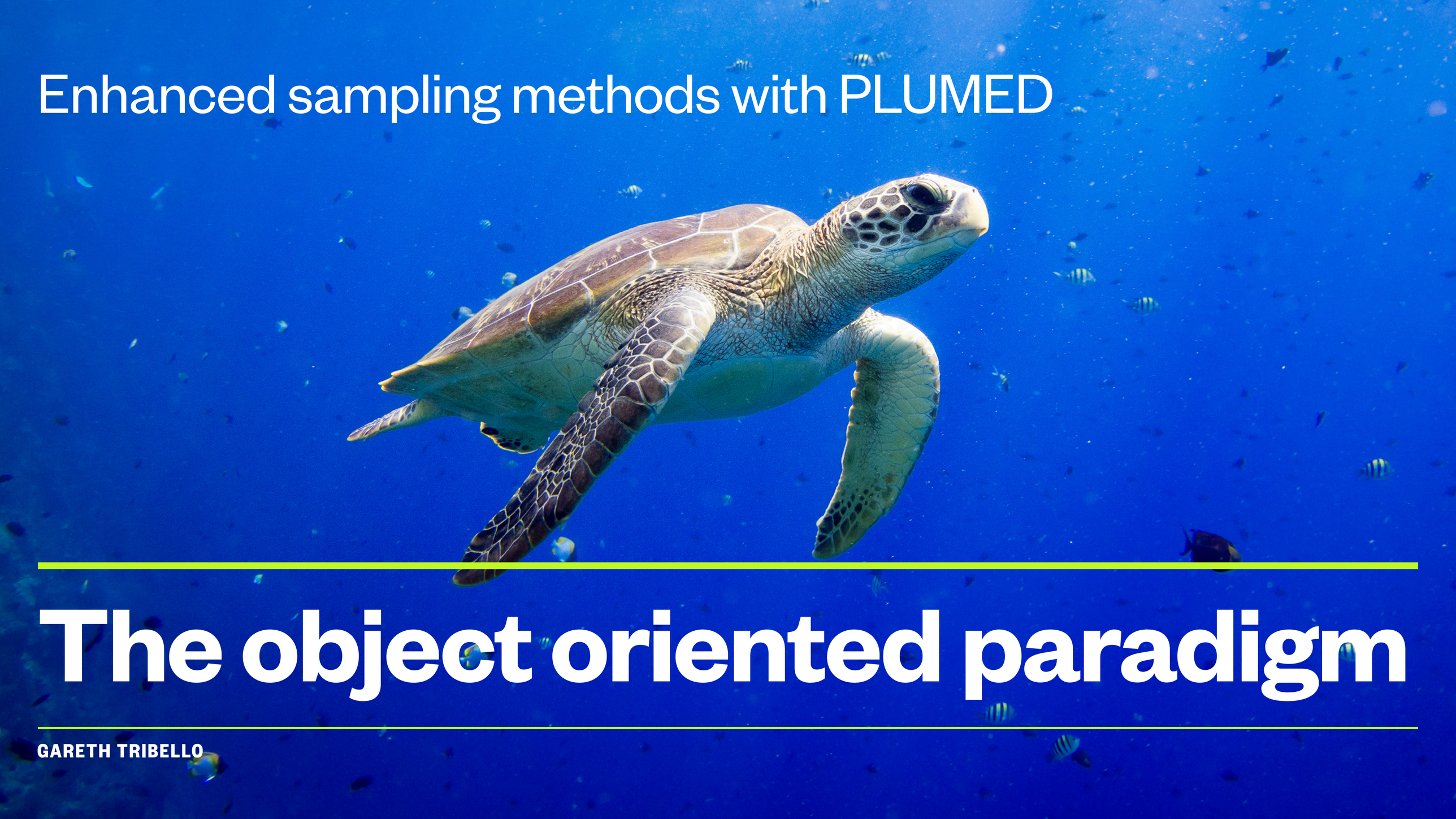
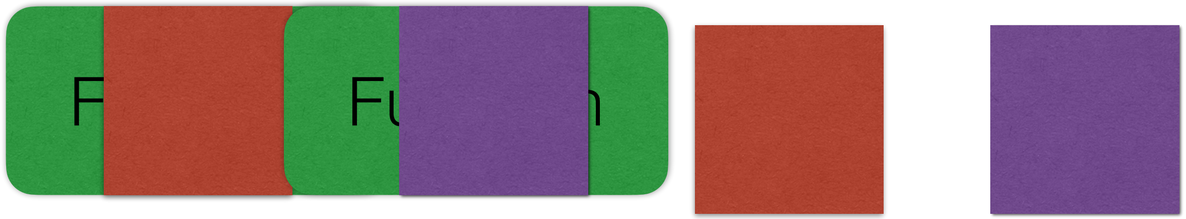
Enhanced sampling methods with PLUMED



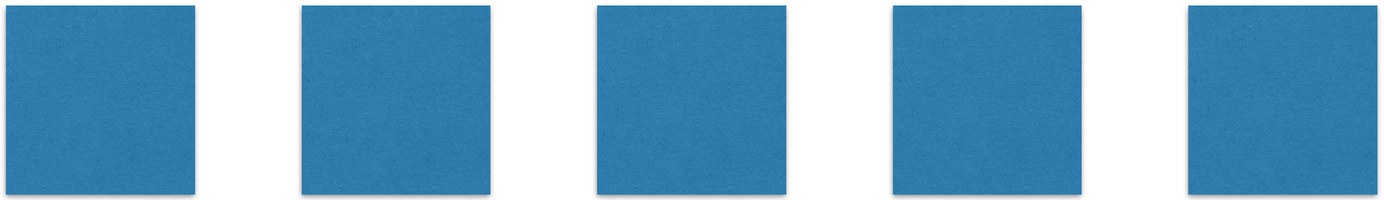
The object oriented paradigm

GARETH TRIBELLO

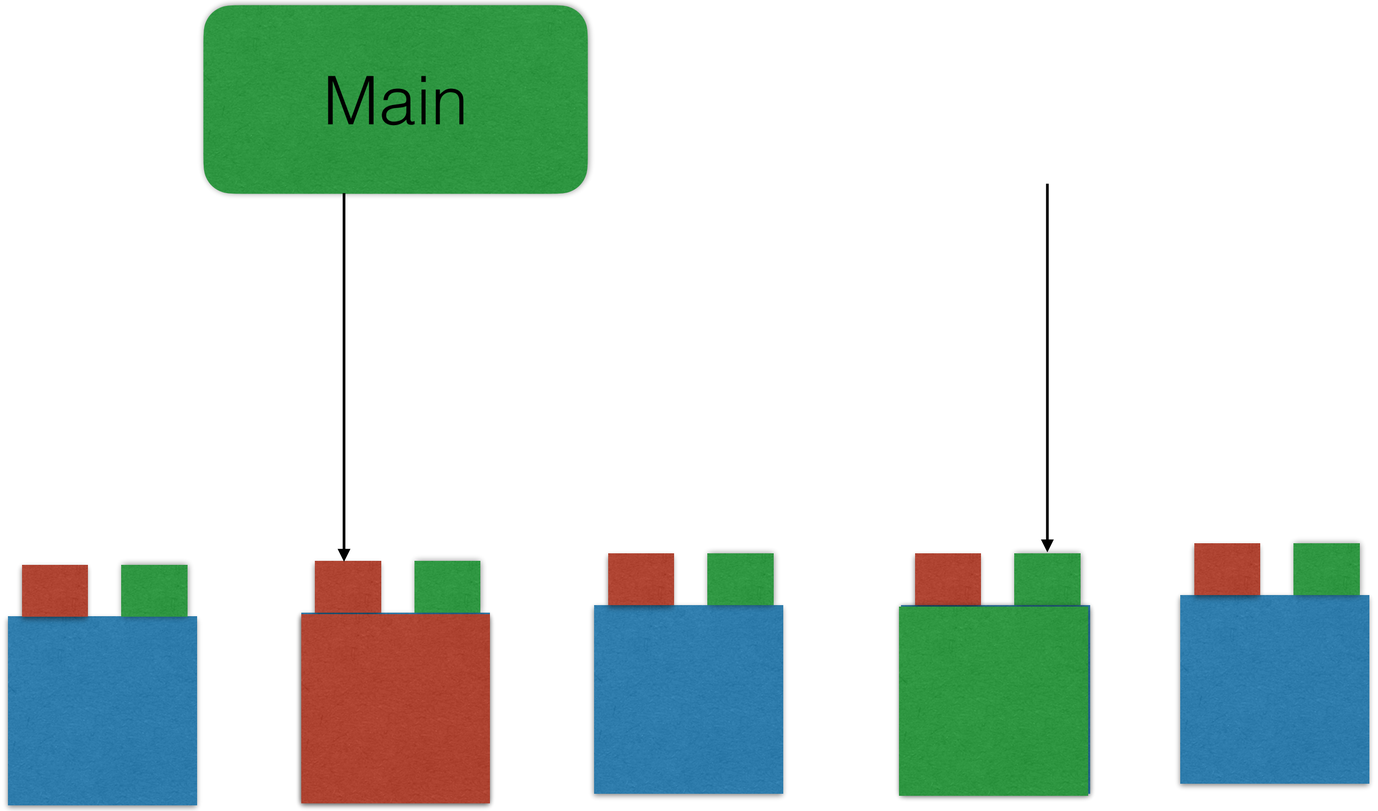
Programming pictorially



Variables:



Object Oriented Programming pictorially

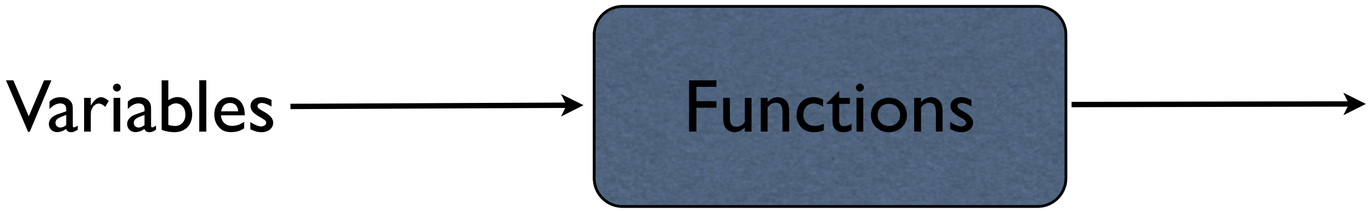


The most important thing

Everything you can do with an object oriented language can be done in an imperative language. The difference is codes written in object oriented languages contain less code, which is good:

less code = fewer bugs

Basic imperative programming




```
double myfunction( unsigned a, double b, bool c );

unsigned mya=2; double myb=20.; bool myc=false;
double answer = myfunction( mya, myb, myc );
```

```
typedef struct {
  unsigned a;
  double b;
  bool c;
} mystruct;
```

→

```
double myfunction( struct mystruct mydata );
mystruct mydata; mydata.a=2;
mydata.b=20.; mydata.c=false
double answer = myfunction( mydata );
```

Object oriented programming

```
typedef struct {
  unsigned a;
  double b;
  bool c;
} mystruct;
```

```
double myfunction( struct mystruct mydata );
mystruct mydata; mydata.a=2;
mydata.b=20.; mydata.c=false
double answer = myfunction( mydata );
```

CLASS DECLARATION

```
class myclass {
private:
   unsigned a;
   double b;
   bool c;
public:
   myclass(): a(2), b(20.), c(false) {}
   double myfunction();
};
```

```
myclass hel;
double answer = hel.myfunction();
```

Creating objects: **The constructor**

```
class m
private
    unsi
    doul
    boo
public:
    myc
    doul
};
```

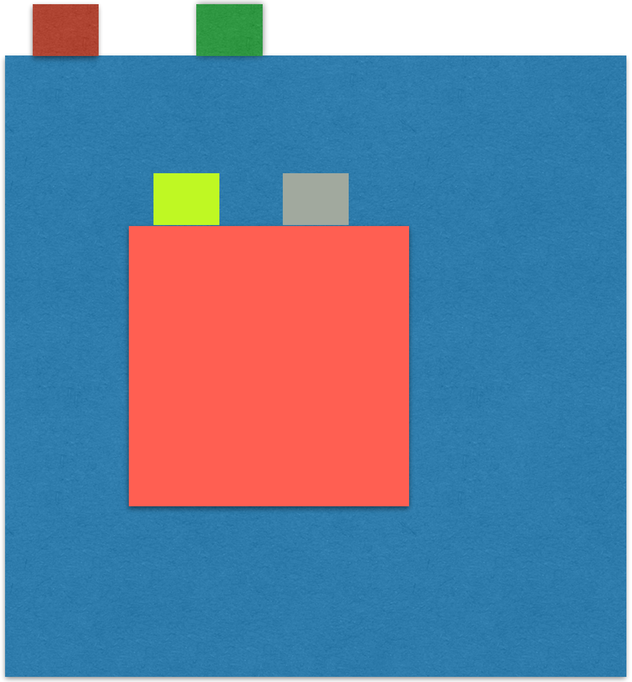
bias — vi MovingRestraint.cpp — 196×58

```
MovingRestraint::MovingRestraint(const ActionOptions&ao):
  PLUMED_BIAS_INIT(ao),
  verse(getNumberOfArguments())
{
  parseVector("VERSE",verse);
  std::vector<long int> ss(1); ss[
  std::vector<double> kk( getNumberO uments() ), aa( getNumberOfArguments() );
  for(int i=0;; i++) {
    // Read in step
    if( !parseNumberedVector("STEP",i,ss) ) break;
    for(unsigned j=0; j<step.size(); j++) {
      if(ss[0]<step[j]) error("in moving restraint step number must always increase");
    }
    step.push_back(ss[0]);

    // Try to read kappa
    if( !parseNumberedVector("KAPPA",i,kk) ) kk=kappa[i-1];
    kappa.push_back(kk);

    // Now read AT
    if( !parseNumberedVector("AT",i,aa) ) aa=at[i-1];
    at.push_back(aa);
  }
  checkRead();

  for(unsigned i=0; i<step.size(); i++) {
    log.printf("  step%u %ld\n",i,step[i]);
    log.printf("  at");
    for(unsigned j=0; j<at[i].size(); j++) log.printf(" %f",at[i][j]);
    log.printf("\n");
    log.printf("  with force constant");
    for(unsigned j=0; j<kappa[i].size(); j++) log.printf(" %f",kappa[i][j]);
    log.printf("\n");
  };

  addComponent("force2"); componentIsNotPeriodic("force2");

  // add the centers of the restraint as additional components th can be retrieved (useful for debug)

  std::string comp;
  for(unsigned i=0; i< getNumberOfArguments() ; i++) {
    comp=getPntrToArgument(i)->getName()+"_cntr"; // each spring has its own center
    addComponent(comp); componentIsNotPeriodic(comp);
    comp=getPntrToArgument(i)->getName()+"_work"; // each spring has its own work
    addComponent(comp); componentIsNotPeriodic(comp);
    comp=getPntrToArgument(i)->getName()+"_kappa"; // each spring has its own kappa
    addComponent(comp); componentIsNotPeriodic(comp);
    work.push_back(0.); // initialize the work value
  }
  addComponent("work"); componentIsNotPeriodic("work");
  tot_work=0.0;

  log<<"  Bibliography ";
  log<<cite("Grubmuller, Heymann, and Tavan, Science 271, 997 (1996)")<<"\n";

}
```

Read keywords from input file

Create values to pass between actions

Inclusion and inheritance

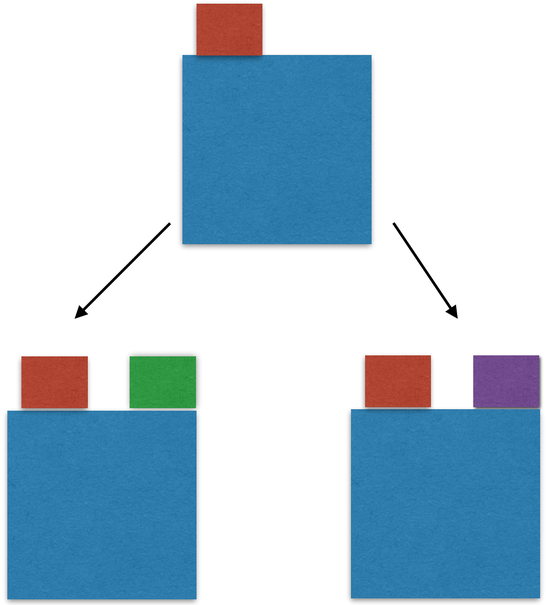


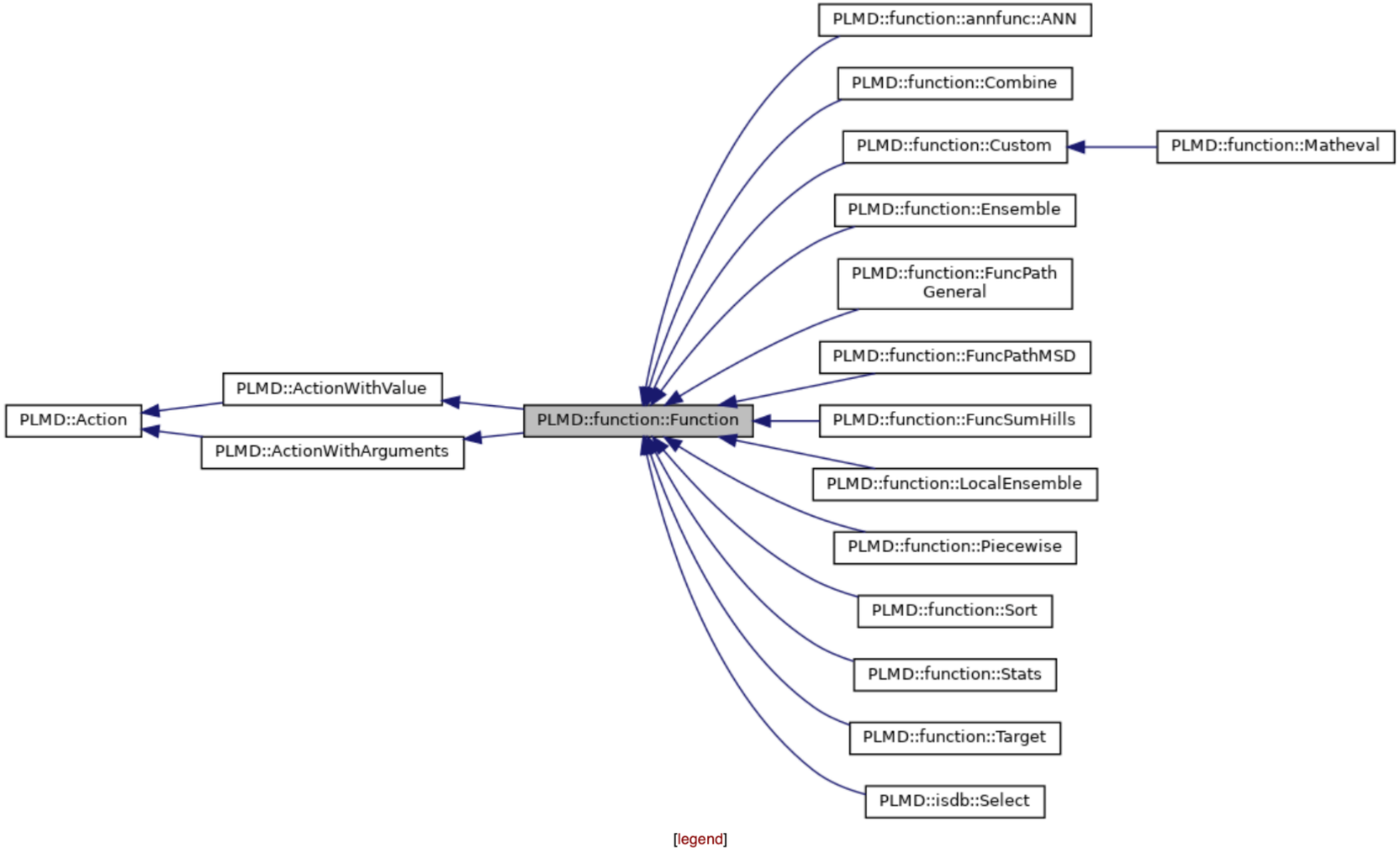
```
class AdjacencyMatrixBase : public ActionWithMatrix {
private:
  bool nopbc, components, read_one_group;
  LinkCells linkcells, threecells;
  std::vector<unsigned> ablocks, threeblocks;
  double nl_cut, nl_cut2;
  unsigned nl_stride;
  unsigned natoms_per_list;
  std::vector<unsigned> nlist;
  void setupThirdAtomBlock( const std::vector<AtomNumber>& tc, std::vector<AtomNumber>& t );
protected:
  Vector getPosition( const unsigned& indno, MultiValue& myvals ) const ;
  void addAtomDerivatives( const unsigned& indno, const Vector& der, MultiValue& myvals ) const ;
  void addThirdAtomDerivatives( const unsigned& indno, const Vector& der, MultiValue& myvals ) const ;
  void setLinkCellCutoff( const bool& symmetric, const double& lcut, double tcut=-1.0 );
  void addBoxDerivatives( const Tensor& vir, MultiValue& myvals ) const ;
public:
  static void registerKeywords( Keywords& keys );
  explicit AdjacencyMatrixBase(const ActionOptions&);
  bool canBeAfterInChain( ActionWithVector* av ) override;
  unsigned getNumberOfDerivatives() override ;
  unsigned getNumberOfColumns() const override;
  void setupForTask( const unsigned& current, std::vector<unsigned> & indices, MultiValue& myvals ) const override;
  // void setupCurrentTaskList() override;
  void updateNeighbourList() override ;
  unsigned retrieveNeighbours( const unsigned& current, std::vector<unsigned> & indices ) const ;
  void performTask( const std::string& controller, const unsigned& index1, const unsigned& index2, MultiValue& myvals ) const override ;
  virtual double calculateWeight( const Vector& pos1, const Vector& pos2, const unsigned& natoms, MultiValue& myvals ) const = 0;
  void runEndOfRowJobs( const unsigned& ival, const std::vector<unsigned> & indices, MultiValue& myvals ) const override ;
};
```

Inheriting functionality from another object

Including functionality from another object

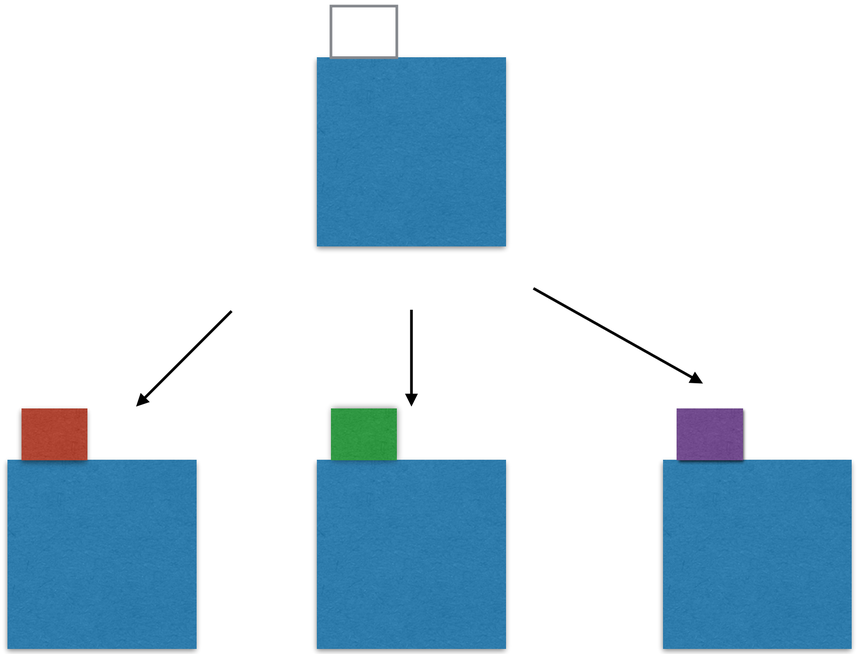
Inheritance





Polymorphism

```
if (type==1){
    dofirstthing(a);
} else if (type==2){
    doseconthing(a);
} else if (type==3){
    dothirdthing(a);
} else if (type==4 ){
    dofourththing(a);
} else if(type==5){
    dofifththing(a);
}
```



```
myclass->dothing();
```

```
  // Check the input actions to determine if we need to calculate constants that
  // depend on masses and charges
  bool firststep=false;
  for(const auto & ip : inputs) {
    if( ip->firststep ) firststep=true;
  }

  int iaction=0;
// calculate the active actions in order (assuming *backward* dependence)
  for(const auto & pp : actionSet) {
    Action* p(pp.get());
    try {
      if(p->isActive()) {
// Stopwatch is stopped when sw goes out of scope.
// We explicitly declare a Stopwatch::Handler here to allow for conditional initialization.
        Stopwatch::Handler sw;
        if(detailedTimers) {
          std::string actionNumberLabel;
          Tools::convert(iaction,actionNumberLabel);
          const unsigned m=actionSet.size();
          unsigned k=0; unsigned n=1; while(n<m) { n*=10; k++; }
          const int pad=k-actionNumberLabel.length();
          for(int i=0; i<pad; i++) actionNumberLabel=" "+actionNumberLabel;
          sw=stopwatch.startStop("4A "+actionNumberLabel+" "+p->getLabel());
        }
        ActionWithValue*av=dynamic_cast<ActionWithValue*>(p);
        if( av && av->calculateOnUpdate() ) continue ;
        ActionAtomistic*aa=dynamic_cast<ActionAtomistic*>(p);
        {
          if(av) av->clearInputForces();
          if(av) av->clearDerivatives();
        }
        {
          if(aa) if(aa->isActive()) aa->retrieveAtoms();
        }
        if(p->checkNumericalDeri ves()) p->calculateNumericalDerivatives();
        else p->calculate();
        // This retrieves compo  alled bias
        if(av) {
          bias+=av->getOutputQuantity("bias");
          work+=av->getOutputQuantity("work");
          av->setGradientsIfNeeded();
        }
        // This makes all values that depend on the (fixed) masses and charges constant
        if( firststep ) p->setupConstantValues( true );
        ActionWithVirtualAtom*avv=dynamic_cast<ActionWithVirtualAtom*>(p);
        if(avv)avv->setGradientsIfNeeded();
      }
    } catch(...) {
      plumed_error_nested() << "An error happened while calculating " << p->getLabel();
    }
    iaction++;
  }
}

void PlumedMain::justApply() {
```

Compute your action's values

Summary

colvar — vi Template.cpp — 300×81

```
class Template : public Colvar {
  bool pbc;

public:
  explicit Template(const ActionOptions&);
// active methods:
  void calculate() override;
  static void registerKeywords(Keywords& keys);
};

PLUMED_REGISTER_ACTION(Template,"TEMPLATE")

void Template::registerKeywords(Keywords& keys) {
  Colvar::registerKeywords(keys);
  keys.addFlag("TEMPLATE_DEFAULT_OFF_FLAG",false,"flags that are by default not performed should be specified like this");
  keys.addFlag("TEMPLATE_DEFAULT_ON_FLAG",true,"flags that are by default performed should be specified like this");
  keys.add("compulsory","TEMPLATE_COMPULSORY","all compulsory keywords should be added like this with a description here");
  keys.add("optional","TEMPLATE_OPTIONAL","all optional keywords that have input should be added like a description here");
  keys.add("atoms","ATOMS","the keyword with which you specify what atoms to use should be added like this");
}

Template::Template(const ActionOptions&ao):
  PLUMED_COLVAR_INIT(ao),
  pbc(true)
{
  std::vector<AtomNumber> atoms;
  parseAtomList("ATOMS",atoms);
  if(atoms.size()!=2)
    error("Number of specified atoms should be 2");
  bool nopbc=!pbc;
  parseFlag("NOPBC",nopbc);
  pbc=!nopbc;
  checkRead();

  log.printf("  between atoms %d %d\n",atoms[0].serial(),atoms[1].serial());
  if(pbc) log.printf("  using periodic boundary conditions\n");
  else    log.printf("  without periodic boundary conditions\n");

  addValueWithDerivatives(); setNotPeriodic();

  requestAtoms(atoms);
}


// calculator
void Template::calculate() {

  Vector distance;
  if(pbc) {
    distance=pbcDistance(getPosition(0),getPosition(1));
  } else {
    distance=delta(getPosition(0),getPosition(1));
  }
  const double value=distance.modulo();
  const double invvalue=1.0/value;

  setAtomsDerivatives(0,-invvalue*distance);
  setAtomsDerivatives(1,invvalue*distance);
  setBoxDerivatives  (-invvalue*Tensor(distance,distance));
  setValue           (value);
}

}
}
```

Declare your new action

Register the name of the action

Write documentation

Read keywords

Create values

Calculate the values