INTEGRATING ATOMISTIC AND CONTINUUM MODELS WITHIN THE DIGITAL MATERIAL SOFTWARE FRAMEWORK

C.-S. Chen, T. Cretegny, A. J. Dolgert, N. Bailey, C. R. Myers, J. P. Sethna, and A. R. Ingraffea
Cornell Theory Center and Laboratory of Atomistic and Solid State Physics, Cornell University

www.tc.cornell.edu/Research/Multiscale

Supported by NSF/KDI #9873214, "Multiscale Modeling of Defects in Solids"

Knowledge & Distributed Intelligence

DIGITAL MATERIAL

Introduction: Digital Material is an extensible modeling and software infrastructure to support the representation and simulation of material structure and evolution across multiple length and time scales. Such an environment must balance the need for high performance against the need for lightweight prototyping and interrogation. It must be able to integrate a variety of programs and tools into a consistent and seamless framework for multiscale materials simulation. And it must support code reuse across scales through appropriate decomposition of functionality among collections of collaborating objects and modules.

Objectives: Digital Material is intended to function in several roles:

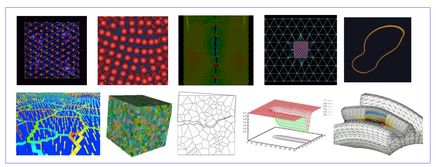
- *as a material representation* supporting descriptions of features across many length scales (and compositions of those features into virtual material samples)
- *as a programming environment* supporting simulation and analysis of material samples at various scales
- *as a problem-solving environment* supporting interactive control and interrogation of simulations, data, etc.

Strategies: The strategies we are pursuing to build this environment are several:

- *the use of a two-level software architecture*, combining low-level numerical kernels written in compiled languages and a high-level interpreted control and integration layer written in the interpreted programming language Python
- *the use of Design Patterns and related object-oriented programming techniques* to decompose simulation functionality among collaborating objects, in order to support flexible program composition
- *the development of object models for the hierarchy of structures arising in materials*, separated into explicitly modeled geometric components and implicitly modeled attributes, so as to support the migration of information across scales.

Python & friends
interactive control & component glue

DFT	MD	QC	Hysteresis	Phase Field
KMC	Texture	Plasticity	Fatigue	Fracture
Parallelization	Visualization		Mesh	IO

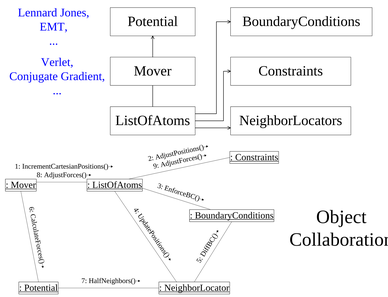


Advantages:

- code sharing and reuse among projects
- flexible coupling and decoupling of computational components
- collaboration from researchers among different disciplines, and with different codes
- direct benefit from world wide development of Python modules
- a set of interoperable components to create a whole that is much greater than the sum of its parts
- software adaptivity in a responsive way to match the volatility of changing software and hardware requirements

MOLECULAR DYNAMICS (MD) FRAMEWORK

Design Philosophy: We develop a molecular dynamics framework with an eye for flexibility and efficiency. We design and implement our numerical kernels using C++ and control and integrate the kernels using Python. The functionalities of our framework are decomposed into 6 major base classes (see class diagram on the right). We identify class interfaces and inheritance hierarchies that are suitable for MD simulations. We develop patterns to organize the interactions among computational objects, and encapsulate those aspects of a program that are likely to change.



Object Collaboration

MD IN ACTION

SWIG
(Simplified Wrapper and Interface Generator)
C++ → Python

Potential

```
class EMTPotential : public Potential
{
public:
    EMTPotential(char *element = "Cu");
    double GetLengthScale() const;
...
```

AtomsInitializers For the Domain

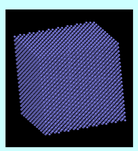
```
class RectangularClusterInitializer : public
ClusterInitializer
{
public:
    RectangularClusterInitializer(double
        rectangleSize[DIMENSION],
        BravaisLatticeWithBasis *bravais);
...
class ClusterInitializer: public AtomsInitializer
{
public:
    ClusterInitializer(BravaisLatticeWithBasis
        *bravais);
    void SetCenter(double
        clusterCenter[DIMENSION]);
    void Create(ListOfAtoms *atoms);
...
```

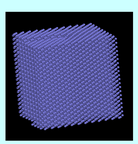
Mover

```
class ConjugateGradientMinimizer: public
Minimizer
{
public:
    ConjugateGradientMinimizer(double tolerance,
        int maximumIterations);
...
class MinimizerAtomsMover : public AtomsMover
{
public:
    MinimizerAtomsMover(Minimizer *minimizer,
        Potential *potential);
    void Move(ListOfAtoms *atoms);
...
```

```
>>>from MD import *   # import the molecular dynamics package
>>>potential = EMTPotential()  # we first choose to use Effective Medium Theory
>>>latticeSpacing = potential.GetLengthScale()*sqrt(2)  # GetLengthScale() returns the interatomic
distance in the lattice at 0 K.
>>>lattice = FCCLattice(latticeSpacing)   # we need an FCC lattice, but we could choose any type of
lattice
# we need a list of atoms, where each atom has a position and a velocity.(->PrimitiveListOfAtoms)
>>>atoms = PrimitiveListOfAtoms()
# we want to initialize the atoms' positions in a cube of size [100,100,100] in FCC configuration
>>>cube = RectangularClusterInitializer([100, 100, 100], lattice)
# if we want to have the faces of the cube with the (111), (211), (011) orientations
>>>lattice.RotateLatticeVectors( array([ [1/sqrt(3), 1/sqrt(3), 1/sqrt(3)], \
                                       [-2/sqrt(6), 1/sqrt(6), 1/sqrt(6)], \
                                       [0        , -1/sqrt(2),1/sqrt(2)]]))
# we can of course change the center of the cube
>>>cube.SetCenter([1., 1., -2.])
# now let's populate the atoms in the cube
>>>cube.Create(atoms)
>>>print "Number of Atoms: ", atoms.GetNumber() # how many atoms in the cube?
>>>plot = RasMol(atoms) # we like to see them and play
# we can easily introduce a screw dislocation (i.e. shift the atoms according to Linear Elasticity)
>>>direction = [0, 0, 1]
>>>burgersVector = [0, 0, latticeSpacing/sqrt(2.)]  # this corresponds to a 1/2(0 1 1) dislocation
>>>center = [0.1, 0.2, 0.3] # avoid cutting an atom
>>>dislocation = DislocationMaker(burgersVector, direction, center, 0.3) # last argument is an
estimation of the Poisson ratio
>>>dislocation.Transform(atoms)
>>>plot.Update()
# now we let the atoms find their equilibrium configuration
>>>cg = ConjugateGradientMinimizer(1.0e-13, 1000) # argument is tolerance and maximum
number of iterations
>>>minimizer = MinimizerAtomsMover(cg, potential)
>>>print 'Initial potential energy in the cube', potential.CalculateEnergy(atoms)
# ah! we forgot the set a neighbor locator
>>>neighborLocator = CellNeighborList(potential.GetCutoffDistance(), \
        0.05*potential.GetCutoffDistance()) # the 2nd parameter is the drift for the neighbor list
>>>atoms.SetNeighborLocator(neighborLocator)
>>>print 'Initial potential energy in the cube', potential.CalculateEnergy(atoms)
>>>minimizer.Move(atoms)
>>>print 'Potential energy after relaxation', potential.CalculateEnergy(atoms)
>>>plot.Update()
>>>writer = NetCDFWriter('tmp.cdf')  # create a file to save our atoms in NetCDF format
>>>writer.Put(atoms, 'myAtoms')   # put the atoms in the file, under the name myAtoms
# etc.
```

Number of Atoms: 12792

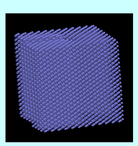




RuntimeError:
NoNeighborLocator::ReturnTrue:
A function called Neighbors. Please set a neighborLocator

Initial potential energy in the cube
1220.53

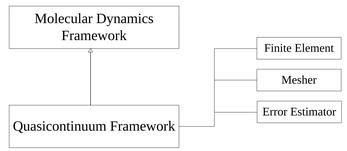
Potential energy after relaxation
1193.38



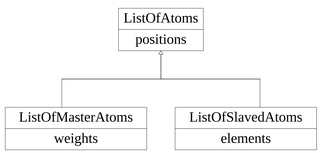
QUASICONTINUUM (QC) FRAMEWORK

Introduction: Quasicontinuum is a mixed atomistic-continuum approach to blend the atomistic realism with finite element interpolation. Our quasicontinuum framework is a natural outgrowth of our molecular dynamics framework; we have quite more the atomistic point of view than the finite element's. Thus, our 6 major base classes designed in molecular dynamics are heavily reused and extended in our quasicontinuum code.

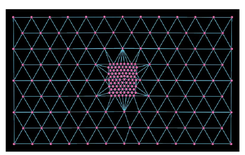
Our quasicontinuum framework is still under intensive development and testing. Results presented here are preliminary.



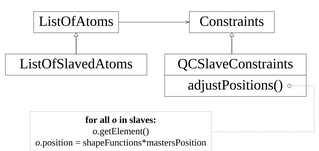
Atom Representation: We assume that in the solid only a few atoms are representatives, whereas the other degrees of freedom are somewhat constrained, or slaved to the position of these representative atoms. We call the former master atoms, the latter slave atoms and we have a container class for both of them: *ListOfMasterAtoms* and *ListOfSlavedAtoms*. Both classes are derived from *ListOfAtoms* from our molecular dynamics framework.



Meshing: We would like to sample defects down to the atomic scale, while use "coarse-grained" atoms to represent the homogeneous regions. We triangulate the representative atoms, which defines a set of finite elements. The nodes of these elements are initialized with slaved atoms.

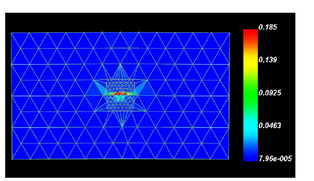


Energy and Force Calculation: We compute the total potential energy as the sum of the weighted potential from each master atom. To find the equilibrium configuration we compute the molecular dynamics forces on the masters due to the other masters and the slaves using conjugate gradient algorithm. When the master atoms move, the constraint on the slave atoms increments their position according to the shape functions of the element they belong to.



Error Estimation and Adaptive Meshing: To optimally represent the atoms, we select our master atoms guided by *a posteriori* error estimation from energy and force calculation. Our re-meshing scheme for refinement and coarsening is to:

- perform local deletion of elements (a.k.a. local cluster),
- reconstruct underlying elementwise lattice system, and
- re-mesh the cluster based on new internal points (derived from error indicators) and boundary connectivity of the cluster.



Reference: V.B. Shenoy, R. Miller, E.B. Tadmor, D. Rodney, R. Phillips, M. Ortiz (1999) "An adaptive finite element approach to atomic-scale mechanics-the quasicontinuum method" Journal of Mechanics and Physics of Solids, 47, pp. 611-642.