Introduction to Netgen-NGSolve with Python
Getting started with C++ coding

Christopher Lackner

June, 15th 2017

Inst. for Analysis and Scientific Computing, TU Wien

Schedule for today

- Introduction into C++ coding with NGSolve
- Run time evaluation vs. Code generation
- Shared memory parallelization
- Distributed memory parallelization
- 
- Guest Lectures

Overview C++ Introduction

- How to build NGSolve extensions
- My Little NGSolve: create your own...
  - Finite Elements
  - DifferentialOperators
  - Finite Element Spaces
  - Integrators
- Some useful concepts: AutoDiff, IterateElements,...
- Create utility functions for performance critical operations
- Use NGSolve in a C++ only environment

How to build a NGSolve extension

- CMake: Easy, platform independent
- Export your classes and functions to Python

CMake makes setting up a new project with NGSolve easy: You need to provide a file CMakeLists.txt with:

```
# Find NGSolve
find_package(NGSolve CONFIG REQUIRED
    ...
  )
# Create a new Python extension module 'myngspy'
add_ngsolve_python_module(myngspy myngspy.cpp
                    1_myFEM/myElement.cpp ...)
```

This works for Linux, Windows and Mac!

If you want your package to be installable we recommend additionally:

```
# check if CMAKE_INSTALL_PREFIX is set by user, if not install in NGSolve python dir
if(CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT)
  set(CMAKE_INSTALL_PREFIX ${NGSOLVE_INSTALL_DIR}/${NGSOLVE_INSTALL_DIR_PYTHON} CACHE
     PATH "Install dir" FORCE)
endif(CMAKE_INSTALL_PREFIX_INITIALIZED_TO_DEFAULT)

install(TARGETS myngspy DESTINATION .)
```

There must be one .cpp file with the macro PYBIND11_PLUGIN.

In this macro we define all our exported classes/functions

```
namespace py=pybind11;
PYBIND11_PLUGIN(myngspy) {
  py::module m("myngspy", "myngspy documentation string");
  ...
  return m.ptr();
}
```

If the arguments and the return value can be directly converted:

```
m.def("SomeFunction", &SomeCppFunction, "documentation");
```

Else:

```
m.def("SomeFunction", [] (First & first,
                          shared_ptr<Second> second,
                          py::object obj)
{
  if(py::is_none(obj))
    do_some_stuff(first, second);
  else
    {
      auto third = py::cast<shared_ptr<Third>>(obj);
      do_some_other_stuff(first, second, third);
    }
  return some_exported object;
}, arg("first"), arg("second"), arg("third") = py::none(),
"some documentation");
```

Methods of classes are defined the same way.

Overload functions just by defining multiple versions.

Exporting Classes

Short constructor, arguments can be parsed into the c++ constructor:

```
py::class_<ParameterCoefficientFunction /*type*/,
           shared_ptr<ParameterCoefficientFunction> /*holder type*/,
           CoefficientFunction /*base class(es)*/>
    (m, 'Parameter', 'docu string')
    .def (py::init<double>())
    ;
```

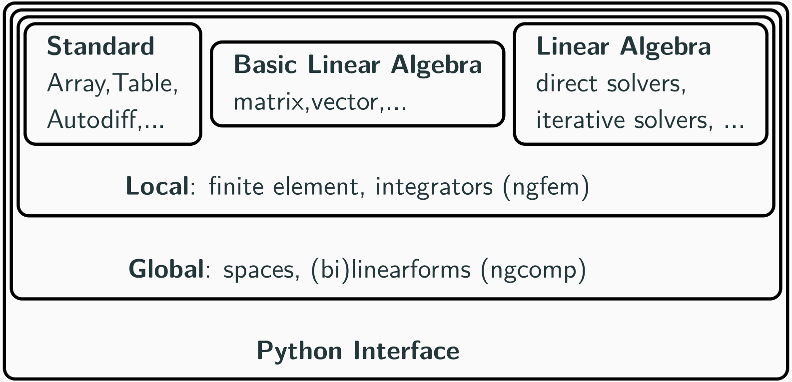
No trivial constructor:

```
py::class_<ParameterCoefficientFunction,
           shared_ptr<ParameterCoefficientFunction>,
           CoefficientFunction>
    (m, 'Parameter', 'docu string')
    .def ('__init__',
          [] (ParameterCoefficientFunction *instance, double val)
             {
               auto newVal = do_something_with_val(val);
        new (instance) ParameterCoefficientFunction(newVal);
          })
    ;
```

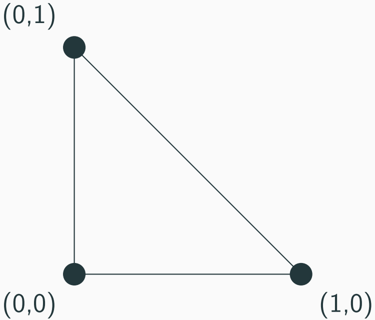
CAVE: Some NGSolve classes have creator functions and do not need to (but still can) export a constructor (i.e. FESpace)

- Tutorial project for C++ programming with NGSolve
- 3 Sections:
  - **Basic**: How to create your own elements, spaces,... - compiles into a python module, installs to CMAKE_INSTALL_PREFIX (default NGSolve python dir)
  - **Advanced**: Some further examples. Compile into local python module (cmake . && make)
  - **Legacy**: Old MyLittleNGS code, not maintained

The layers of NGSolve



Reference element:



Hat functions on reference element:

$$\varphi_1 = x$$
$$\varphi_2 = y$$
$$\varphi_3 = 1 - x - y$$

MyElement

```
class MyLinearTrig : public ScalarFiniteElement<2>
{
public:
  MyLinearTrig ();

  virtual ELEMENT_TYPE ElementType() const { return ET_TRIG; }

  virtual void CalcShape (const IntegrationPoint & ip,
                          BareSliceVector<> shape) const;

  virtual void CalcDShape (const IntegrationPoint & ip,
                           SliceMatrix<> dshape) const;
};
```

These functions calculate point evaluation of the shape functions and their derivatives on the reference domain.

CalcShape & CalcDShape

```
void MyLinearTrig ::  CalcShape
  (const IntegrationPoint & ip,
   BareSliceVector<> shape) const
{
  double x = ip(0);
  double y = ip(1);
  shape(0) = x;
  shape(1) = y;
  shape(2) = 1-x-y;
}
```

```
void MyLinearTrig :: CalcDShape
  (const IntegrationPoint & ip,
   SliceMatrix<> dshape)
   const
{
  dshape(0,0) = 1;
  dshape(0,1) = 0;
  dshape(1,0) = 0;
  dshape(1,1) = 1;
  dshape(2,0) = -1;
  dshape(2,1) = -1;
}
```

My FESpace

```
class MyFESpace : public FESpace
{
  int ndof, nvert;

public:
  MyFESpace (shared_ptr<MeshAccess> ama, const Flags & flags);

  // calculate number of dofs
  virtual void Update(LocalHeap & lh);
  virtual size_t GetNDof () const { return ndof; }

  // return dofs of element ei in dnums array
  virtual void GetDofNrs (ElementId ei, Array<DofId> & dnums) const;

  // return finite element ei allocated on alloc
  virtual FiniteElement & GetFE (ElementId ei,
                                 Allocator & alloc) const;

  // some new functionality our space should have in Python
  int GetNVert() { return nvert; }
};
```

Constructor & Update

```
MyFESpace :: MyFESpace (const MeshAccess & ama,
  const Flags & flags) : FESpace (ama, flags)
{
  evaluator[VOL] =
      make_shared<T_DifferentialOperator<DiffOpId<2>>>();
  flux_evaluator[VOL] =
      make_shared<T_DifferentialOperator<DiffOpGradient<2>>>();
  evaluator[BND] =
      make_shared<T_DifferentialOperator<DiffOpIdBoundary<2>>>();

  // (still) needed to draw solution, SetValues
  integrator[VOL] =
      GetIntegrators().CreateBFI("mass", ma->GetDimension(),
        make_shared<ConstantCoefficientFunction>(1));
}

void MyFESpace :: Update(LocalHeap & lh)
{
  ndof = ma.GetNV(); // number of vertices
  nvert = ndof;
}
```

```
 FiniteElement & MyFESpace :: GetFE (ElementId ei,
                     Allocator & alloc) const
{
  if (ei.IsVolume())
    return * new (alloc) MyLinearTrig;
  else
    return * new (alloc) FE_Segm1;
}

void MyFESpace :: GetDofNrs (ElementId ei,
                Array<DofId> & dnums) const
{
  // returns dofs of element ei
  // may be a volume triangle or boundary segment
  dnums.SetSize(0);
  // first dofs are vertex numbers:
  for (auto v : ma->GetElVertices(ei))
    dnums.Append (v);
}
```

Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position $\Rightarrow$ free memory

Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position $\Rightarrow$ free memory

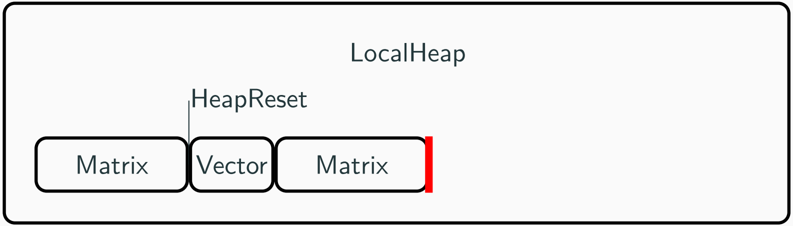
Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position $\Rightarrow$ free memory

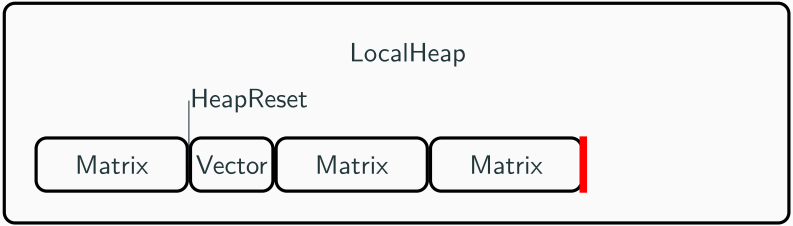
Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory



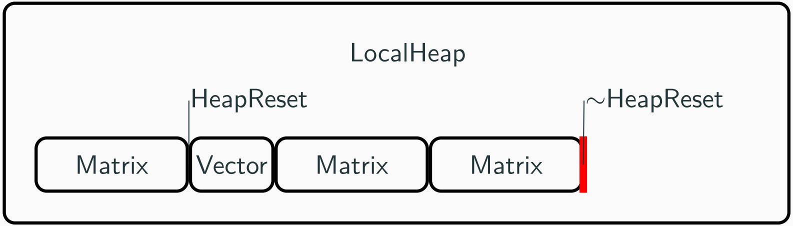
Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory



Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position $\Rightarrow$ free memory



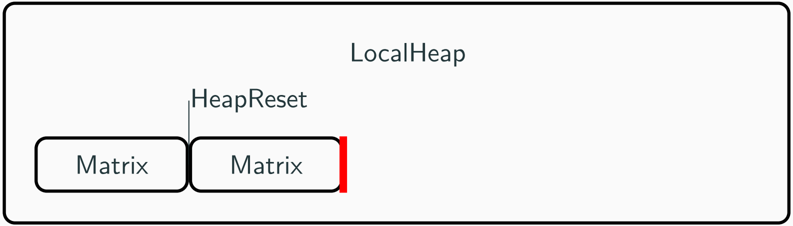
Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

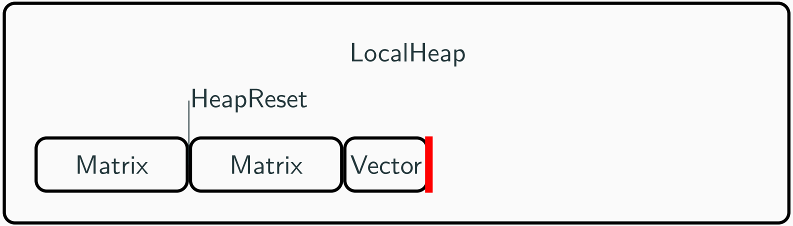
- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

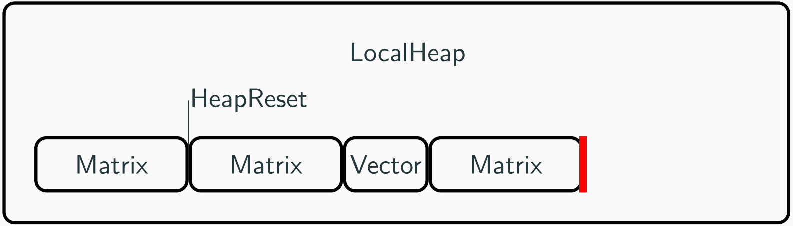


- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

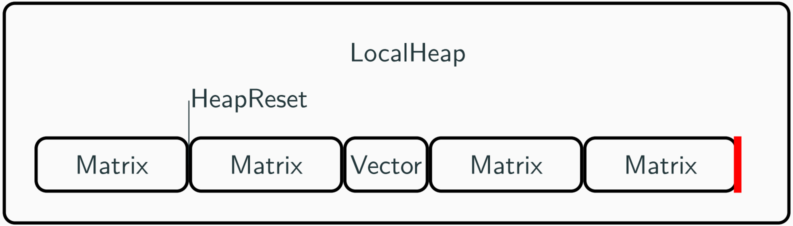


Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory

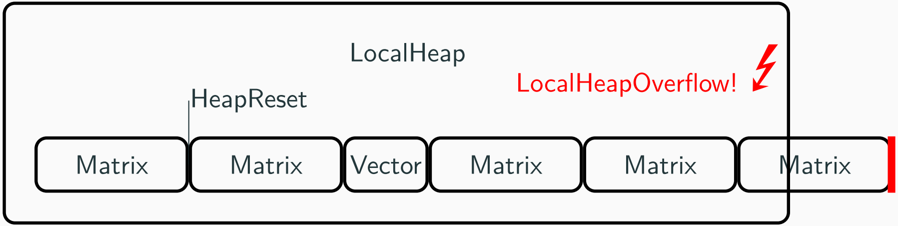


- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position $\Rightarrow$ free memory



Efficient Memory Management - LocalHeap

- Allocator - Base class calls new to allocate object
- LocalHeap - Optimized memory handler:
  - Calls new for a memory block with predefined size
  - Overloaded new operator with efficient allocation on block
  - HeapReset to reset pointer to stored position ⇒ free memory



FESpace constructor converts kwargs to Flags

⇒ use FESpace constructor and don't write your own Python __init__

For this we need to register the FESpace in NGSolve

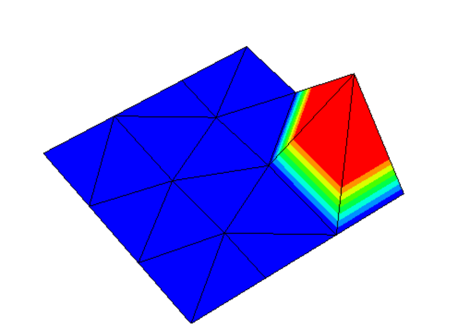
```
static RegisterFESpace<MyFESpace> initifes ("myfespace");
```

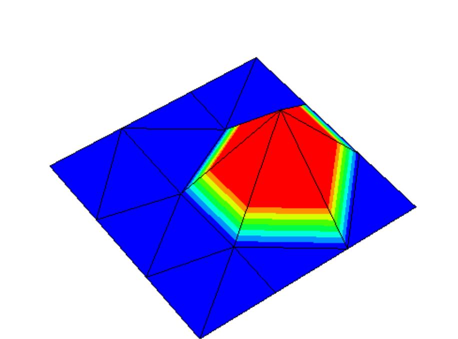
If we want additional functionality: export without constructor

```
py::class_<MyFESpace, shared_ptr<MyFESpace>, FESpace>
(m, "MyFESpace", "FESpace with first order trigs on 2d mesh")
// export some additional function
.def("GetNVert", &MyFESpace::GetNVert)
;
```

Shapetester

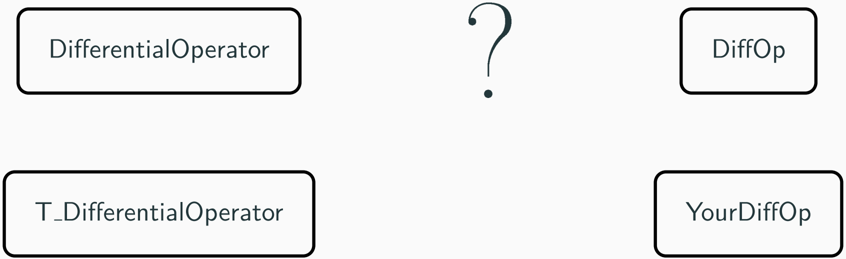
```
mesh = Mesh(unit_square.GenerateMesh(maxh=0.2))
fes = FESpace("myfespace", mesh)
u = GridFunction(fes,"shapes")
Draw(u)

# we can use the additionally exported function here
for i in range(fes.GetNVert()):
    print("Draw basis function ", i)
    u.vec[:] = 0
    u.vec[i] = 1
    Redraw()
    input("press key to draw next shape function")
```



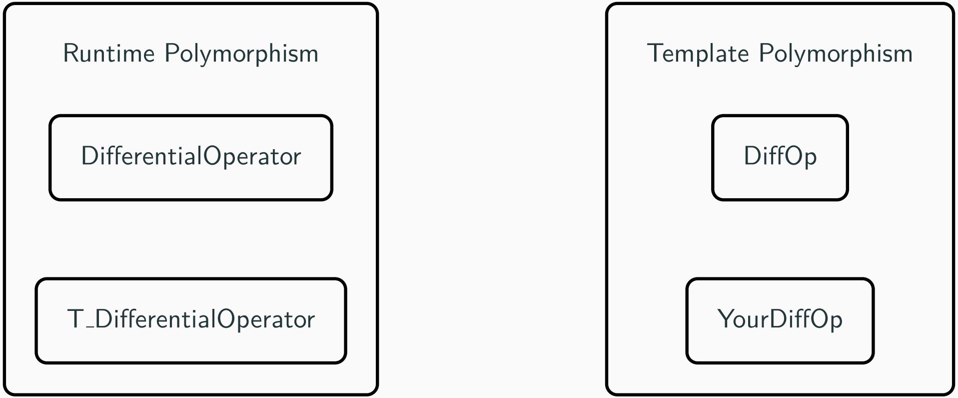


DifferentialOperator

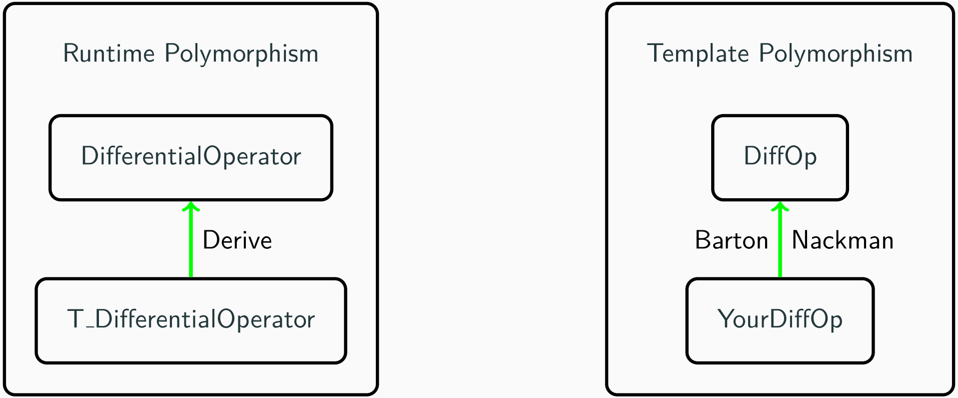
DifferentialOperator



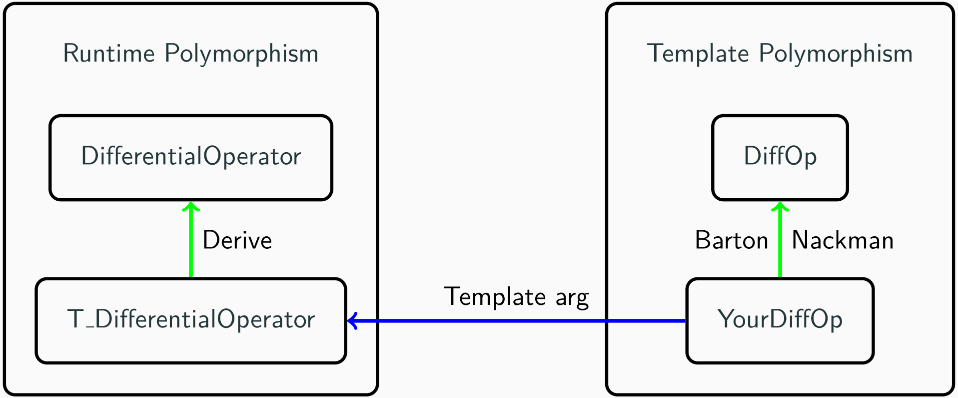
DifferentialOperator

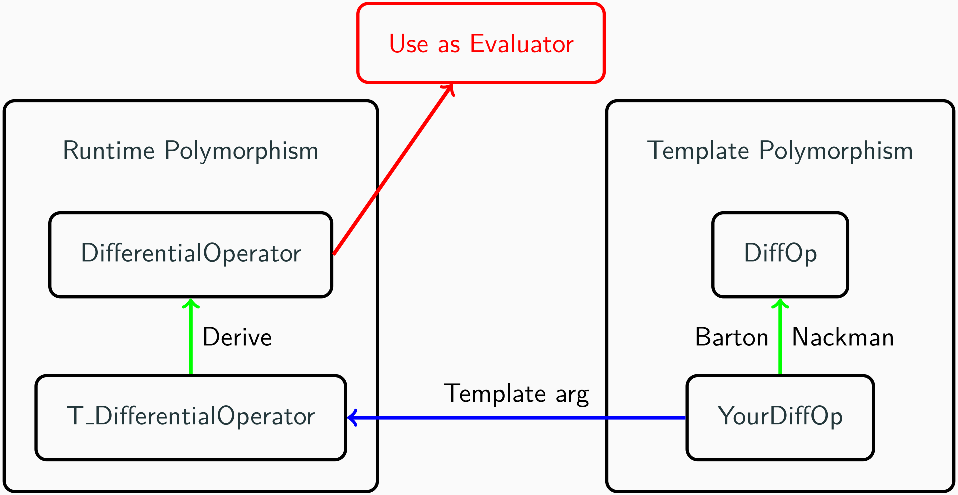


DifferentialOperator



DifferentialOperator




Use as Evaluator
Runtime Polymorphism
DifferentialOperator
Derive
T_DifferentialOperator
Template arg
Template Polymorphism
DiffOp
Barton
Nackman
YourDiffOp

```
class MyIdentity : public DiffOp<MyIdentity>
{
public:
  enum { DIM = 1 };
  enum { DIM_SPACE = 2 };
  enum { DIM_ELEMENT = 2 };
  enum { DIM_DMAT = 1 };
  enum { DIFFORDER = 0 };

  template <typename FEL, typename MIP, typename MAT>
  static void GenerateMatrix (const FEL & fel, const MIP & mip,
                              MAT && mat, LocalHeap & lh)
  {
    HeapReset hr(lh);
    mat.Row(0) = static_cast<const ScalarFiniteElement<2>&>(fel).GetShape(mip.IP(), lh);
  }

};
```

Set Additional Operators

Add GetAdditionalEvaluators to MyFESpace

```
virtual SymbolTable<shared_ptr<DifferentialOperator>> GetAdditionalEvaluators () const
    override
{
  SymbolTable<shared_ptr<DifferentialOperator>> additional;
  additional.Set ("myId", make_shared<T_DifferentialOperator<MyIdentity>>());
  return additional;
}
```

From Python we can call it on ProxyFunctions and GridFunctions

```
u,v = fes.TrialFunction(), fes.TestFunction()
a += SymbolicBFI(u.Operator("myId")*v.Operator("myId"))
gfu = GridFunction(fes)
Draw(gfu.Operator("myId"),mesh,"myId")
```

MyLaplaceIntegrator

$$\int_\Omega \lambda \nabla u \nabla v \, dx$$

```
class MyLaplaceIntegrator : public BilinearFormIntegrator
{
  shared_ptr<CoefficientFunction> coef_lambda;
public:
  MyLaplaceIntegrator(shared_ptr<CoefficientFunction> coeffs)
  : coef_lambda(coef) { ; }
  virtual bool IsSymmetric () const { return true; }
  virtual VorB VB() const { return VOL; }

  virtual void
  CalcElementMatrix (const FiniteElement & fel,
                     const ElementTransformation & eltrans,
                     FlatMatrix<double> elmat,
                     LocalHeap & lh) const;
};
```

CalcElementMatrix

```
void MyLaplaceIntegrator ::
CalcElementMatrix (const FiniteElement & base_fel,
            const ElementTransformation & eltrans,
            FlatMatrix<double> elmat,
            LocalHeap & lh) const
{
  const ScalarFiniteElement<2> & fel =
    dynamic_cast<const ScalarFiniteElement<2> &> (base_fel);
  int ndof = fel.GetNDof();
  elmat = 0;
  Matrix<> dshape_ref(ndof, 2);
  Matrix<> dshape(ndof, 2);
  IntegrationRule ir(fel.ElementType(), 2*fel.Order());
  for (int i = 0 ; i < ir.GetNIP(); i++)
    {
      MappedIntegrationPoint<2,2> mip(ir[i], eltrans);
      double lam = coef_lambda -> Evaluate (mip);
      fel.CalcDShape (ir[i], dshape_ref);
      dshape = dshape_ref * mip.GetJacobianInverse();
      double fac = mip.IP().Weight() * mip.GetMeasure();
      elmat += (fac*lam) * dshape * Trans(dshape);
    }
}
```

MySourceIntegrator

$$\int_\Omega fv \, dx$$

```
class MySourceIntegrator : public LinearFormIntegrator
{
  shared_ptr<CoefficientFunction> coef_f;
public:
  MySourceIntegrator (shared_ptr<CoefficientFunction> coef)
  : coef_f(coef) { ; }
  virtual VorB VB() const { return VOL; }

  virtual void
  CalcElementVector (const FiniteElement & fel,
                     const ElementTransformation & eltrans,
                     FlatVector<double> elvec,
                     LocalHeap & lh) const;
};
```

CalcElementVector

```
void MySourceIntegrator ::
CalcElementVector (const FiniteElement & base_fel,
            const ElementTransformation & eltrans,
            FlatVector<double> elvec,
            LocalHeap & lh) const
{
  const ScalarFiniteElement<2> & fel =
    dynamic_cast<const ScalarFiniteElement<2> &> (base_fel);
  int ndof = fel.GetNDof();
  elvec = 0;
  Vector<> shape(ndof);
  IntegrationRule ir(fel.ElementType(), 2*fel.Order());
  for (int i = 0 ; i < ir.GetNIP(); i++)
    {
      MappedIntegrationPoint<2,2> mip(ir[i], eltrans);
      double f = coef_f -> Evaluate (mip);
      fel.CalcShape (ir[i], shape);
      double fac = mip.IP().Weight() * mip.GetMeasure();
      elvec += (fac*f) * shape;
    }
}
```

Exporting Integrators

```
py::class_<MyLaplaceIntegrator, shared_ptr<MyLaplaceIntegrator>,
           BilinearFormIntegrator>
(m, "MyLaplace")
.def(py::init<shared_ptr<CoefficientFunction>>())
;
py::class_<MySourceIntegrator, shared_ptr<MySourceIntegrator>,
           LinearFormIntegrator>
(m, "MySource")
.def(py::init<shared_ptr<CoefficientFunction>>())
;
```

```
from netgen.geom2d import unit_square
from ngsolve import *
from myngspy import *

mesh = Mesh(unit_square.GenerateMesh(maxh=0.2))
fes = FESpace("myfespace", mesh, dirichlet="top|bottom|right|left")
u,v = fes.TrialFunction(), fes.TestFunction()

a = BilinearForm(fes)
a += MyLaplace(CoefficientFunction(1))

f = LinearForm(fes)
f += MySource(x*y)

a.Assemble()
f.Assemble()

u = GridFunction(fes)
u.vec.data = a.mat.Inverse(fes.FreeDofs()) * f.vec
Draw(u)
```

MyLaplace vs. SymbolicIntegrator

```
mesh = Mesh(unit_square.GenerateMesh(maxh=0.1))
for i in range(6):
    mesh.Refine()

fes = H1(mesh,order=1)
print("ndof = ", fes.ndof)

timing={}
integrators = {"MyLaplace" : MyLaplace(CoefficientFunction(1)),
          "Symbolic" : SymbolicBFI(grad(fes.TrialFunction())*grad(fes.TestFunction()))}
for name, integrator in integrators.items():
    start = time()
    with TaskManager():
        a = BilinearForm(fes)
        a += integrator
        a.Assemble()
    timing[name] = time()-start

for name, value in timing.items():
    print(name, " needed ", value, " seconds for assembling")
```

MyHOElement

```
class MyHighOrderTrig : public ScalarFiniteElement<2>,
                        public VertexOrientedFE<ET_TRIG>
{
public:
  // ScalarFE with (order+1)*(order+2)/2 ndofs and order 'order'
  MyHighOrderTrig (int order)
  : ScalarFiniteElement<2> ((order+1)*(order+2)/2, order) { ; }
  virtual ELEMENT_TYPE ElementType() const { return ET_TRIG; }

  virtual void CalcShape (const IntegrationPoint & ip,
                          BareSliceVector<> shape) const;

  virtual void CalcDShape (const IntegrationPoint & ip,
                           SliceMatrix<> dshape) const;

private:
  template <class T>
  void T_CalcShape (const T & x, const T & y,
                    BareSliceVector<T> shape) const;
};
```

```
template <class T>
void MyHighOrderTrig :: T_CalcShape (const T & x, const T & y,
                                     BareSliceVector<T> shape) const {
  T lami[3] = { x, y, 1-x-y };
  for (int i = 0; i < 3; i++)  shape[i] = lami[i];
  int ii = 3;
  ArrayMem<T, 20> polx(order+1), poly(order+1);
  for (int i = 0; i < 3; i++)
    if (order >= 2) {
    auto edge = GetVertexOrientedEdge(i);
    ScaledIntegratedLegendrePolynomial (order,
                 lami[edge[1]]-lami[edge[0]],
                 lami[edge[0]]+lami[edge[1]], polx);
    for (int j = 2; j <= order; j++)
      shape[ii++] = polx[j];
      }
  if (order >= 3) {
      T bub = x * y * (1-x-y);
      ScaledLegendrePolynomial(order-2, lami[1]-lami[0],
                      lami[1]+lami[0], polx);
      LegendrePolynomial (order-1, 2*lami[2]-1, poly);
      for (int i = 0; i <= order-3; i++)
    for (int j = 0; j <= order-3-i; j++)
      shape[ii++] = bub * polx[i] * poly[j];
    }
}
```

- Class supporting automatic differentiation
- instance knows it's value and derivative
- Algebraic operations are overloaded using product-rule, ...

```
// AutoDiff with 2 dim derivative, value is 3,
// derivative is 0-th unit vector
AutoDiff<2> x (3.0, 0);
```

CalcShape & CalcDShape

```
void MyHighOrderTrig :: CalcShape (const IntegrationPoint & ip,
                            BareSliceVector<> shape) const
{
  double x = ip(0);
  double y = ip(1);
  T_CalcShape (x, y, shape);
}


void MyHighOrderTrig :: CalcDShape (const IntegrationPoint & ip,
                            SliceMatrix<> dshape) const
{
  AutoDiff<2> adx (ip(0), 0);
  AutoDiff<2> ady (ip(1), 1);
  Vector<AutoDiff<2> > shapearray(ndof);
  T_CalcShape<AutoDiff<2>> (adx, ady, shapearray);
  for (int i = 0; i < ndof; i++)
    {
      dshape(i, 0) = shapearray[i].DValue(0);
      dshape(i, 1) = shapearray[i].DValue(1);
    }
}
```

MyHOFESpace

```
class MyHighOrderFESpace : public FESpace
{
  int order;
  int ndof;
  Array<int> first_edge_dof;
  Array<int> first_cell_dof;
public:
  MyHighOrderFESpace (shared_ptr<MeshAccess> ama,
                      const Flags & flags);

  virtual void Update(LocalHeap & lh);
  virtual size_t GetNDof () const { return ndof; }

  virtual void GetDofNrs (ElementId ei, Array<DofId> & dnums) const;
  virtual FiniteElement & GetFE (ElementId ei,
                                 Allocator & alloc) const;
};
```

Update

```
void MyHighOrderFESpace :: Update(LocalHeap & lh)
{
  int n_vert = ma->GetNV();
  int n_edge = ma->GetNEdges();
  int n_cell = ma->GetNE();

  first_edge_dof.SetSize (n_edge+1);
  int ii = n_vert;
  for (int i = 0; i < n_edge; i++, ii+=order-1)
    first_edge_dof[i] = ii;
  first_edge_dof[n_edge] = ii;

  first_cell_dof.SetSize (n_cell+1);
  for (int i = 0; i < n_cell; i++, ii+=(order-1)*(order-2)/2)
    first_cell_dof[i] = ii;
  first_cell_dof[n_cell] = ii;

  ndof = ii;
}
```

GetDofNrs

```
void MyHighOrderFESpace :: GetDofNrs (ElementId ei,
                                      Array<DofId> & dnums) const
{
  dnums.SetSize(0);
  Ngs_Element ngel = ma->GetElement (ei);
  // vertex dofs
  for (auto v : ngel.Vertices())
    dnums.Append(v);
  // edge dofs
  for (auto e : ngel.Edges())
    {
      int first = first_edge_dof[e];
      int next  = first_edge_dof[e+1];
      for (int j = first; j < next; j++)
        dnums.Append (j);
    }
  // cell dofs
  if (ei.IsVolume())
    {
      int first = first_cell_dof[ei.Nr()];
      int next  = first_cell_dof[ei.Nr()+1];
      for (int j = first; j < next; j++)
        dnums.Append (j);
    }
}
```

```
FiniteElement & MyHighOrderFESpace :: GetFE (ElementId ei,
                                             Allocator & alloc) const
{
  Ngs_Element ngel = ma->GetElement (ei);
  switch (ngel.GetType())
    {
    case ET_TRIG:
      {
    MyHighOrderTrig * trig = new (alloc) MyHighOrderTrig(order);
    trig->SetVertexNumbers (ngel.vertices);
    return *trig;
      }
    case ET_SEGM:
      {
    MyHighOrderSegm * segm = new (alloc) MyHighOrderSegm(order);
    segm->SetVertexNumbers (ngel.vertices);
    return *segm;
      }
    default:
      throw Exception (string("Element type ")+
                  ToString(ngel.GetType())+" not supported");
    }
}
```

```
from netgen.geom2d import unit_square
from ngsolve import *
from myngspy import *
mesh = Mesh(unit_square.GenerateMesh(maxh=0.2))
fes = FESpace("myhofespace", mesh, dirichlet="top|bottom|right|left", order = 5)
u,v = fes.TrialFunction(), fes.TestFunction()

a = BilinearForm(fes)
a += MyLaplace(CoefficientFunction(1))

f = LinearForm(fes)
f += MySource(x*y)

a.Assemble()
f.Assemble()

u = GridFunction(fes)
u.vec.data = a.mat.Inverse(fes.FreeDofs()) * f.vec
Draw(u)
```

all_in_one.cpp - Part 1

```
#include <solve.hpp>
using namespace ngsolve;
int main(int argc, char** argv)
{
  auto ma = make_shared<MeshAccess>("square.vol");
  Flags flags_fes;
  flags_fes.SetFlag ("order", 4);
  auto fes = make_shared<H1HighOrderFESpace> (ma, flags_fes);
  Flags flags_gfu;
  auto gfu = make_shared<T_GridFunction<double>> (fes, "u",
                                                  flags_gfu);
  Flags flags_bfa;
  auto bfa = make_shared<T_BilinearFormSymmetric<double>> (fes,
                                                  "a", flags_bfa);
  shared_ptr<BilinearFormIntegrator> bfi =
      make_shared<LaplaceIntegrator<2>>(
           make_shared<ConstantCoefficientFunction> (1));
  bfa -> AddIntegrator (bfi);
  Array<double> penalty(ma->GetNBoundaries());
  penalty = 0.0;
  penalty[0] = 1e10;
  bfi = make_shared<RobinIntegrator<2>> (
          make_shared<DomainConstantCoefficientFunction> (penalty));
  bfa -> AddIntegrator (bfi);
  ...
```

all_in_one.cpp - Part 2

```
  ...
  Flags flags_lff;
  auto lff = make_shared<T_LinearForm<double>>(fes, "f", flags_lff);
  auto lfi = make_shared<SourceIntegrator<2>>(
                      make_shared<ConstantCoefficientFunction>(5));
  lff -> AddIntegrator (lfi);

  LocalHeap lh(100000);
  fes -> Update(lh);
  fes -> FinalizeUpdate(lh);
  gfu -> Update();
  bfa -> Assemble(lh);
  lff -> Assemble(lh);

  const BaseMatrix & mata = bfa -> GetMatrix();
  const BaseVector & vecf = lff -> GetVector();
  BaseVector & vecu = gfu -> GetVector();

  auto inverse = mata.InverseMatrix(fes->GetFreeDofs());

  vecu = *inverse * vecf;
  cout << "Solution vector = " << endl << vecu << endl;
  return 0;
}
```

myAssembling.cpp

```cpp
shared_ptr<GridFunction> MyAssemble(shared_ptr<FESpace> fes,
          shared_ptr<BilinearFormIntegrator> bfi,
          shared_ptr<LinearFormIntegrator> lfi)
{
  auto ma = fes->GetMeshAccess();
  int ndof = fes->GetNDof();
  int ne = ma->GetNE();
  Array<int> dnums;
  Array<int> cnt(ne);
  for (auto ei : ma->Elements(VOL))
    {
      fes->GetDofNrs (ei, dnums);
      cnt[ei.Nr()] = dnums.Size();
    }
  Table<int> el2dof(cnt);
  for (auto ei : ma->Elements(VOL))
    {
      fes->GetDofNrs (ei, dnums);
      el2dof[ei.Nr()] = dnums;
    }
  auto mat = make_shared<SparseMatrixSymmetric<double>> (ndof,
                                                          el2dof);
  VVector<double> vecf (fes->GetNDof());
  *mat = 0.0;
  vecf = 0.0;
  ...
```

IterateElements

```
  ...
  LocalHeap lh(100000);
  IterateElements(*fes, VOL, lh, [&] (FESpace::Element el,
                                      LocalHeap &lh)
        {
    const ElementTransformation& eltrans = ma->GetTrafo(el,lh);
    const FiniteElement& fel = fes->GetFE(el,lh);
    auto dofs = el.GetDofs();
    FlatMatrix<> elmat(dofs.Size(),lh);
    bfi->CalcElementMatrix(fel,eltrans,elmat,lh);
    mat->AddElementMatrix(dofs,elmat);
    FlatVector<> elvec(dofs.Size(),lh);
    lfi->CalcElementVector(fel, eltrans, elvec, lh);
    vecf.AddIndirect(dofs,elvec);
    });
  shared_ptr<BaseMatrix> inv=mat->InverseMatrix(fes->GetFreeDofs());
  Flags gfuFlags;
  auto gfu=make_shared<T_GridFunction<double>>(fes, "u", gfuFlags);
  gfu->Update();
  gfu -> GetVector() = (*inv) * vecf;
  return gfu;
}
```

Thank you for your attention!