia[Q44_indx], 1/Q44_tibia[Q44_indx], \
         'r^', linewidth=2)
plt.legend(["$Q_{44}^{-1}$ (FEM)", "$Q_{44}^{-1}$ (SB)"])
plt.xlabel("Density $[g \, cm^{-3}]$")
plt.xlim([1.5, 2.1])
#plt.ylabel('$Q_{ij}^{-1}$')
#plt.grid()
filename = 'Plots/Q44_'+str(structure)+\
           'm14-3_f6-3_Freq01_Rel_Tibia'
for _ in ['.pdf', '.png']:
    plt.savefig(filename+_, dpi=200, bbox_inches='tight')

plt.show()
```



Now, it's only plotted the comparison between the other Q_{ij} factors against the references

In [11]:
```
# Consider only relevant simulation for density > 1.85
relevant_indx = density_sim > 1.85
plt.title("Quality factors")

aux_33 = 2*np.pi*np.array(data_saved['D33'])/np.array(data_saved['C33'])
plt.plot(density_sim[relevant_indx], 0.5*aux_33[relevant_indx],
         'bx', linewidth=2)
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[0, :], \
             yerr=Q_inv_error_grimal[0, :], fmt='db', linewidth=1)

aux_66 = 2*np.pi*np.array(data_saved['D66'])/np.array(data_saved['C66'])
plt.plot(density_sim[relevant_indx], 0.5*aux_66[relevant_indx],
         'kx', linewidth=2)
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[1, :], \
             yerr=Q_inv_error_grimal[1, :], fmt='dk', linewidth=1)


aux_55 = 2*np.pi*np.array(data_saved['D55'])/np.array(data_saved['C55'])
plt.plot(density_sim[relevant_indx], 0.5*aux_55[relevant_indx],
         'rx',linewidth=2)
plt.errorbar(d_grimal.squeeze(), Q_inv_grimal[2, :], \
             yerr=Q_inv_error_grimal[2, :], fmt='dr', linewidth=1)

plt.legend(["$Q_{33}^{-1}$","$Q_{66}^{-1}$","$Q_{44}^{-1}$"])
plt.xlabel("Density $[g \, cm^{-3}]$")
plt.ylabel('Experimental $Q_{ij}^{-1}$')
plt.show()
#plt.grid()
```

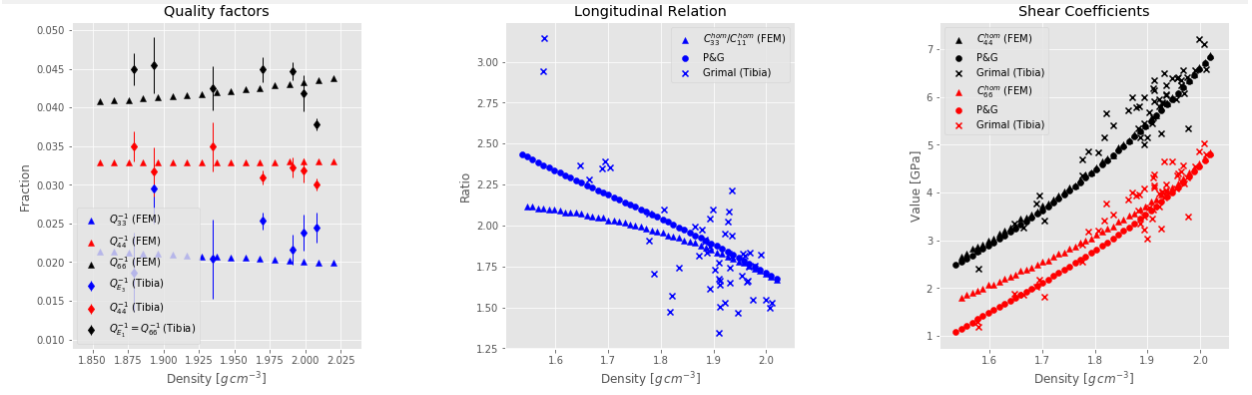


The main figure is now generated, it contains the quality factor relations, Longitudinal Elastic and shear coefficient relations.

As before, the plot is saved on a the folder "Plots" that must be already on the directory.

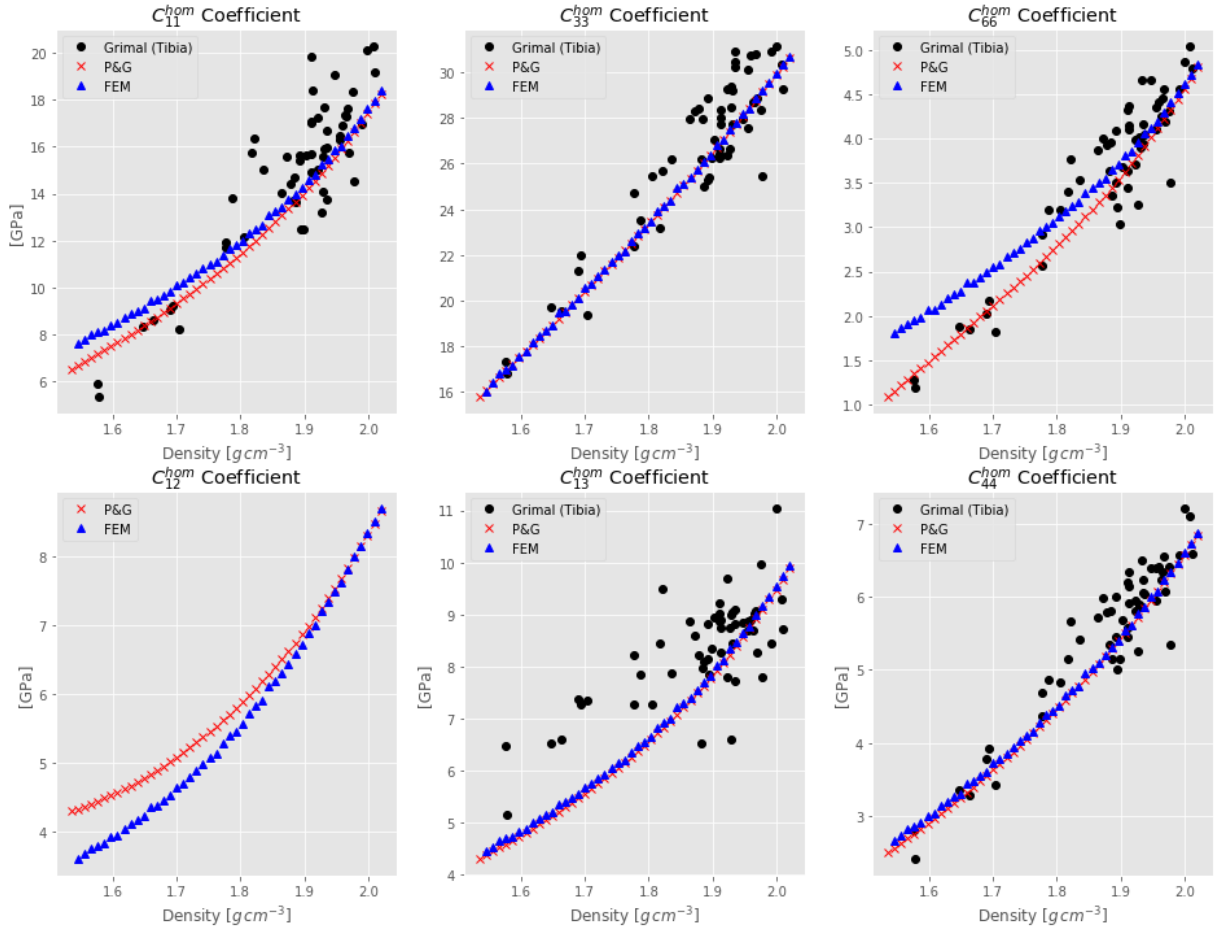
In [12]:
```
# Create the plot figure
#fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(17,4))
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(23,6))
fig.subplots_adjust(left=0.2, wspace=0.5)
# Define legend label
legend_name = []
# Obtain the mesh partition, fixed!
part = 50
# Iterate over the files
# plot the case of Inverse Q_33
IQ_33 = 2*np.pi*0.5*np.array(data_saved['D33'])/np.array(data_saved['C33'])
ax[0].scatter(density_sim[relevant_indx], IQ_33[relevant_indx], color='b', marker
ax[0].errorbar(d_grimal.squeeze(), Q_inv_grimal[0, :], \
               yerr=Q_inv_error_grimal[0, :], fmt='db', linewidth=1)

# Now, for the case inverse Q_44 = Q_55
IQ_55 = 2*np.pi*0.5*np.array(data_saved['D55'])/np.array(data_saved['C55'])
ax[0].scatter(density_sim[relevant_indx], IQ_55[relevant_indx], color='r', marker

ax[0].errorbar(d_grimal.squeeze(), Q_inv_grimal[2, :], \
               yerr=Q_inv_error_grimal[2, :], fmt='dr', linewidth=1)

# Finally, case of inverse Q_66
IQ_66 = 2*np.pi*0.5*np.array(data_saved['D66'])/np.array(data_saved['C66'])
ax[0].scatter(density_sim[relevant_indx], IQ_66[relevant_indx], color='k', marker

ax[0].errorbar(d_grimal.squeeze(), Q_inv_grimal[1, :], \
               yerr=Q_inv_error_grimal[1, :], fmt='dk', linewidth=1)

# Set limits of both boundaries
#ax[0].set_xlim([1.59, 1.92])
#ax[0].set_ylim([0.046, 0.0505])

# plot the elasctic coefficients also
ratio_sim = np.array(data_saved['C33'])/\
            np.array(data_saved['C22'])
ax[1].scatter(density_sim, ratio_sim, color='b', \
              marker='^')
ax[1].scatter(density_pg, C33_pg/C22_pg, color='b', marker='o')
ax[1].scatter(density_tibia, C33_tibia/C11_tibia, color='b', \
              marker='x')
#ax[1].set_xlim([1.59, 1.92])
#ax[1].set_ylim([1.1, 3])
#Ratio_5566 = np.array(data_saved['C55'])/\
#             np.array(data_saved['C66'])
#ax[1].plot(d, Ratio_5566, 'ro--', linewidth=2)

# Plot other coeffs for comparison with Bernard et al. 2015
ax[2].scatter(density_sim, np.array(data_saved['C55']), \
              color='k', marker='^')
ax[2].scatter(density_pg, C44_pg, color='k', marker='o')
ax[2].scatter(density_tibia, C44_tibia, \
              color='k', marker='x')

ax[2].scatter(density_sim, np.array(data_saved['C66']), \
              color='r', marker='^')
```

```
ax[2].scatter(density_pg, C66_pg, color='r', marker='o')
ax[2].scatter(density_tibia, C66_tibia, \
              color='r', marker='x')

#ax[2].set_xlim([1.59, 1.92])

# Add more formating to the plot.
# set ylabel at each case
ax[0].set_ylabel('Fraction')
ax[1].set_ylabel('Ratio')
ax[2].set_ylabel('Value [GPa]')
# Mark the other options for the figure
ax[0].set_xlabel('Density $[g\, cm^{-3}]$')
ax[1].set_xlabel('Density $[g\, cm^{-3}]$')
ax[2].set_xlabel('Density $[g\, cm^{-3}]$')
# Give titles to the plots
ax[0].set_title('Quality factors')
ax[1].set_title('Longitudinal Relation')
ax[2].set_title('Shear Coefficients')

# Add legend
ax[0].legend(["$Q_{33}^{-1}$ (FEM)", "$Q_{44}^{-1}$ (FEM)", "$Q_{66}^{-1}$ (FEM)'
              "$Q_{E_{3}}^{-1}$ (Tibia)", "$Q_{44}^{-1}$ (Tibia)", "$Q_{E_{1}}^{-
              loc='best')
ax[1].legend(["$C^{hom}_{33}/C^{hom}_{11}$ (FEM)", "P&G", "Grimal (Tibia)"], loc=
ax[2].legend(['$C^{hom}_{44}$ (FEM)', 'P&G', 'Grimal (Tibia)','$C^{hom}_{66}$ (FE

#ax[0].grid()
#ax[1].grid()
#ax[2].grid()
# save it!
filename = 'Plots/CellProb_Qfactor'+str(structure)+\
           'm14-3_f6-3_Freq05_Rel_Tibia'
for _ in ['.pdf', '.png']:
    plt.savefig(filename+_, dpi=200, bbox_inches='tight')
# show plot
plt.show()
```



Moreover, just for study of the elastic coefficients, they are plotted here and saved as before in the folder "Plots"

In [13]:
```
plt.figure(figsize=(16, 12))
plt.subplot(2,3,1)
plt.plot(density_tibia, C11_tibia, 'ko')
plt.plot(density_pg, C22_pg, 'rx')
plt.plot(density_sim, data_saved['C22'], 'b^')
plt.title('$C_{11}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.ylabel('[GPa]')
plt.legend(['Grimal (Tibia)', 'P&G', 'FEM'])

plt.subplot(2,3,2)
plt.plot(density_tibia, C33_tibia, 'ko')
plt.plot(density_pg, C33_pg, 'rx')
plt.plot(density_sim, data_saved['C33'],'b^')
plt.title('$C_{33}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.legend(['Grimal (Tibia)', 'P&G', 'FEM'])

plt.subplot(2,3,3)
plt.plot(density_tibia, C66_tibia, 'ko')
plt.plot(density_pg, C66_pg, 'rx')
plt.plot(density_sim, data_saved['C66'],'b^')
plt.title('$C_{66}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.legend(['Grimal (Tibia)', 'P&G', 'FEM'])

#check THIS ONE!
plt.subplot(2,3,4)
plt.plot(density_pg, C12_pg, 'rx')
plt.plot(density_sim, data_saved['C12'],'b^')
plt.title('$C_{12}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.ylabel('[GPa]')
plt.legend(['P&G', 'FEM'])

plt.subplot(2,3,5)
plt.plot(density_tibia, C13_tibia, 'ko')
plt.plot(density_pg, C13_pg, 'rx')
plt.plot(density_sim, data_saved['C23'],'b^')
plt.title('$C_{13}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.ylabel('[GPa]')
plt.legend(['Grimal (Tibia)', 'P&G', 'FEM'])

plt.subplot(2,3,6)
plt.plot(density_tibia, C44_tibia, 'ko')
plt.plot(density_pg, C44_pg, 'rx')
plt.plot(density_sim, data_saved['C55'],'b^')
plt.title('$C_{44}^{hom}$ Coefficient')
plt.xlabel('Density $[g\, cm^{-3}]$')
plt.ylabel('[GPa]')
plt.legend(['Grimal (Tibia)', 'P&G', 'FEM'])

plt.savefig('Plots/Elastic_Coeffs_m14-3_f6-1_freq05.pdf', dpi=200, bbox_inches='t
plt.show()
```



Finally, just for exporting procedures, the elastic and viscous coefficients are exported in .mat format.

The .mat file generated is created and saved on a folder within the directory currently used, called "DataMAT"

In [14]:
```
# Exporting to .mat files
filename_mat = 'DataMAT/Visc_'+str(structure)+'2DPart'+\
               str(part)+'m14-3_f6-3_Freq'+str(omega)+'_Tibia'
data_to_save = {'density_sim': density_sim,
                'frequency': omega,
                'C22': data_saved['C22'],
                'C33': data_saved['C33'],
                'C55': data_saved['C55'],
                'C66': data_saved['C66'],
                'C12': data_saved['C12'],
                'C23': data_saved['C23'],
                'D22': data_saved['D22'],
                'D33': data_saved['D33'],
                'D55': data_saved['D55'],
                'D66': data_saved['D66'],
                'D12': data_saved['D12'],
                'D23': data_saved['D23']}
sio.savemat(filename_mat, data_to_save, appendmat=True)
```

In [ ]: