KIM API alpha version

(Knowledgebase of Interatomic Models Application Program Interface alpha version)

Prepared by: Valeriu Smirichinski and Ryan S. Elliott

February , 2011

This document describes how KIM **Tests** and **Models** written in different languages work together. A unified interface, tuned for the specific needs of atomistic simulations, is presented. This interface is based on the concept of "descriptor files". A descriptor file specifies all variables and methods required for communication between a particular **Model** and a **Test**. A "KIM API object" is created, based on the descriptor files, that holds all variable/data and method pointers needed for **Test**/**Model** interaction. A complete set of KIM API service routines are available for accessing the various pointers in the KIM API object.

Contents

1

KIM API concept and implementation:

1. The KIM repository contains **Models** and **Tests**
2. The most challenging technical requirement is the need for multi-language support
3. The KIM API is based on exchanging pointers to data and methods
4. How can a **Test** know what type of input/output data is required by a **Model**?
   We have solved this problem by introducing the KIM API descriptor file
5. Each line in the descriptor file defines a variable (scalar, array, method, etc.) and its properties
6. **Test/Model** coupling: The **Model's** initialization routine stores a pointer to the "compute" routine in the KIM_API object
7. An example of a simple **Test** (written in C) using the KIM API
8. An example of a simple **Model** (written in FORTRAN 90) using the KIM API
9. The KIM service routines
10. KIM installation, compilation, linking and troubleshooting

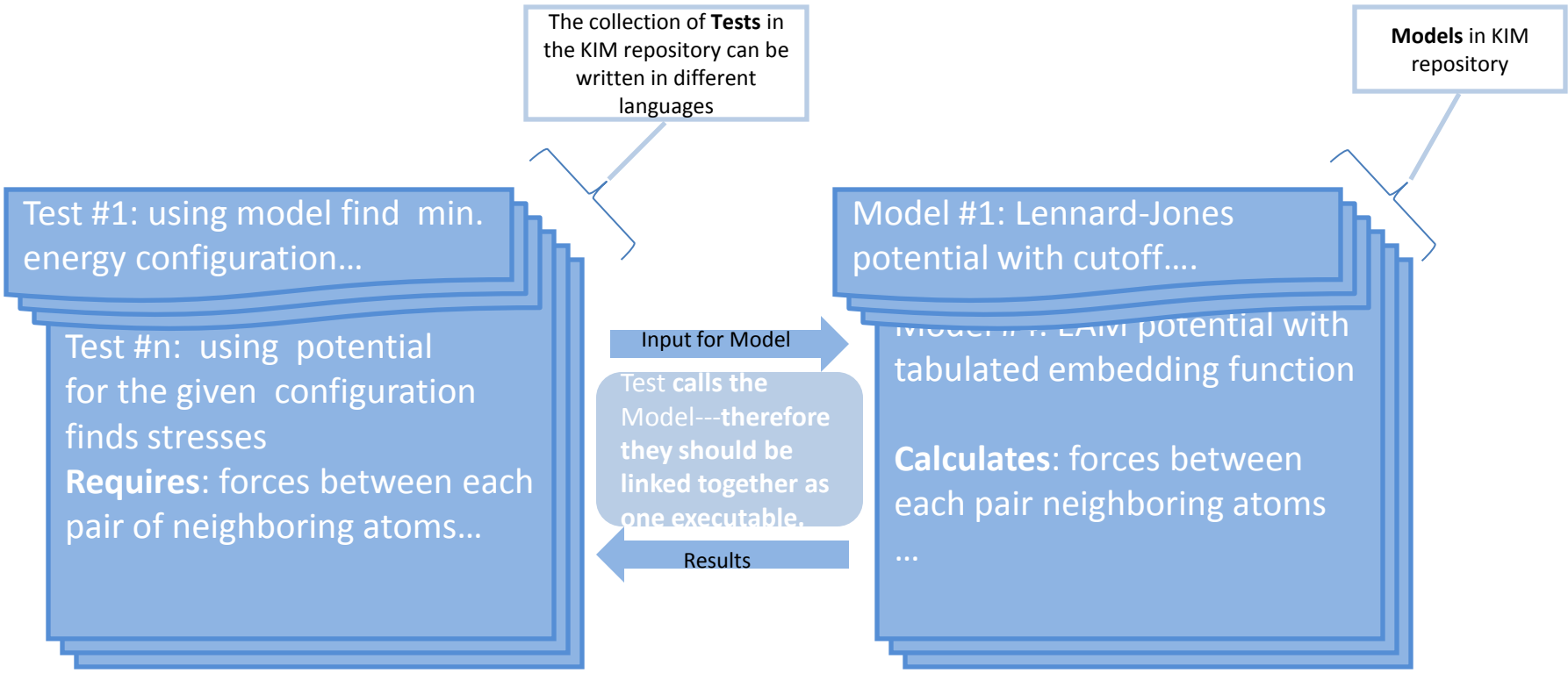
2

Appendix

1. Every variable that needs to be communicated between **Tests** and **Models** must be in the descriptor file
2. The KIM API directory structure
3. Six **Test-Model** examples are available in this KIM API version: (C-C, C-C++, C-F90, F90-C, F90-C++, F90-F90)
4. The KIM API object is an array of Base data elements.
   Each Base data element can hold a pointer to any relevant data (scalar, array, method, etc.)
5. Three Lennard-Jones models (in the C, C++ and F90) are available in this release. Each uses a neighbor-list iterator (written in F90) as part of its internal calculation loop.

KIM API concept

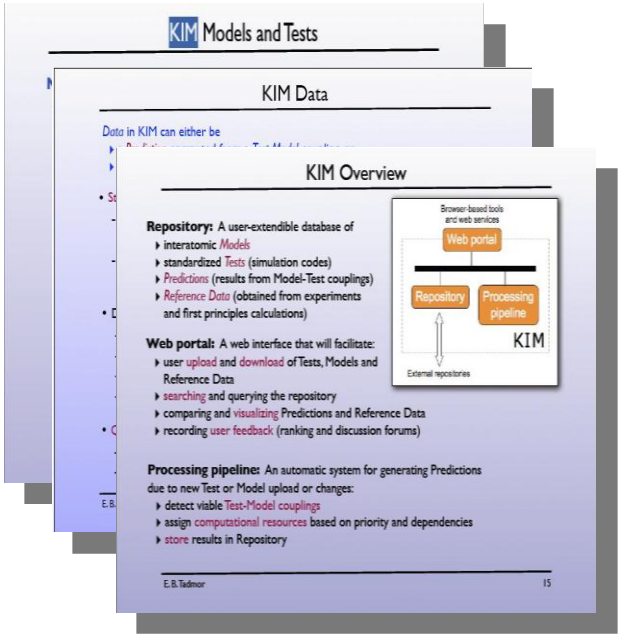
The KIM repository contains Models and Tests



Users and developers will be able to download **Tests** and **Models** , then compile, link and run the resulting programs to produce new results.

The most challenging technical requirement is the need for multi-language support

KIM framework



Processing pipeline: an automatic system for generating predictions when Tests or Models are uploaded or changed.

Requirements:

- Multilanguage support (C, C++, FORTRAN 90, Python …)
- A variety of data structure need to be accommodated: scalars, multidimensional arrays, variable size arrays, etc..
- Speed & performance are very important
- Standardized API, version tracking, etc…

Processing pipeline: sequence of actions

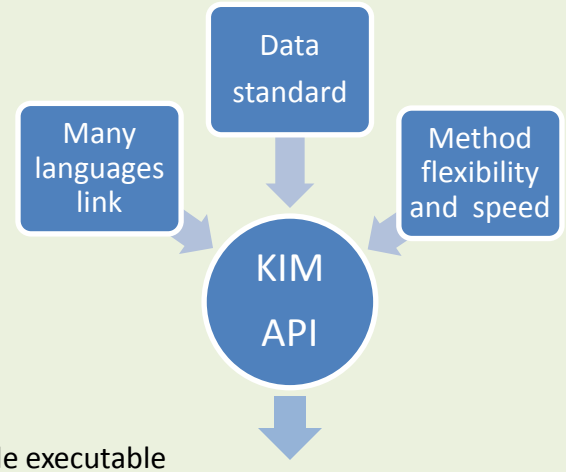
- detect a viable **Model/Test** coupling
- **build (compile and link)** Tests **against** Model
- **run probe-tests**
- assign computational resources
- run full-scale **Test** against **Model**
- analyze results …
- store results in the repository

Need a simple interface : ideally just one argument per call

Source: KIM kickoff presentation

The KIM API is based on exchanging pointers to data and methods

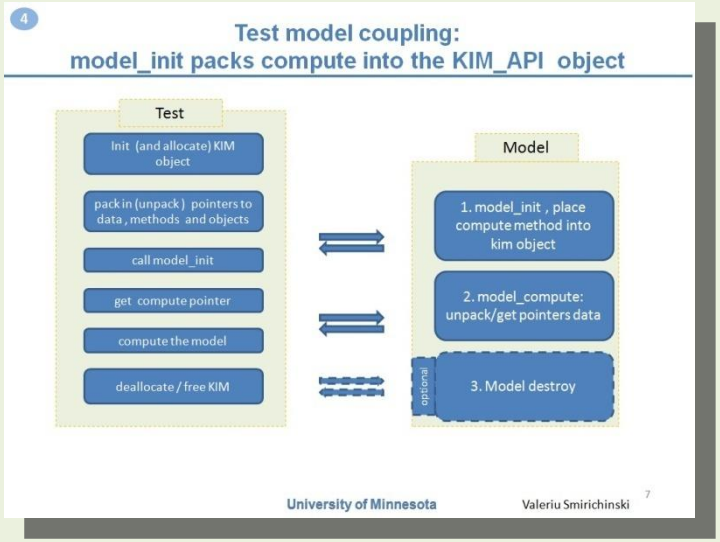
Concept



Data standard should accommodate every possible data set required for the model

Schematic of implementation

1. Data and method pointers are packed in one object. The Interface consists of exchanging one pointer to the KIM API object between a **Test** and a **Model**
2. All languages naturally support pointers:
   - FORTRAN (cray or 2003 standard)
   - C/C++
   - Java
   - Python



How can a Test know what type of input/output data is required by a Model? We have solved this problem by introducing the KIM API descriptor file

Sample_01_lj_cutoff.kim

```
MODEL_NAME        := model_val01
SystemOfUnitsFix  := fixed    #can work only with units system defined bellow

MODEL_INPUT:
# Name                Type            Unit        SystemU/Scale       Shape                 requirements
numberOfAtoms         integer*8       none        none                []

coordinates           real*8          length      standard            [numberOfAtoms,3]

compute               method          none        none                []

neighIterator         method          none        none                []

neighObject           pointer         none        none                []

cutoff                real*8          length      standard            []

MODEL_OUTPUT:
# Name                Type            Unit        SystemU/Scale       Shape                 requirements
energy                real*8          energy      standard            []

forces                real*8          force       standard            [numberOfAtoms,3]
```

Each line in the descriptor file describes a variable and its properties

Sample_model.kim

All characters after a '#' are ignored (a comment field)

```
MODEL_NAME        := model_val01
SystemOfUnitsFix  := fixed     #can work only with units system defined bellow
….
compute             method      none       none                      []

MODEL_OUTPUT:
# Name              Type        Unit       SystemU/Scale             Shape                requirements
energy              real*8      energy     standard                   []

energyPerAtom       real*8      energy     standard                  [numberOfAtoms]      optional
….
```

Method means a subroutine or function pointer

The name of a variable is its "key word". By using key words, the KIM service routines can pack/unpack data pointers from the KIM API object. Key words will be standardized as part of the KIM API.

Type of data in computer representation

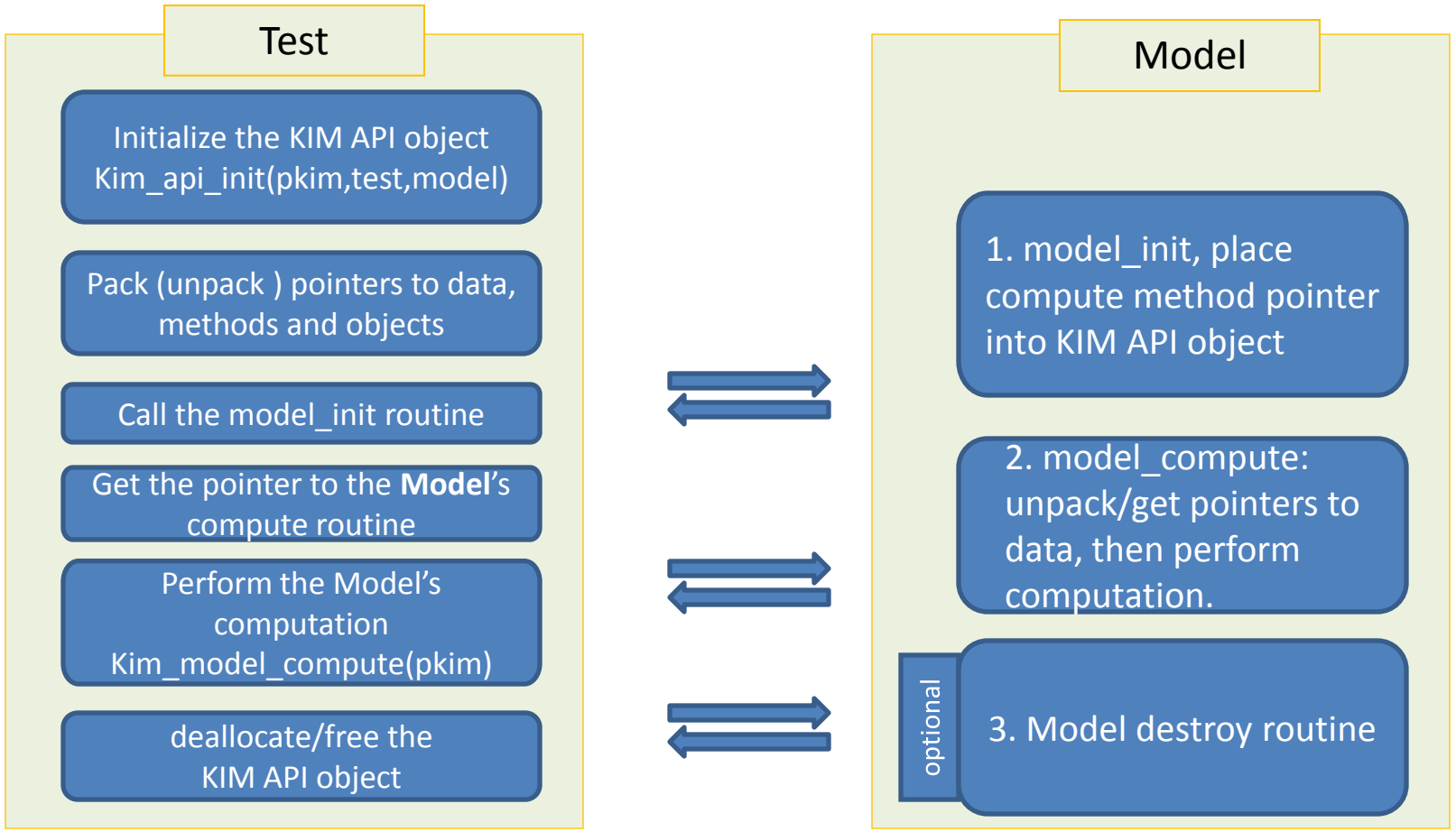
Physical dimensions:

System of units: standard, SI

The shape of a variable describes its array properties . It specifies the number and size (range) of indices. For example, [] means a scalar (zero-dimensional array), [NumberOfAtoms] means a one-dimensional array and [N, 3] means a two-dimensional array of size N x 3.

The "requirements" field is only used in **Model** descriptor files. An empty field indicates that the variable is required. A value of "optional" indicates that the associated data will be computed only if the variable is in the **Test**'s descriptor file and if the **Test** explicitly requests it.

Test/Model coupling: The Model's initialization routine stores a pointer to the "compute" routine in the KIM_API object



Pointer to KIM API object is the only argument communicated between **Tests** and **Models**

An example of a simple Test (written in C) using the KIM API (page 1)

lj_test_f.c

```
#include <stdlib.h>
#include <stdio.h>
#include "KIMserviceC.h"

/* Define prototypes for neighbor list handeling */
void neighobj_allocate_(intptr_t **);
void neighobj_deallocate_(intptr_t **);
void neighborscalculate_(intptr_t **,double**,int*,double*);
intptr_t * get_neigh_iterator_();

/* Define prototypes for model initiation routines */
void sample_01_lj_cutoff_init_(intptr_t **);

int main(){
        /* KIM API potiner declarations */
        intptr_t * pkim;
        double * penergy;
        double * pcutoff;
        intptr_t * numberOfAtoms;
        double * x;
        double * f;
        intptr_t * neighObject;
    /* Defines function pointer to model compute routine */
        void (*model_compute)(intptr_t **);

        /* test  and model name */
        char testname[80] ="Sample_01_compute_example_c";
       /*in the future will be passed as argument for the test */
        char modelname[80] ="Sample_01_lj_cutoff";
```

```
/* Local declarations */
  char infile[80] = "./data_output/dumpval10.xyz";
  double cutofeps;
  int i,n,id,ntypes;
  float t0,t1,t2;
  FILE*fl;

/* Initialized KIM API object */
  KIM_API_init((void *)&pkim, testname ,modelname);

/* open input atomic configuration file */
  fl=fopen(&infile[0],"r");
/* read number of atoms in configuration */
  fscanf(fl,"%d",&n);
  ntypes = 1; /* one atomic species only */

/*Allocate memory and associated it with KIM API object*/
KIM_API_allocate((void*)pkim,(intptr_t)n,ntypes);
```

KIM_API_allocate will allocate memory for all arrays pointers stored in KIM API object

An example of a simple Test (written in C) using the KIM API (page 2)

lj_test_f.c

```
/*Make local pointers point to allocated memory(in KIM API object)*/
 penergy=(double *)KIM_API_get_data((void *)pkim,"energy");
 pcutoff=(double *) KIM_API_get_data((void *)pkim,"cutoff");
 numberOfAtoms = (intptr_t
                 *)KIM_API_get_data(pkim,"numberOfAtoms");
*numberOfAtoms = (intptr_t)n;
  x  = (double *)KIM_API_get_data(pkim,"coordinates");
  f  = (double *)KIM_API_get_data(pkim,"forces");

  /* Read in the atomic positions for all atoms */
   for(i=0; i<n;i++){
          fscanf(fl,"%d %f %f %f",&id,&t0,&t1,&t2);
         *(x+i*3+0)=t0; *(x+i*3+1)=t1; *(x+i*3+2)=t2;
   }
   /* close input file */
   fclose(fl);

   /* Setup neighbor list */
   neighobj_allocate_(&neighObject);
   /* Set calculation parameters */
   *pcutoff=1.8;
   cutofeps=2.1;

   /* calculate neighbor list for the configuration */
   neighborscalculate_(&neighObject,&x,&n,&cutofeps);
 /* Inform KIM API object about neighbor list iterator and object */
  KIM_API_set_data(pkim,"neighIterator",1,(void*)
                     get_neigh_iterator_());
  KIM_API_set_data(pkim,"neighObject",1,(void*) neighObject);
```

```
/* All setup finished -- ready to compute */

/*Call Model compute routine,e.g.,compute energy/force*/
  KIM_API_model_compute(pkim);

  /* output KIM API object to screen (optional) */
  KIM_API_print(pkim);

  /* Print out energy and list of forces */
  printf("\n\n============================\n");
  printf("Energy = %10.5f\n",*penergy);
  printf("Forces:\n");
  for (i=0;i<n;i++) {
      printf("%10.5f, %10.5f, %10.5f \n",
             f[i*3+0],f[i*3+1],f[i*3+2]);
  }

  /* clean up */
  neighobj_deallocate_(&neighObject);
  KIM_API_free(&pkim);
}
```

An example of a simple Model (written in FORTRAN 90) using the KIM API (page 1)

LJ_MOD.F90

```
module lj_test_mod
       use  KIMservice
       implicit none
      !defining the structure that holds variables for calculation
       type lj_test_object
          integer :: numberatoms
          ! position, forces and saved position
          real*8,pointer :: x(:,:), f(:,:)
          ! parameters of lj potential
          real*8 :: a= 0.0002441,b=0.03125,cutof=2.3
          real*8 :: energy ! energy
       end type lj_test_object

       !defining cray pointer to neighbor iterator external
       !neighbors iterate
       pointer (piterator,neighborsiterate)

       type (lj_test_object) :: lj_obj
       SAVE :: lj_obj
contains
        ! actual initialization routine
    subroutine lj_init(pkim )
          use KIMservice
          implicit none
          ! KIM API related declaration
          integer(kind=kim_intptr) :: kim;
          pointer(pkim,kim)
          real*8 :: xstub(3,1);
          pointer(px,xstub)  ! cray pointer to position
          real*8 :: fstub(3,1);
          pointer(pf,fstub)  ! cray pointer to forces
```

```
integer(kind=kim_intptr) ::sz;
integer(kind=8):: numatoms;
pointer(patoms,numatoms) ! pointer number of atoms
real*8::cutoff; pointer(pcutoff,cutoff)
real*8::skin; pointer(pskin,skin)
real*8::energy; pointer(penergy,energy)

!getting pointers data from kim
patoms = kim_api_get_data_f(pkim,"numberOfAtoms") ;
lj_obj%numberatoms=numatoms

px=kim_api_get_data_f(pkim,"coordinates");
call toRealArrayWithDescriptor2d(xstub,lj_obj%x,
                                 3,lj_obj%numberatoms)

pf=kim_api_get_data_f(pkim,"forces");
call toRealArrayWithDescriptor2d(fstub,lj_obj%f,
                                 3,lj_obj%numberatoms)
pcutoff = kim_api_get_data_f(pkim,"cutoff");
lj_obj%cutof = cutoff;
!getting pointer to neighbor iterator from KIM API object
! get pointer to iterator
piterator=kim_api_get_data_f(pkim,"neighIterator")

!setting pointer to compute method
sz=1
if(kim_api_set_data_f(pkim,"compute",sz,
                      loc(lj_calculate)).ne.1) then
      stop ' compute not found in kim'
 end if
end subroutine lj_init
```

An example of a simple Model (written in FORTRAN 90) using the KIM API (page 2)

LJ_MOD.F90

```
subroutine lj_calculate(pkim)
! compute routine with KIM interface
   implicit none
   integer(kind=kim_intptr) :: kim; pointer(pkim,kim)
   call lj_calculate2(pkim,lj_obj%x,lj_obj%f)
end subroutine lj_calculate

subroutine lj_calculate2(pkim,x,f) ! actual compute routine
   use KIMservice
   implicit none

   !KIM related declaration
   integer(kind=kim_intptr) :: kim; pointer(pkim,kim)
   real*8,pointer,dimension(:,:) :: x,f
   real*8 :: vij,dvmr,v,sumv,cutof,cut2,energycutof
   integer :: i,j,jj,numnei=0;      real*8 :: r2,dv;
   real*8,dimension(3):: xi,xj,dx,fij
   real*8::energy; pointer(penergy,energy)
   integer(kind=kim_intptr) :: nei_obj;
   pointer(pnei_obj,nei_obj) !pointer to neighborlist object
   !local declaration
   integer :: nnn(512)
   integer:: nei1atom(1); pointer (pnei1atom,nei1atom)
   integer restart;
   nnn=0
   pnei1atom = loc(nnn)
   !get pointer to energy
   penergy = kim_api_get_data_f(pkim,"energy")
   ! get pointer to neighbor list object
   pnei_obj=kim_api_get_data_f(pkim,"neighObject")
   cutof = lj_obj%cutof    !get pointer to cutoff
```

```
sumv=0.0;
f(:,:)= 0.0;
numnei = 0
cut2 = cutof*cutof
! store energy of LJ at rcut in 'energycutof'
!to be used for shifting LJ so energy is zero at cutoff
call  ljpotr(cut2,energycutof,dvmr)

restart = 0; !reset nei. iterator to beginning
call neighborsiterate (pnei_obj,pnei1atom,
                                    numnei,restart)
restart=1   !increment flag for neighbor iterator
do while (numnei .ge. 0)
   !increment iterator
   call neighborsiterate (pnei_obj,pnei1atom,
                                      numnei,restart)
   i = nei1atom(2)
   xi = x(:,i)
   do jj=3, nei1atom(1)
       j=nei1atom(jj)
       xj = x(:,j)
       dx = xi-xj
       r2=dx(1)*dx(1) + dx(2)*dx(2) + dx(3)*dx(3)
       if (r2.le.cut2) then
           call ljpotr(r2,vij,dvmr)
           sumv = sumv + vij-energycutof
           f(:,i) =f(:,i) - dvmr*dx
           f(:,j) =f(:,j) + dvmr*dx
       end if
   end do
end do
```

An example of a simple Model (written in FORTRAN 90) using the KIM API (page 3)

LJ_MOD.F90

```
       v=sumv
       lj_obj%energy = v
       energy = v !set energy in KIM API object
       !forces are already stored in KIM API object
       return
       contains
       !Computational core of LJ potential
       subroutine ljpotr(rr,v,dvmr)
            implicit none
            real*8 rr,v,dvmr,       a,b,r1,r2,rm6,rm8
            a=lj_obj%a;
            b=lj_obj%b; ! LJ parameters
            if (rr.lt.0.000000000000001) stop 'rr is zero'
            r2=rr;
            rm6=1.0/(r2*r2*r2);
            rm8=rm6/r2
            v=(a*rm6 - b)*rm6;
            dvmr = 6.0*rm8*(-2*rm6*a + b)
        end subroutine ljpotr
   end subroutine lj_calculate2
end module lj_test_mod

!  Model Initiation routine (it calls actual initialization
!  routine in the module lj_test_mod)
subroutine sample_01_lj_cutoff_init(pkim)
       use lj_test_mod
       implicit none
       integer(kind=kim_intptr) :: kim; pointer(pkim,kim)
       call lj_init(pkim)
end subroutine sample_01_lj_cutoff_init
```

Also, examples in C and C++ are available in the directories:
MODELS/Sample_01_lj_cutoff_c
MODELS/Sample_01_lj_cutoff_cpp

Initialization of KIM API object, packing and unpacking of data-pointers can be done through the KIM service routines

KIMserviceC.h

```
#include <stdint.h>
#ifdef __cplusplus
extern "C" {
#endif
//global methods
int KIM_API_init(void * kimmdl, char * testname, char *mdlname);

void KIM_API_allocate(void *kimmdl, intptr_t natoms, int ntypes);

void KIM_API_free(void *kimmdl);

void KIM_API_print(void *kimmdl);

void KIM_API_model_compute(void * kimmdl);
…
//element access methods
int  KIM_API_set_data(void *kimmdl,char *nm,  intptr_t size, void *dt);

void * KIM_API_get_data(void *kimmdl,char *nm);

Intptr_t  KIM_API_get_size(void *kimmdl,char *nm);

Intptr_t KIM_API_get_rank_shape(void *kimmdl,char *nm, int * shape);

…
```

Initialization is done by analyzing test and model configuration files

One can use optional KIM service allocating and deallocating standard variable and data

Call model compute routine by address stored in KIM API object

Directly place data pointer into the KIM API object

Get size or rank and shape of the data (array)

Description of all KIM API service routines are located in the file: **KIM_API/ KIMserviceDescription.txt**

KIM installation, compilation, linking and troubleshooting

Instructions for installing, compiling and linking KIM:

1. In the desired directory, execute the command: 'tar xvfz KIMdevelAlphaFeb11.tgz'
2. Change directory to KIMdevel/KIM_API and, in the file Include.mk, edit the line:

   KIM_DIR =$(HOME)/KIMdevel/

   so that it points to the correct path where the KIM files are located.
3. By default all make files use the GNU compilers for 64 bit linux. In order to use the Intel compiler, define the environment variable INTEL (bash:export INTEL="yes"). For using a 32 bit machine, define the environment variable SYSTEM32 (bash:export SYSTEM32="yes").
4. Change to the KIMdevel directory and execute the commands:

   'make clean'

   'make'

   This will compile the KIM API and example **Tests** and **Models**.
5. Each **Test** (and **Model**) has its own make file for compiling and linking. If changes are made to the code, perform the second command of step (4) again.

Troubleshooting: By default, the make system uses the **g++** compiler with the **–lgfortran** option to link compiled FORTRAN executables. If (for some versions of linux or GNU compilers) linking of FORTRAN executables with **g++** compiler doesn't work, try **gfortran** with the **–lstdc++** option. See lines 52-55 in Include.mk file.

GNU compilers must be version 4.4.1 or up, Intel compiler must be version 11.1 or up.

Appendix

Every variable that needs to be communicated between tests and models must be in the descriptor file

Each **Test** has its own descriptor file that describes the data it can supply to the **Model** and what data it expects the **Model** to compute. There are no optional variables in a **Test**'s descriptor file ("the test knows, a priori, what to compute").

Each **Model** has its own descriptor file that describes the data it needs to perform its computations and what results it can compute. Some of the variables/methods can be identified as optional. Optional variables/methods are ones that the **Test** does not have to provide or are results that the **Model** will only compute if the **Test** explicitly requests it.

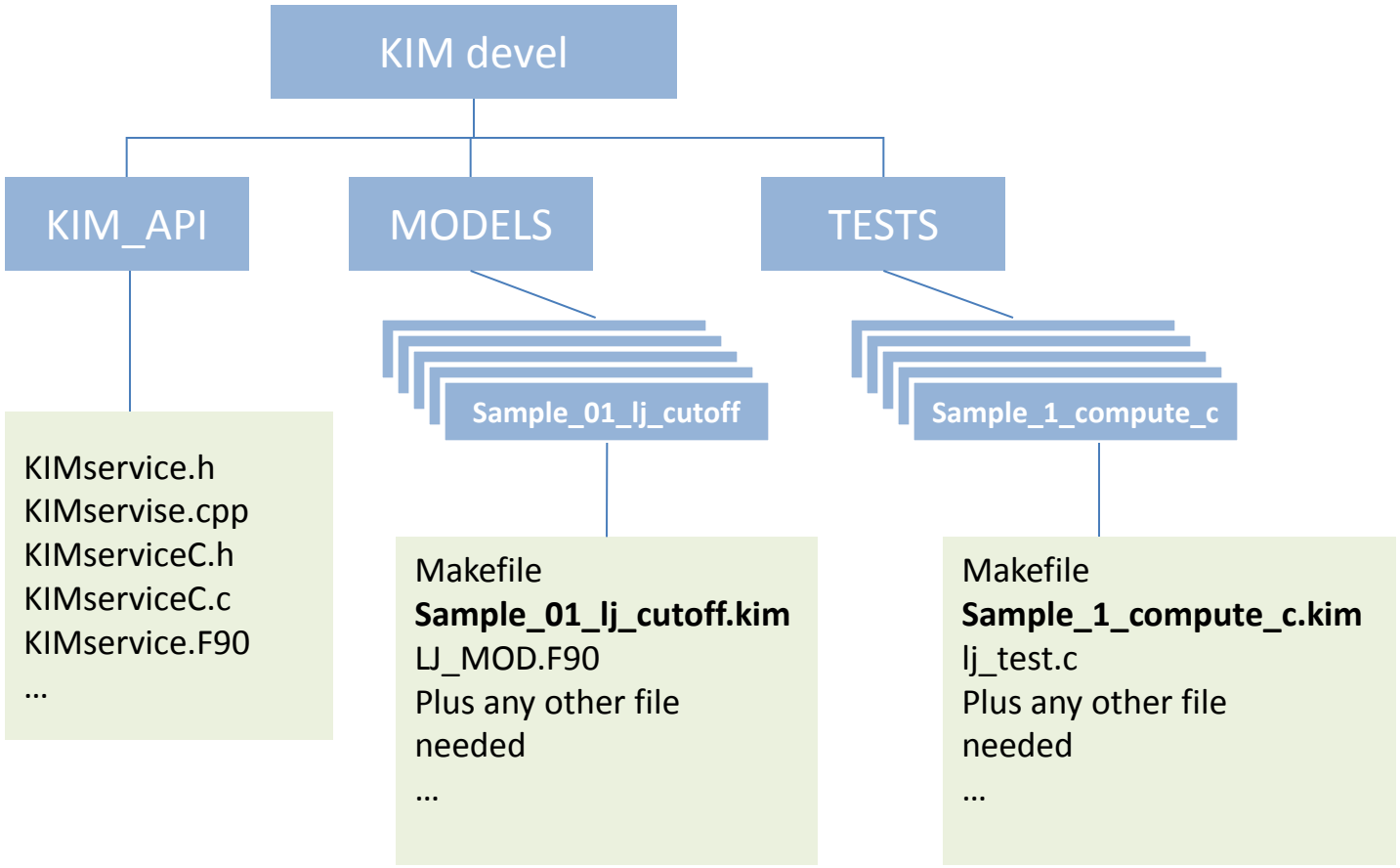
KIM service routines (such as kim_api_init_) use both **Test** and **Model** descriptor files to:

- Check if the **Model** and **Test** match, also check if their descriptor files conform to the KIM API standard
- If they do -- create a KIM API object to store all variables described in the **Model**'s descriptor file
- Mark each optional variable that is not used by the **Test** "uncompute" (i.e., do not compute)

Other service routines are used to:

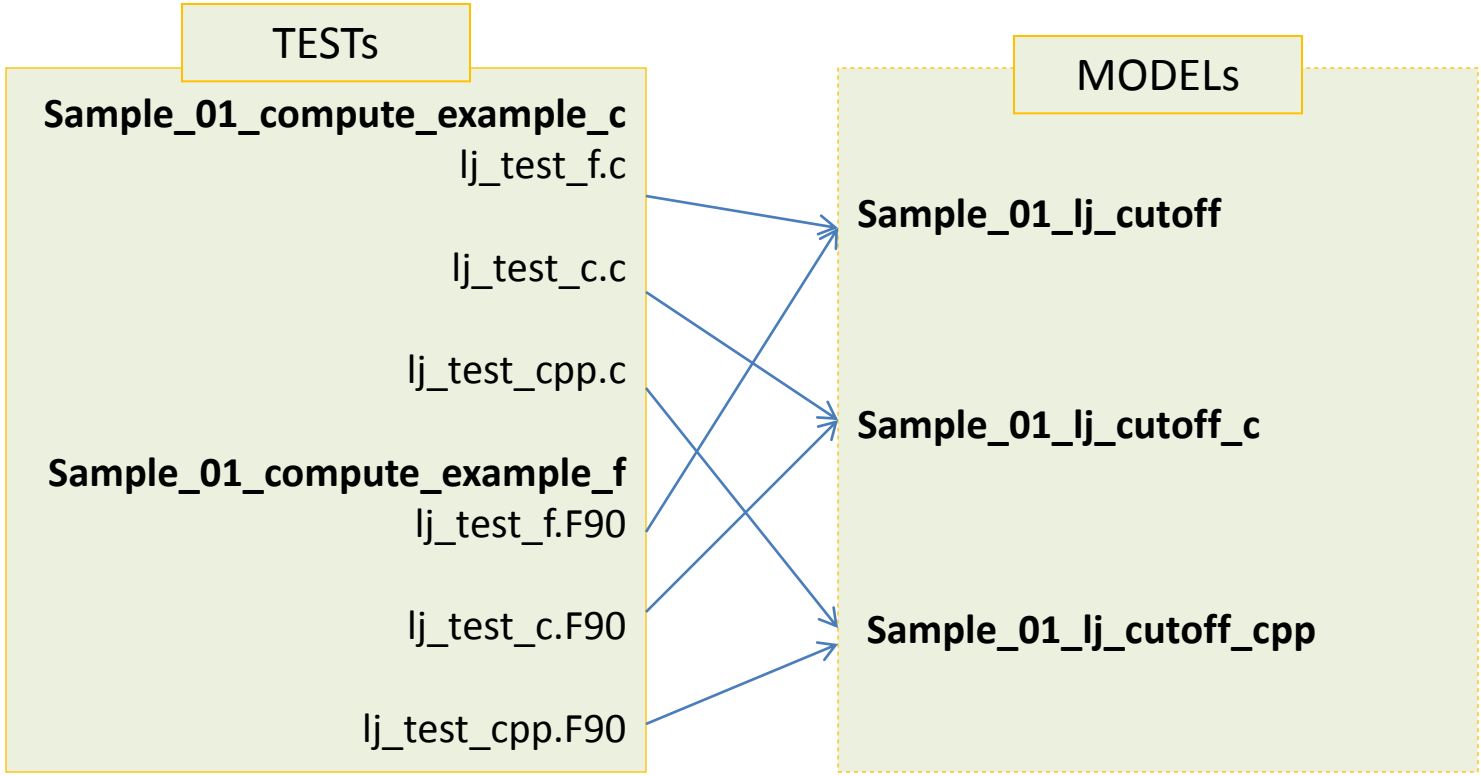
- Pack (unpack) variable or method pointers into (from) the KIM API object (e.g., kim_api_set_data, kim_api_get_data, etc.)
- Check if the "compute flag" is set to "compute" for a variable in the KIM_API obejct (kim_api_isit_compute)
- Execute the Model's compute method (kim_api_model_compute)
- etc…

KIM API directory structure



Each **Test** and **Model** has its own descriptor file

Six Test-Model examples are available in this KIM API version: (C-C, C-C++, C-F90, F90-C, F90-C++, F90-F90)



KIM API object is an array of Base data elements. Each Base data element can hold a pointer to any relevant data

Number of fields is fixed to 9

Base data:

- Can hold any type of array: real, integer, pointer…
- Stores enough information for a complete description of the data

pointer to contiguous array of data of size "size"

size of data in terms of underlying elements

auxiliary array: very useful for 2d arrays (variable and fixed dimensions)

name contains description information like "coordinates", "velocities", "forces", etc…

type tells the type for elements of array: "real", "real*8"integer", "integer*8", pointer

rank and shape:
- rank is the number of indices for the array: for 2D it is 2, for 3D it is 3 etc…
- shape is an integer array of size rank and holds the size (range) of each index.

Flag contains additional information (all fields are integers)

peratom
freeable
pointerchanged
ID

Every field in the Base data structure is a pointer or "pointer size" integer.

Base data type can be used to store all needed data for **Tests** and **Models**

2

Three Lennard-Jones models (in the C, C++ and F90 languages) are available in this release. Each uses a neighbor-list iterator (written in F90) for its internal calculation loop.

MODELs/Sample_01_lj_cutoff_c/LJ_sample1.f90

```
void (*nei_iterator)(void *,int **,int *,int *); //prototype for iterator
(*nei_iterator)(&neighObject,&n1atom,&numnei,&restart);
```

MODELs/Sample_01_lj_cutoff_cpp/ LJ_sample1.cpp

```
typedef void (*NEI_Iterator)(void *,int **,int *, int *);
NEI_Iterator nei_iterator;
(*nei_iterator)(&neighObject,&n1atom,&numnei,&restart);
```

MODELs/Sample_01_lj_cutoff/ LJ_mod.f90

```
interface
    subroutine neighborsiterate(pneiobj,pnei1atom,numnei,restart)
        integer*8 ::pneiobj,pnei1atom
        integer :: numnei,restart! if restart = 0 set iterator to start
                                 ! if restart != 0 proceed next
        end subroutine neighborsiterate
  end interface
  pointer (piterator,neighborsiterate)
  call neighborsiterate (pnei_obj,pnei1atom,numnei,restart)
```

Neighbor -list object – supplied by the **Test**

-1 when reached the end of neighbor list

Array of integer : integer nei1atom; pointer(pnei1atom,nei1atom)
nei1atom(1) -- size of all elements in the array
nei1atom(2) -- atom id and
nei1atom(3:size) -- it's neighbors

The neighbor-list iterator and the neighbor-list object are supplied by the **Test**. If the neighbor-list iterator is called with restart=0, then it resets its internal state to the beginning of the list.
When restart is not zero, it returns the id of the next atom and all of its neighbor atom ids.
In the examples it is assumed that the neighbor-list has a short form (i.e., only atoms with id i<j are included in the neighbor list. This avoids "multiple counting").