Matrix free linear algebra in OOPS

Roel Stappers

[1]Norwegian Meteorological Institute
roels@met.no

ECMWF
23 November 2016

Formulations of DA and flexibility in OOPS

Formulations of DA and flexibility in OOPS

Primal formulation ($\mathbf{d} = \mathbf{y} - \mathcal{H}(x_0^g)$, $b = x_0^b - x_0^g$)

$$(\mathbf{B}^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\delta x_0 = \mathbf{B}^{-1}b + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

Formulations of DA and flexibility in OOPS

Primal formulation ($\mathbf{d} = \mathbf{y} - \mathcal{H}(x_0^g)$, $b = x_0^b - x_0^g$)

$$(\mathbf{B}^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\delta x_0 = \mathbf{B}^{-1}b + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

Saddle point formulation

$$\begin{bmatrix} \mathbf{B}^{-1} & \mathbf{H}^T \\ \mathbf{H} & -\mathbf{R} \end{bmatrix} \begin{bmatrix} \delta x \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{B}^{-1}b \\ \mathbf{d} \end{bmatrix}$$

Dual formulation (3D/4D-PSAS)

$$(\mathbf{HBH}^T + \mathbf{R})\lambda = -\mathbf{d} + \mathbf{H}b$$
$$\delta x = -\mathbf{BH}^T\lambda + b$$

Formulations of DA and flexibility in OOPS

Primal formulation ($\mathbf{d} = \mathbf{y} - \mathcal{H}(x_0^g)$, $b = x_0^b - x_0^g$)

$$(\mathbf{B}^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\delta x_0 = \mathbf{B}^{-1}b + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

Saddle point formulation

$$\begin{bmatrix} \mathbf{B}^{-1} & \mathbf{H}^T \\ \mathbf{H} & -\mathbf{R} \end{bmatrix} \begin{bmatrix} \delta x \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{B}^{-1}b \\ \mathbf{d} \end{bmatrix}$$

Dual formulation (3D/4D-PSAS)

$$(\mathbf{HBH}^T + \mathbf{R})\lambda = -\mathbf{d} + \mathbf{H}b$$
$$\delta x = -\mathbf{BH}^T\lambda + b$$

Weak constraint 4D-VAR

$$(\mathbf{L}^T\mathbf{D}^{-1}\mathbf{L} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\delta\mathbf{x} = \mathbf{L}^T\mathbf{D}^{-1}\mathbf{b} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

- Saddle point weak constraint 4D-VAR etc. EDA, EnKF, ETKF
- Flexibility to change linear equation solvers (PCG, MINRES, RPCG, GMRES)

Saddle point formulations in OOPS

Currently the saddle point formulation introduces new classes for

- `SaddlePointMatrix`,
- `SaddlePointVector`,
- `SaddlePointMinimizer`,
- `SaddlePointPreconditionerMatrix`,
- `SaddlePointLMPMatrix`

One of the aims of the mfla-lib is to simplify the construction of these block Matrices, e.g. to construct the operator

$$S = \begin{bmatrix} B^{-1} & H^T \\ H & -R \end{bmatrix}$$

we write

```
auto S = Binv & ~H | H & -R;
```

Here `Binv` acts on `ModelIncrements` and `~H` acts on `Departures`. `S` will act on objects of the form

```
auto xvy = x | y;
```

Where `x` is an `ModelIncrement` and `y` is a `Departure`.
No need to introduce new classes for new saddle point formulation.

DualVectors (container classes) and matrix multiplication

- The classes `HessianMatrix`, `HtRinvHMatrix` and `HBHtMatrix` in OOPS can be generated automatically at compile time, e.g.

```
auto HBHt = H*B*~H;
```

- The class `DualVector` that contains `Departures` for J_o, `Increments` for J_c, `ControlIncrements` for J_b and J_q should be generate automatically at compile time.
- Note
  - `class HMatrix` in OOPS maps `ControlIncrement` to `DualVector`[1]

[1] The adjoint maps from `const DualVector`. See later slides on signatures for TL and AD operators

A brief “introduction” to C++ (templates)

C++ introduction: Classes

```
class myFunctorClass {
 public:                                         // Access specifier
  myFunctorClass (int x) : _x( x ) {}           // Constructor
  int operator() (int y) { return _x + y; } // Overloaded operator
 private:                                        // Access specifier
  int _x;                                        // Data member
};

int main() {
  myFunctorClass addFive( 5 ); // addFive is an object of type myFunctorClass

  std::cout << addFive( 6 );  // Calls operator()

  return 0;
}
```

C++ introduction: Non-type template parameters

```
template <int N>
struct Factorial {
 static const int result = N * Factorial<N-1>::result;
};
```

C++ introduction: Non-type template parameters

```
template <int N>
struct Factorial {
 static const int result = N * Factorial<N-1>::result;
};

template <>
struct Factorial<0> {
 static const int result = 1;
};

int main() {
 std::cout << Factorial<5>::result << "\n";
 return 0;
}
```

- The value of `Factorial<5>::result` is determined at compile time.
- Recursion instead of for-loops
- Template specialization (`Factorial<0>`) instead of if-then-else constructions

C++ introduction: Type template parameters

```
template <typename T>
inline T const Max (T const& a, T const& b)
{
    return a < b ? b:a;
}

int main () {
    int i = 39;
    int j = 20;
    cout << "Max(i,␣j):␣" << Max(i, j) << endl;

    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1,␣f2):␣" << Max(f1, f2) << endl;
}
```

- `const&` similar to `intent(in)`
- Function template with automatic type deduction
- If Max was a class template we would write `Max<int>`
- Compile-time polymorphism instead of run-time polymorphism.

C++ introduction: Partial template specialization

```cpp
template<class T1, class T2, int I>
class A {}; // primary template

template<class T, int I>
class A<T, T*, I> {}; // partial specialization where T2 is a pointer to T1

template<class T, class T2, int I>
class A<T*, T2, I> {}; // partial specialization where T1 is a pointer

template<class T>
class A<int, T*, 5> {}; // partial specialization where T1 is int, I is 5,
                        // and T2 is a pointer
```

Current OOPS implementation

DualVector (`dxjb` is `ControlIncrement`, `dxjo` is `vector<Departure>`, `dxjc` is `vector<Increment>`)

```
template<typename MODEL>
DualVector<MODEL> & DualVector<MODEL>::operator+=(const DualVector & rhs) {
  ASSERT(this->compatible(rhs));
  if (dxjb_ != 0) {
    *dxjb_ += *rhs.dxjb_;
  }
  for (unsigned jj = 0; jj < dxjo_.size(); ++jj) {
    *dxjo_[jj] += *rhs.dxjo_[jj];
  }
  for (unsigned jj = 0; jj < dxjc_.size(); ++jj) {
    *dxjc_[jj] += *rhs.dxjc_[jj];
  }
  return *this;
}
// -----------------------------------------------------------------------
template<typename MODEL>
DualVector<MODEL> & DualVector<MODEL>::operator-=(const DualVector & rhs) {
  ASSERT(this->compatible(rhs));
  if (dxjb_ != 0) {
    *dxjb_ -= *rhs.dxjb_;
  }
  for (unsigned jj = 0; jj < dxjo_.size(); ++jj) {
    *dxjo_[jj] -= *rhs.dxjo_[jj];
  }
  for (unsigned jj = 0; jj < dxjc_.size(); ++jj) {
    *dxjc_[jj] -= *rhs.dxjc_[jj];
  }
  return *this;
}
// -----------------------------------------------------------------------
template<typename MODEL>
DualVector<MODEL> & DualVector<MODEL>::operator*=(const double zz) {
  if (dxjb_ != 0) {
    *dxjb_ *= zz;
  }
  for (unsigned jj = 0; jj < dxjo_.size(); ++jj) {
    *dxjo_[jj] *= zz;
  }
  for (unsigned jj = 0; jj < dxjc_.size(); ++jj) {
    *dxjc_[jj] *= zz;
```

SaddlePointVector (lambda is a DualVector, dx is a ControlIncrement)

```
template<typename MODEL> SaddlePointVector<MODEL> &
        SaddlePointVector<MODEL>::operator=(const SaddlePointVector & rhs) {
  *lambda_ = *rhs.lambda_;
  *dx_     = *rhs.dx_;
  return *this;
}
template<typename MODEL> SaddlePointVector<MODEL> &
        SaddlePointVector<MODEL>::operator+=(const SaddlePointVector & rhs) {
  *lambda_ += *rhs.lambda_;
  *dx_     += *rhs.dx_;
  return *this;
}
template<typename MODEL> SaddlePointVector<MODEL> &
        SaddlePointVector<MODEL>::operator-=(const SaddlePointVector & rhs) {
  *lambda_ -= *rhs.lambda_;
  *dx_     -= *rhs.dx_;
  return *this;
}
template<typename MODEL> SaddlePointVector<MODEL> &
        SaddlePointVector<MODEL>::operator*=(const double rhs) {
  *lambda_ *= rhs;
  *dx_     *= rhs;
  return *this;
}
template<typename MODEL> void SaddlePointVector<MODEL>::zero() {
  lambda_->zero();
  dx_->zero();
}
template<typename MODEL> void SaddlePointVector<MODEL>::axpy(const double zz,
                                             const SaddlePointVector & rhs) {
  lambda_->axpy(zz, *rhs.lambda_);
  dx_->axpy(zz, *rhs.dx_);
}
template<typename MODEL> double SaddlePointVector<MODEL>::dot_product_with(
                                   const SaddlePointVector & x2) const {
return dot_product(*lambda_, *x2.lambda_)
      +dot_product(*dx_, *x2.dx_);
}
```

HMatrix and HtMatrix

```
template<typename MODEL> class HMatrix : private boost::noncopyable {
  typedef typename MODEL::Increment             Increment_;
  typedef ControlIncrement<MODEL>    CtrlInc_;
  typedef CostFunction<MODEL>        CostFct_;
 public:
  explicit HMatrix(const CostFct_ & j): j_(j) {}
  void multiply(const CtrlInc_ & dx, DualVector<MODEL> & dy) const {
    PostProcessorTL<Increment_> cost;
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      cost.enrollProcessor(j_.jterm(jj).setupTL(dx));
    }

    CtrlInc_ ww(dx);
    j_.runTLM(ww, cost);

    dy.clear();
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      dy.append(cost.releaseOutputFromTL(jj));
    }
  }
 private:
  CostFct_ const & j_;
};

template<typename MODEL> class HtMatrix : private boost::noncopyable {
  typedef typename MODEL::Increment             Increment_;
  typedef CostFunction<MODEL>        CostFct_;
 public:
  explicit HtMatrix(const CostFct_ & j): j_(j) {}
  void multiply(const DualVector<MODEL> & dy, ControlIncrement<MODEL> & dx) const {
    j_.zeroAD(dx);
    PostProcessorAD<Increment_> cost;
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      cost.enrollProcessor(j_.jterm(jj).setupAD(dy.getv(jj), dx));
    }
    j_.runADJ(dx, cost);
  }
 private:
  CostFct_ const & j_;
};
```

HBHtMatrix

```
template<typename MODEL> class HBHtMatrix : private boost::noncopyable {
  typedef typename MODEL::Increment             Increment_;
  typedef ControlIncrement<MODEL>    CtrlInc_;
  typedef CostFunction<MODEL>        CostFct_;
  typedef DualVector<MODEL>          Dual_;

 public:
  explicit HBHtMatrix(const CostFct_ & j): j_(j) {}

  void multiply(const Dual_ & dy, Dual_ & dz) const {
//   Run ADJ
    CtrlInc_ ww(j_.jb());
    j_.zeroAD(ww);
    PostProcessorAD<Increment_> costad;
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      costad.enrollProcessor(j_.jterm(jj).setupAD(dy.getv(jj), ww));
    }
    j_.runADJ(ww, costad);

//   Multiply by B
    CtrlInc_ zz(j_.jb());
    j_.jb().multiplyB(ww, zz);

//   Run TLM
    PostProcessorTL<Increment_> costtl;
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      costtl.enrollProcessor(j_.jterm(jj).setupTL(zz));
    }
    j_.runTLM(zz, costtl);

//   Get TLM outputs
    dz.clear();
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      dz.append(costtl.releaseOutputFromTL(jj));
    }
  }

 private:
  CostFct_ const & j_;
};
```

SaddlePointMatrix

```
template<typename MODEL>
void SaddlePointMatrix<MODEL>::multiply(const SPVector_ & x, SPVector_ & z) const {
  CtrlInc_ ww(j_.jb());
// The three blocks below could be done in parallel
// ADJ block
  PostProcessorAD<Increment_> costad;
  j_.zeroAD(ww);
  z.dx(new CtrlInc_(j_.jb()));
  JqTermAD_ * jqad = j_.jb().initializeAD(z.dx(), x.lambda().dx());
  costad.enrollProcessor(jqad);
  for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
    costad.enrollProcessor(j_.jterm(jj).setupAD(x.lambda().getv(jj), ww));
  }
  j_.runADJ(ww, costad);
  z.dx() += ww;
// TLM block
  PostProcessorTL<Increment_> costtl;
  JqTermTL_ * jqtl = j_.jb().initializeTL();
  costtl.enrollProcessor(jqtl);
  for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
    costtl.enrollProcessor(j_.jterm(jj).setupTL(x.dx()));
  }
  j_.runTLM(x.dx(), costtl);
  z.lambda().clear();
  z.lambda().dx(new CtrlInc_(j_.jb()));
  j_.jb().finalizeTL(jqtl, x.dx(), z.lambda().dx());
  for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
    z.lambda().append(costtl.releaseOutputFromTL(jj+1));
  }
// Diagonal block
  DualVector<MODEL> diag;
  diag.dx(new CtrlInc_(j_.jb()));
  j_.jb().multiplyB(x.lambda().dx(), diag.dx());
  for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
    diag.append(j_.jterm(jj).multiplyCovar(*x.lambda().getv(jj)));
  }
// The three blocks above could be done in parallel
  z.lambda() += diag;
}
```

HessianMatrix

```
    void multiply(const CtrlInc_ & dx, CtrlInc_ & dz) const {
//  Setup TL terms of cost function
    PostProcessorTL<Increment_> costtl;
    JqTermTL_ * jqtl = j_.jb().initializeTL();
    costtl.enrollProcessor(jqtl);
    unsigned iq = 0;
    if (jqtl) iq = 1;
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      costtl.enrollProcessor(j_.jterm(jj).setupTL(dx));
    }
//  Run TLM
    j_.runTLM(dx, costtl);
//  Finalize Jb+Jq
//  Get TLM outputs, multiply by covariance inverses and setup ADJ forcing terms
    PostProcessorAD<Increment_> costad;
    dz.zero();
    CtrlInc_ dw(j_.jb());
//  Jb
    CtrlInc_ tmp(j_.jb());
    j_.jb().finalizeTL(jqtl, dx, dw);
    j_.jb().multiplyBinv(dw, tmp);
    JqTermAD_ * jqad = j_.jb().initializeAD(dz, tmp);
    costad.enrollProcessor(jqad);
    j_.zeroAD(dw);
//  Jo + Jc
    for (unsigned jj = 0; jj < j_.nterms(); ++jj) {
      boost::scoped_ptr<GeneralizedDepartures> ww(costtl.releaseOutputFromTL(iq+jj));
      boost::shared_ptr<GeneralizedDepartures> zz(j_.jterm(jj).multiplyCoInv(*ww));
      costad.enrollProcessor(j_.jterm(jj).setupAD(zz, dw));
    }
//  Run ADJ
    j_.runADJ(dw, costad);
    dz += dw;
    j_.jb().finalizeAD(jqad);
  }
```

Matrix free linear algebra in OOPS

Linear operators in mfla

Every linear operator in mfla has the form

```
class Myop {
 public:
  typedef xxx domain_type;  // e.g. xxx =  ModelIncrement
  typedef yyy codomain_type; // e.g. yyy = Departure
  Myop(...) {...}
  codomain_type operator*(const domain_type & v )  const {...}
  domain_type   leval(const codomain_type & v ) const {... }
};
```

The leval method implements the action of the adjoint (Alternative design shown later).

Linear operators in mfla

Every linear operator in mfla has the form

```
class Myop {
 public:
  typedef xxx domain_type;  // e.g. xxx =  ModelIncrement
  typedef yyy codomain_type; // e.g. yyy = Departure
  Myop(...) {...}
  codomain_type operator*(const domain_type & v )  const {...}
  domain_type   leval(const codomain_type & v ) const {... }
};
```

The leval method implements the action of the adjoint (Alternative design shown later). Vectors are linear operators. The domain is double the codomain is the vector class itself.

```
class ModelIncrement {
 public:
  typedef double domain_type;
  typedef ModelIncrement codomain_type;
  ModelIncrement(...) {...}
  codomain_type operator*(const domain_type & v )  const {...}
  domain_type   leval(const codomain_type & v ) const {... }
};
```

Here `leval` implements the inner product of the vector space.

Composition (Matrix multiplication)

```
template<class ExprT1, class ExprT2>
class Prod {
 private:
  typedef typename ExprT1::domain_type dom1;
  typedef typename ExprT2::codomain_type cod2;
  static_assert(std::is_same<dom1, cod2>::value, "domain1 != codomain2");
 public:
  typedef typename ExprT2::domain_type   domain_type;
  typedef typename ExprT1::codomain_type codomain_type;
  Prod(const ExprT1 & e1,const ExprT2 & e2) : _expr1(e1), _expr2(e2) { }
  const codomain_type operator*(const domain_type & v )  const {
    return _expr1*(_expr2*v);}
  const domain_type leval(const codomain_type & v ) const {
     return _expr2.leval(_expr1.leval(v));}
 private:
  const ExprT1 & _expr1;
  const ExprT2 & _expr2;
};

// Creator functions
template<class ExprT1, class ExprT2>
Prod<ExprT1, ExprT2> operator*(const ExprT1& e1, const ExprT2& e2) {
  return Prod<ExprT1, ExprT2>(e1, e2);}
```

Composition (Matrix Multiplication)

To allow e.g. `2.*B` and `B*2.` we use type traits

```
// General case
template <class ExprT1, class ExprT2>
struct exprTraits {
 typedef ExprT1                                 expr_type1;
 typedef ExprT2                                 expr_type2;
};

// Template specialization for the case Prod<double, ExprT2>
template <class ExprT2>
struct exprTraits<double, ExprT2> {
 typedef Scalar<typename ExprT2::codomain_type> expr_type1;
 typedef ExprT2                                 expr_type2;
};

// Template specialization for the case Prod<ExprT1, double>
template <class ExprT1>
struct exprTraits<ExprT1, double> {
  typedef ExprT1                                 expr_type1;
  typedef Scalar<typename ExprT1::domain_type>   expr_type2;
};
```

Class `Prod` is changed accordingly.

Composition (Matrix Multiplication)

To allow e.g. `2.*B` and `B*2.` we use type traits

```
// General case
template <class ExprT1, class ExprT2>
struct exprTraits {
 typedef ExprT1                                   expr_type1;
 typedef ExprT2                                   expr_type2;
};

// Template specialization for the case Prod<double, ExprT2>
template <class ExprT2>
struct exprTraits<double, ExprT2> {
 typedef Scalar<typename ExprT2::codomain_type> expr_type1;
 typedef ExprT2                                   expr_type2;
};

// Template specialization for the case Prod<ExprT1, double>
template <class ExprT1>
struct exprTraits<ExprT1, double> {
  typedef ExprT1                                   expr_type1;
  typedef Scalar<typename ExprT1::domain_type>     expr_type2;
};
```

Class `Prod` is changed accordingly.

Sum.h

```
template<class ExprT1, class ExprT2>
class Sum {
 private:
  typedef typename ExprT2::domain_type   dom2;
  typedef typename ExprT2::codomain_type cod2;
 public:
  typedef typename ExprT1::domain_type   domain_type;
  typedef typename ExprT1::codomain_type codomain_type;
  static_assert(std::is_same<domain_type, dom2>::value, "domain1 != domain2");
  static_assert(std::is_same<codomain_type, cod2>::value, "codomain1 != codoma
  Sum(const ExprT1 & e1,const ExprT2 & e2) : _expr1(e1), _expr2(e2) {}
  codomain_type operator*(const domain_type & v )  const {
    return _expr1*v + _expr2*v;
  }
  domain_type leval(const codomain_type & v ) const {
    return _expr1.leval(v)+ _expr2.leval(v);
  }
 private:
  const ExprT1 & _expr1;
  const ExprT2 & _expr2;
};

// Creator functions
template<class ExprT1, class ExprT2>
 Sum<ExprT1, ExprT2> operator+(const ExprT1& e1, const ExprT2& e2) {
 return Sum<ExprT1, ExprT2>(e1, e2);}
```

Vertcat.h

```
template<class ExprT1, class ExprT2>
class Vertcat {
 private:
  typedef typename ExprT2::domain_type    dom2;
  typedef typename ExprT1::codomain_type  codomain_type1;
  typedef typename ExprT2::codomain_type  codomain_type2;
 public:
  typedef typename ExprT1::domain_type    domain_type;
  typedef Vertcat<codomain_type1, codomain_type2> codomain_type;
  static_assert(std::is_same<domain_type, dom2>::value, "domain1 != domain2");
  Vertcat(const ExprT1 & e1,const ExprT2 & e2) : _expr1(e1), _expr2(e2) {}
  codomain_type operator*(const domain_type &v) const {
    return (_expr1*v | _expr2*v);
  }
  domain_type leval(const codomain_type &v ) const {
   return _expr1.leval(v.getexpr1()) + _expr2.leval(v.getexpr2());
  }
  const ExprT1& getexpr1() const {return _expr1;}
  const ExprT2& getexpr2() const {return _expr2;}
 private:
  const ExprT1 & _expr1;
  const ExprT2 & _expr2;
};

template<class ExprT1, class ExprT2>
Vertcat<ExprT1, ExprT2> operator|(const ExprT1& e1, const ExprT2& e2) {
  return Vertcat<ExprT1, ExprT2>(e1, e2);
}
```

Horzcat.h

```
template<class ExprT1, class ExprT2>
class Horzcat {
 private:
  typedef typename ExprT1::domain_type  domain_type1;
  typedef typename ExprT2::domain_type  domain_type2;
  typedef typename ExprT2::codomain_type  cod2;
 public:
  typedef Vertcat<domain_type1, domain_type2> domain_type;
  typedef typename ExprT1::codomain_type  codomain_type;
  static_assert(std::is_same<codomain_type, cod2>::value, "codomain1␣!=␣codoma
  Horzcat(const ExprT1 & e1,const ExprT2 & e2) : _expr1(e1), _expr2(e2) {}
  codomain_type operator*(const domain_type &v )  const {
   return _expr1*v.getexpr1()+_expr2*v.getexpr2();
  }
  domain_type leval(const codomain_type &v ) const {
   return (_expr1.leval(v) | _expr2.leval(v));
  }
  const ExprT1& getexpr1() const {return _expr1;}
  const ExprT2& getexpr2() const {return _expr2;}
 private:
  const ExprT1 & _expr1;
  const ExprT2 & _expr2;
};

template<class ExprT1, class ExprT2>
Horzcat<ExprT1, ExprT2> operator&(const ExprT1& e1, const ExprT2& e2) {
  return Horzcat<ExprT1, ExprT2>(e1, e2);
}
```

Transpose.h

```
template<class ExprT>
class Transpose {
 public:
  typedef typename ExprT::codomain_type domain_type;
  typedef typename ExprT::domain_type codomain_type;
  Transpose(const ExprT & e) : _expr(e) {}
  codomain_type operator*(const domain_type &w)   const {
    return _expr.leval(w);
  }
  domain_type   leval(const codomain_type &w)     const {
    return _expr*w;
  }
 private:
  const ExprT & _expr;
};

// Creator functions
template<class ExprT>
Transpose<ExprT> operator~(const ExprT& e) {return Transpose<ExprT>(e);}

template<class ExprT>
Transpose<ExprT> transpose(const ExprT& e) {return Transpose<ExprT>(e);}
```

Inner products and rank one matrices

Taking the transpose of a vector gives a new linear operator with domain the vector class and codomain the scalar field. In particular inner products can be written as

```
auto a = ~v*v;
```

Given two vectors v, w a rank-one matrix can be constructed as

```
auto P = v*~w;
```

This operator acts on elements in the space of w and maps to the space of v.

Inner products and rank one matrices

Taking the transpose of a vector gives a new linear operator with domain the vector class and codomain the scalar field. In particular inner products can be written as

```
auto a = ~v*v;
```

Given two vectors v, w a rank-one matrix can be constructed as

```
auto P = v*~w;
```

This operator acts on elements in the space of w and maps to the space of v. E.g. A Householder reflection is written in mfla as

```
// Construct a Householder reflection from v
Identity<Dual_> I; // Identity matrix in Dualspace
auto P = I + -2./(~v*v)*v*~v;  // Note for now we need + -2. because there
                               // is only class Sum not Diff in mfla
```

Similar for projection operators in Gram-Schmidt and also BFGS updates of the estimate of the Hessian in quasi-Newton methods.

Block matrices and composition

$$\mathbf{S} = \begin{bmatrix} \mathbf{D} & \mathbf{0} & \mathbf{L} \\ \mathbf{0} & \mathbf{R} & \mathbf{H} \\ \mathbf{L}^T & \mathbf{H}^T & \mathbf{0} \end{bmatrix}$$

```
auto S = D  & 0 & L  | 0 &  R & H | ~L & ~H & 0;
auto v = lambda | mu | dx;
auto w = S*v;
```

And

```
auto Hessian = Binv + ~H*Rinv*H;
```

- The code for `S`, `v`, `Hessian` is generated automatically at compile time.
- Straightforward to introduce new Saddle Point formulations.

Ensembles

Given vectors $x_1, \ldots, x_n \in W$ we can construct an ensemble as

```
auto X = x1 & x2 & ... & xn; X = X*1/sqrt(N-1);
```

Here $X \colon \mathbb{R}^n \to W$. We can then construct new operators

```
auto P = X*~X;
```

and

```
auto T = ~X*X;
```

Here $P \colon W \to W$ and $T \colon \mathbb{R}^n \to \mathbb{R}^n$.

Ensembles

Given vectors $x_1, \ldots, x_n \in W$ we can construct an ensemble as

```
auto X = x1 & x2 & ... & xn; X = X*1/sqrt(N-1);
```

Here $X\colon \mathbb{R}^n \to W$. We can then construct new operators

```
auto P = X*~X;
```

and

```
auto T = ~X*X;
```

Here $P\colon W \to W$ and $T\colon \mathbb{R}^n \to \mathbb{R}^n$.
Open issue

- Given an operator $A : V \to W$ should we consider a horizontal concatenation of elements V to be part of the domain.
- e.g. for operator `~X` should we consider `X` to be in the domain and compute the inner products during the construction of T. How to detect that we only need to compute the upper/lower triangular part in this case.
- Similarly for e.g.

  ```
  auto X = x1 & x2 & ... & xn;
  auto Y = H*X;
  ```

Further development for mfla

- Make all binary operators associative? (see next slide)
- Define an interface for the NL, TL and AD for each operator to simplify unit-tests.
- Automatically generate the TL and AD code?
- Ensemble of `ModelStates`. Is this ever needed? Interpretation as a (nonlinear) operator?
- Replace the observer design pattern (`PostProcessors`) in `HMatrix` etc. by composition of operators?
- (Implement Krylov and Lanczos methods, and extract duplicate code in the CG, MINRES, GMRES etc. algorithms.)

Current limitations (features?) of mfla

- Note currently

```
auto V = (v1 | v2 ) | v3;
auto W = v1 | (v2  | v3);
```

type of V is `Vertcat<Vertcat<T,T> ,T>` while type of W is `Vertcat<T,Vertcat<T,T>>` We can't do addition because of the type mismatch

Current limitations (features?) of mfla

- Note currently

  ```
  auto V = (v1 | v2 ) | v3;
  auto W = v1 | (v2  | v3);
  ```

  type of V is `Vertcat<Vertcat<T,T> ,T>` while type of W is `Vertcat<T,Vertcat<T,T>>` We can't do addition because of the type mismatch
- Also for matrices `A & ~H | H & C` has a different type than `(A | H) & (~H | C)`

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \begin{bmatrix} A \\ H \end{bmatrix} & \begin{bmatrix} H^T \\ C \end{bmatrix} \end{bmatrix}$$

Current limitations (features?) of mfla

- Note currently

```
auto V = (v1 | v2 ) | v3;
auto W = v1 | (v2  | v3);
```

  type of V is `Vertcat<Vertcat<T,T> ,T>` while type of W is `Vertcat<T,Vertcat<T,T>>` We can't do addition because of the type mismatch

- Also for matrices `A & ~H | H & C` has a different type than `(A | H) & (~H | C)`

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \begin{bmatrix} A \\ H \end{bmatrix} & \begin{bmatrix} H^T \\ C \end{bmatrix} \end{bmatrix}$$

- Both representations act on vectors `auto xvy = x | y` but they differ internally

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + H^T y \\ Hx + Cy \end{bmatrix} \qquad \begin{bmatrix} A \\ H \end{bmatrix} x + \begin{bmatrix} H^T \\ C \end{bmatrix} y = \begin{bmatrix} Ax \\ Hx \end{bmatrix} + \begin{bmatrix} H^T y \\ Cy \end{bmatrix}$$

Current limitations (features?) of mfla

- Note currently

```
auto V = (v1 | v2 ) | v3;
auto W = v1 | (v2  | v3);
```

  type of V is `Vertcat<Vertcat<T,T> ,T>` while type of W is `Vertcat<T,Vertcat<T,T>>` We can't do addition because of the type mismatch

- Also for matrices `A & ~H | H & C` has a different type than `(A | H) & (~H | C)`

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \begin{bmatrix} A \\ H \end{bmatrix} & \begin{bmatrix} H^T \\ C \end{bmatrix} \end{bmatrix}$$

- Both representations act on vectors `auto xvy = x | y` but they differ internally

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + H^T y \\ Hx + Cy \end{bmatrix} \qquad \begin{bmatrix} A \\ H \end{bmatrix} x + \begin{bmatrix} H^T \\ C \end{bmatrix} y = \begin{bmatrix} Ax \\ Hx \end{bmatrix} + \begin{bmatrix} H^T y \\ Cy \end{bmatrix}$$

- Should we choose a single representation for block matrices in mfla or is the possibility to have some control over the internal expansion useful feature?

Current limitations (features?) of mfla

- Note currently

```
auto V = (v1 | v2 ) | v3;
auto W = v1 | (v2  | v3);
```

type of V is `Vertcat<Vertcat<T,T> ,T>` while type of W is `Vertcat<T,Vertcat<T,T>>` We can't do addition because of the type mismatch

- Also for matrices `A & ~H | H & C` has a different type than `(A | H) & (~H | C)`

$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \end{bmatrix} \qquad \begin{bmatrix} \begin{bmatrix} A \\ H \end{bmatrix} & \begin{bmatrix} H^T \\ C \end{bmatrix} \end{bmatrix}$$

- Both representations act on vectors `auto xvy = x | y` but they differ internally

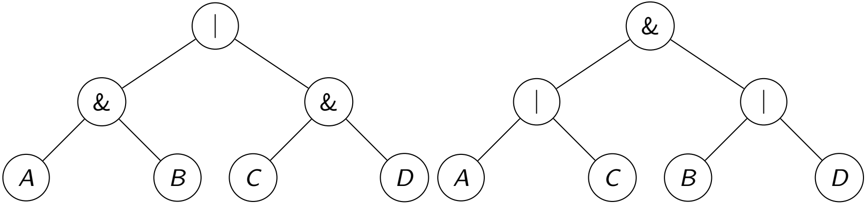
$$\begin{bmatrix} \begin{bmatrix} A & H^T \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} H & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + H^T y \\ Hx + Cy \end{bmatrix} \qquad \begin{bmatrix} A \\ H \end{bmatrix} x + \begin{bmatrix} H^T \\ C \end{bmatrix} y = \begin{bmatrix} Ax \\ Hx \end{bmatrix} + \begin{bmatrix} H^T y \\ Cy \end{bmatrix}$$

- Should we choose a single representation for block matrices in mfla or is the possibility to have some control over the internal expansion useful feature?
- The second representation can currently not act on xvy because we deduce that the `codomain_type` of the Block matrix is `Vertcat<X,Y>` but the `operator+` returns a type `Sum<Vertcat<X,Y>,<Vertcat<X,Y>>` which is not convertible to `Vertcat<X,Y>`

Parallelism in mfla

S1 S2

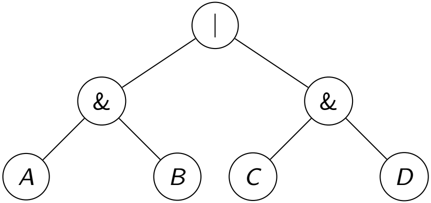
$$\begin{bmatrix} \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Dy \end{bmatrix} \qquad \begin{bmatrix} A \\ C \end{bmatrix} x + \begin{bmatrix} B \\ D \end{bmatrix} y = \begin{bmatrix} Ax \\ Cx \end{bmatrix} + \begin{bmatrix} By \\ Dy \end{bmatrix}$$

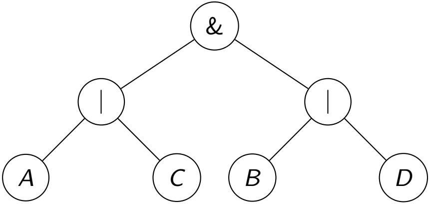


Parallelism in mfla

S1 S2

$$\begin{bmatrix} \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Dy \end{bmatrix} \qquad \begin{bmatrix} A \\ C \end{bmatrix} x + \begin{bmatrix} B \\ D \end{bmatrix} y = \begin{bmatrix} Ax \\ Cx \end{bmatrix} + \begin{bmatrix} By \\ Dy \end{bmatrix}$$





Let $t_A \oplus t_B = \max(t_A, t_B)$ and $t_A \otimes t_B = t_A + t_B$. t_W cost of vector addition in codomain of A and B and t_V cost of vector addition in codomain of C and D

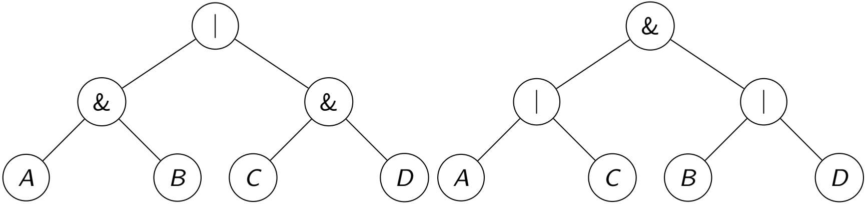
$$t_{S1} = t_W \otimes (t_A \oplus t_B) \oplus t_V \otimes (t_C \oplus t_D) \leq (t_W \oplus t_V) \otimes ((t_A \oplus t_C) \oplus (t_B \oplus t_D)) = t_{S2}$$

with equality iff $t_A = t_B = t_C = t_D$. Showing that $S1$ is never less efficient than $S2$.

Parallelism in mfla

S1 S2

$$\begin{bmatrix} \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \begin{bmatrix} C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Dy \end{bmatrix} \qquad \begin{bmatrix} A \\ C \end{bmatrix} x + \begin{bmatrix} B \\ D \end{bmatrix} y = \begin{bmatrix} Ax \\ Cx \end{bmatrix} + \begin{bmatrix} By \\ Dy \end{bmatrix}$$



Let $t_A \oplus t_B = \max(t_A, t_B)$ and $t_A \otimes t_B = t_A + t_B$. t_W cost of vector addition in codomain of A and B and t_V cost of vector addition in codomain of C and D

$$t_{S1} = t_W \otimes (t_A \oplus t_B) \oplus t_V \otimes (t_C \oplus t_D) \leq (t_W \oplus t_V) \otimes ((t_A \oplus t_C) \oplus (t_B \oplus t_D)) = t_{S2}$$

with equality iff $t_A = t_B = t_C = t_D$. Showing that $S1$ is never less efficient than $S2$.

- Replacing `|` by `&` for the adjoint will not be optimal.

Signature of the TL and AD operators

<table>
<tr><td></td><td><code>void Htl(const X& x, Y& y)</code>

<code>Htl(x,y)</code>
$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I & 0 \\ H & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$</td><td><code>Y Htl(const X& x);</code>

<code>auto y = Htl(x);</code>
$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I \\ H \end{bmatrix} \begin{bmatrix} x \end{bmatrix}$</td><td><code>// Y Htl(X&& x)</code>
<code>auto y = Htl(x);</code>
<code>// auto y = Htl(f())</code>
$\begin{bmatrix} y \end{bmatrix} = \begin{bmatrix} H \end{bmatrix} \begin{bmatrix} x \end{bmatrix}$</td></tr>
<tr><td>TL</td><td></td><td></td><td></td></tr>
<tr><td>AD</td><td>$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I & Ht \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

<code>void Had(X& x, Y& y)</code></td><td>$\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} I & Ht \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
<code>void Had(X& x, Y&& y);</code>
<code>//X Had(Y&& y);</code>
<code>//x += Had(move(y));</code></td><td>$\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} Ht \end{bmatrix} \begin{bmatrix} y \end{bmatrix}$

<code>X Had(Y&& y);</code></td></tr>
</table>

- Stroustrup: return a result as a return value rather than modifying an object through an argument.
- Option 1) In the adjoint we set y to zero but memory can not be deallocated. An "unnecessary" addition in adjoint code for every function call[2]
- Option 2) still requires pass-by-reference in the adjoint
- Option 2+3) Copy assignment needs to be `=delete` for all objects (to avoid `y=Htl(x)`)
- Option 3) `X&&` is not allowed to be a deduced type[3]. Introduce unit-tests for this.

[2] How does this overhead affect the speed of the adjoint?

[3] See https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

Open issues: lvalues, prvalues, xvalues, copy/move-assignment, copy/move-constructor, copy/move elision

Copy construction[4]	Copy-assignment	Move construction	Move-assignment
`T a = b;`	`a = b;`	`T a=std::move(b);`	`a=std::move(b);`
$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} b \end{bmatrix}$	$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$	$\begin{bmatrix} a \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} b \end{bmatrix}$	$\begin{bmatrix} a \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$
$\begin{bmatrix} \tilde{b} \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{a} \\ \tilde{b} \end{bmatrix}$	$\begin{bmatrix} \tilde{a} \\ \tilde{b} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{a} \\ \tilde{b} \end{bmatrix}$	$\begin{bmatrix} \tilde{b} \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} \tilde{a} \end{bmatrix}$	$\begin{bmatrix} \tilde{a} \\ \tilde{b} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} \tilde{a} \end{bmatrix}$
`a += b;` `b =std::move(a);`[5]	`b += a;` `a=0;`	`T b=std::move(a);`	`T b=a; a=0;`

- $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$. `b=a+b;` or `b+=a;`. Note we have $\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$.
- `void swap( T& a, T& b );` $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$
- Destructor `~b;` $\begin{bmatrix} a \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$ adjoint `b=0;`

---

[4] With automatic storage duration

[5] Note that simply `b = a + b` would not release the resources held by `a` at the correct time. Although the destructor would get called when `a` goes out of scope

Reshaping

There is an invertible linear transformation G that maps horizontal concatenations of vectors $v_i \in V$ to vertical concatenations.

$$G : V^n \to V^n, \quad \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \mapsto \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

For clarity distinguish the domain and codomain

$$G : \mathrm{Lin}(\mathbb{R}^n, V) \to \mathrm{Lin}(\mathbb{R}, V^n), \quad \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \mapsto \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Reshaping

There is an invertible linear transformation G that maps horizontal concatenations of vectors $v_i \in V$ to vertical concatenations.

$$G : V^n \to V^n, \quad \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \mapsto \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

For clarity distinguish the domain and codomain

$$G : \mathrm{Lin}(\mathbb{R}^n, V) \to \mathrm{Lin}(\mathbb{R}, V^n), \quad \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \mapsto \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

For operators $A_i : W \to V$

$$G : \mathrm{Lin}(W^n, V) \to \mathrm{Lin}(W, V^n), \quad \begin{bmatrix} A_1 & A_2 & \dots & A_n \end{bmatrix} \mapsto \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix}$$

Iterating

Given a linear operator $A : V \to V$. We define the nonlinear operator

$$\begin{aligned} \mathit{iterate}(n) : \mathrm{Lin}(V, V) &\to \mathrm{Lin}(V, V^n) \\ A &\mapsto \begin{bmatrix} I \\ A \\ \vdots \\ A^{n-1} \end{bmatrix} \end{aligned}$$

Iterating

Given a linear operator $A : V \to V$. We define the nonlinear operator

$$\begin{aligned} \mathit{iterate}(n) : \mathrm{Lin}(V, V) &\to \mathrm{Lin}(V, V^n) \\ A &\mapsto \begin{bmatrix} I \\ A \\ \vdots \\ A^{n-1} \end{bmatrix} \end{aligned}$$

Also the nonlinear operator

$$\begin{aligned} \mathit{normalize} : V &\to V \\ v &\mapsto \frac{1}{\sqrt{v^T v}} v \end{aligned}$$

Naive Krylov methods

Given a linear operator $A : V \to V$ we can construct a new linear operator

$$\begin{aligned} F : V &\to \mathrm{Lin}(\mathbb{R}^n, V), \\ v &\mapsto \begin{bmatrix} v & Av & A^2v & \dots & A^n v \end{bmatrix} \end{aligned}$$

then we can generate

```
auto r = b + B*~H*Rinv*d; // b = xb-xg , d = y - H(xg)
auto A = I + B*~H*Rinv*H;
auto F = Ginv*iterate(A,n);  // or   Ginv*iterate(normalize*A,n);
auto K = F*r;
```

Naive Krylov methods

Given a linear operator $A : V \to V$ we can construct a new linear operator

$$
\begin{aligned}
F : V &\to \mathrm{Lin}(\mathbb{R}^n, V), \\
v &\mapsto \begin{bmatrix} v & Av & A^2 v & \dots & A^n v \end{bmatrix}
\end{aligned}
$$

then we can generate

```
auto r = b + B*~H*Rinv*d; // b = xb-xg , d = y - H(xg)
auto A = I + B*~H*Rinv*H;
auto F = Ginv*iterate(A,n);  // or   Ginv*iterate(normalize*A,n);
auto K = F*r;
```

Note that the Krylov subspace is itself a linear operator $K : \mathbb{R}^n \to V$ If we have a function object for the cost function $J : V \to \mathbb{R}$ we should be able to do composition

```
auto JK = J * K;     // J o K: R^n --> R,  v -->  J(K*v)
```

Here $JK : \mathbb{R}^n \to \mathbb{R}$.
Given an ensemble X we should be able to do

```
auto JKX = J * (K & X);
```

To search for the minimum of J restricted to the combined Krylov and ensemble space.

Summary

- mfla allows composition, addition, horizontal and vertical concatenation and keeps track of the adjoint for each TL.
- Code for e.g. block matrices (saddle point formulations), `DualVectors`, `Hessian` and ensembles can be generated automatically at compile time
- Ease of composition is essential to get flexible code.

Summary

- mfla allows composition, addition, horizontal and vertical concatenation and keeps track of the adjoint for each TL.
- Code for e.g. block matrices (saddle point formulations), `DualVectors`, `Hessian` and ensembles can be generated automatically at compile time
- Ease of composition is essential to get flexible code.
- Open issues
  - Can we impose the single input, single output everywhere?
  - Can we exclude copy assignment (and copy construction) for all objects?
  - how to model the relation between NL and TL/AD
  - How to handle linearization state of operators
  - Automatically generate TL/AD code at compile time?
  - C++11 in OOPS (use of `auto`, move constructors, rvalue references)
  - Which decisions can be made at compile time to simplify the code (avoid unnecessary creation of templated code), e.g. the DA-formulation, the minimization algorithm, model resolution?

Side effects (IO, diagnostics). Note NL/TL/AD in a single object here

```
template<class dom>
struct Statewriter {
  typedef dom domain_type;
  typedef dom codomain_type;
  Statewriter(std::ostream & osnl,std::ostream & ostl, std::ostream &  osad) :
                _osnl(osnl) , _ostl(ostl), _osad(osad) { }
  codomain_type operator()(domain_type x) const {          //
    _osnl << x << "\n"; return x; }
  codomain_type tl(domain_type, domain_type   dx) const { // was operator*
    _ostl << dx << "\n"; return dx; }
  domain_type   ad(domain_type, codomain_type dy) const { // was leval
    _osad << dy << "\n"; return dy; }
 private:
  std::ostream & _osnl,
  std::ostream & _ostl,
  std::ostream & _osad;
};
```

Side effects (IO, diagnostics). Note NL/TL/AD in a single object here

```
template<class dom>
struct Statewriter {
  typedef dom domain_type;
  typedef dom codomain_type;
  Statewriter(std::ostream & osnl,std::ostream & ostl, std::ostream &  osad) :
                _osnl(osnl) , _ostl(ostl), _osad(osad) { }
  codomain_type operator()(domain_type x) const {          //
    _osnl << x << "\n"; return x; }
  codomain_type tl(domain_type, domain_type   dx) const { // was operator*
    _ostl << dx << "\n"; return dx; }
  domain_type   ad(domain_type, codomain_type dy) const { // was leval
    _osad << dy << "\n"; return dy; }
 private:
  std::ostream & _osnl,
  std::ostream & _ostl,
  std::ostream & _osad;
};

int main() {
//  ...
  Propagator          M( ...);
  std::ofstream       osnl("nltraj.txt");
  std::stringstream   osad;     // Or /dev/null implementation
  Statewriter<State> W(osnl, std::cout, osad);
  auto  WM  = W*M;
  auto  WM4 = WM*WM*WM*WM;
};
```

- Other side effects (e.g. canonical injections into Fortran arrays, or diagnostics) should use a similar construction.

Design of objects in OOPS: Interpolate

Current

```
class QgFields {
// Interpolate to given location
  void interpolate  (const LocQG &, GomQG &)        const;
  void interpolateTL(const LocQG &, GomQG &)        const;
  void interpolateAD(const LocQG &, const GomQG &);
// etc
};
```

This signature suggests that $\mathit{interpolate} : \mathit{LocQG} \rightarrow \mathit{GomQG}$. Note AD is a non-const.

Design of objects in OOPS: Interpolate

Current

```
class QgFields {
// Interpolate to given location
  void interpolate  (const LocQG &, GomQG &)        const;
  void interpolateTL(const LocQG &, GomQG &)        const;
  void interpolateAD(const LocQG &, const GomQG &);
// etc
};
```

This signature suggests that *interpolate* : $LocQG \rightarrow GomQG$. Note AD is a non-const.
Instead move interpolation to LocQG $LocQg : QgField \rightarrow GomQg$

```
class LocQG {
  void interpolate  (const Qgfields &, GomQG &)       const;
  void interpolateTL(const Qgfields &, GomQG &)       const;
  void interpolateAD(QgFields       &, const GomQG &) const;
};
```

All operators are now const member function. Const GomQG?

Design of objects in OOPS: Interpolate

Current

```
class QgFields {
// Interpolate to given location
  void interpolate  (const LocQG &, GomQG &)        const;
  void interpolateTL(const LocQG &, GomQG &)        const;
  void interpolateAD(const LocQG &, const GomQG &);
// etc
};
```

This signature suggests that *interpolate* : *LocQG* $\rightarrow$ *GomQG*. Note AD is a non-const.
Instead move interpolation to LocQG *LocQg* : *QgField* $\rightarrow$ *GomQg*

```
class LocQG {
  void interpolate  (const Qgfields &, GomQG &)       const;
  void interpolateTL(const Qgfields &, GomQG &)       const;
  void interpolateAD(QgFields       &, const GomQG &) const;
};
```

All operators are now const member function. Const GomQG?
Perhaps treat interpolation as a “first class citizen”.

```
class Interpolate {
  Interpolate(LocQG locQG) : _locQG(locQG)  { }
  GomQG    NL(const Qgfields &) const;
  GomQG    TL(const Qgfields &) const;
  QgFields AD(const GomQG &)    const;
};
```

Design of objects in OOPS: Derivatives and interpolation

$$LocQg\colon QgField \to GomQg$$

```
class LocQG {
  GomQG    NL(const Qgfields &) const;  // Interpolation
  GomQG    TL(const Qgfields &) const;  // TL of Interpolation
  QgFields AD(const GomQG &)    const;
};
```

$$QgField\colon LocQg \to GomQg$$

```
class QgField {
  GomQG    NL(const LocQG &) const; // Function evaluation
  GomQG    TL(const LocQG &) const; // The spatial derivative at a point
  LocQg    AD(const GomQG &) const; // We need linearization points here
};
```

Design of objects in OOPS: Derivatives and interpolation

$$LocQg : QgField \rightarrow GomQg$$

```
class LocQG {
  GomQG    NL(const Qgfields &) const;  // Interpolation
  GomQG    TL(const Qgfields &) const;  // TL of Interpolation
  QgFields AD(const GomQG &)    const;
};
```

$$QgField : LocQg \rightarrow GomQg$$

```
class QgField {
  GomQG    NL(const LocQG &) const; // Function evaluation
  GomQG    TL(const LocQG &) const; // The spatial derivative at a point
  LocQg    AD(const GomQG &) const; // We need linearization points here
};
```

- Is this "Duality" between Functions and Domains something general?

Design of objects in OOPS: Derivatives and interpolation

$$LocQg : QgField \to GomQg$$

```
class LocQG {
  GomQG    NL(const Qgfields &) const;  // Interpolation
  GomQG    TL(const Qgfields &) const;  // TL of Interpolation
  QgFields AD(const GomQG &)    const;
};
```

$$QgField : LocQg \to GomQg$$

```
class QgField {
  GomQG    NL(const LocQG &) const; // Function evaluation
  GomQG    TL(const LocQG &) const; // The spatial derivative at a point
  LocQg    AD(const GomQG &) const; // We need linearization points here
};
```

- Is this "Duality" between Functions and Domains something general?
- To handle both “views” should we implement $QgField \times LocQg \to GomQg$ and use currying
- E.g. if `QgField qgfield;` and `LocQg logqg;` ($qgfield \in QgField$ and $logqg \in LocQg$)
- Then $qgfield : LocQg \to GomQg$ and $locqg : QgField \to GomQg$

Composition and automatic compile time differentiation

```
SUBROUTINE f(x,y)
  ! f: x -> y
  CALL h(x,z)
  CALL g(z,y)
END
```

```
SUBROUTINE ftl(x,dx,dy)
  CALL h(x,z)
  CALL htl(x,dx,dz)
  CALL gtl(z,dz,dy)
END
```

```
SUBROUTINE fad(x,dy,dx)
  CALL h(x,z)
  CALL gad(z,dy,dz)
  CALL had(x,dz,dx)
END
```

Composition and automatic compile time differentiation

```
SUBROUTINE f(x,y)
  ! f: x -> y
  CALL h(x,z)
  CALL g(z,y)
END
```

```
SUBROUTINE ftl(x,dx,dy)
  CALL h(x,z)
  CALL htl(x,dx,dz)
  CALL gtl(z,dz,dy)
END
```

```
SUBROUTINE fad(x,dy,dx)
  CALL h(x,z)
  CALL gad(z,dy,dz)
  CALL had(x,dz,dx)
END
```

```
template<class G, class H>
class Prod {
 public:
  typedef typename H::domain_type   domain_type;
  typedef typename G::codomain_type codomain_type;
  Prod(const G & g,const H & h) : _g(g), _h(h) { }
  codomain_type operator()(domain_type x) const {
    return _g(_h(x));
  }
  codomain_type tl(const domain_type & x,domain_type  dx) const {
    return _g.tl(_h(x),_h.tl(x,dx));
  }
  domain_type ad(const domain_type & x,codomain_type  dy) const {
    return _h.ad(x,_g.ad(_h(x),dy));
  }
 private:
  const G _g;
  const H _h;
};
```

Composition and automatic compile time differentiation

```
SUBROUTINE f(x,y)
  ! f: x -> y
  CALL h(x,z)
  CALL g(z,y)
END
```

```
SUBROUTINE ftl(x,dx,dy)
  CALL h(x,z)
  CALL htl(x,dx,dz)
  CALL gtl(z,dz,dy)
END
```

```
SUBROUTINE fad(x,dy,dx)
  CALL h(x,z)
  CALL gad(z,dy,dz)
  CALL had(x,dz,dx)
END
```

```
template<class G, class H>
class Prod {
 public:
  typedef typename H::domain_type   domain_type;
  typedef typename G::codomain_type codomain_type;
  Prod(const G & g,const H & h) : _g(g), _h(h) { }
  codomain_type operator()(domain_type x) const {
    return _g(_h(x));
  }
  codomain_type tl(const domain_type & x,domain_type  dx) const {
    return _g.tl(_h(x),_h.tl(x,dx));
  }
  domain_type ad(const domain_type & x,codomain_type  dy) const {
    return _h.ad(x,_g.ad(_h(x),dy));
  }
 private:
  const G _g;
  const H _h;
};
```

- Not optimal for $k \circ f = k \circ (g \circ h)$ but $(k \circ g) \circ h$ is fine if h is "elementary".
- With addition: $k \circ (g + h)$ is fine but $k \circ (g + h \circ p)$ is not optimal

Composition and automatic compile time differentiation

```
template<class G, class H>
class Prod {
 public:
  typedef typename H::domain_type   domain_type;
  typedef typename G::domain_type   Gdomain_type;
  typedef typename G::codomain_type codomain_type;
  Prod(const G & g,const H & h) : _g(g), _h(h), _x(0), _y(0) { }
  codomain_type operator()(domain_type x) const {
    _x = x;
    _y = _h(x);
    return _g(_y);
  }
  codomain_type tl(domain_type dx) const {
    return _g.tl(_y,_h.tl(_x,dx));
  }
  domain_type ad(codomain_type dy) const {
    return _h.ad(_x,_g.ad(_y,dy));
  }
 private:
  domain_type  _x;
  Gdomain_type _y;
  const G _g;
  const H _h;
};
```

Here `_x` and `_y` only get initialized after the call to `operator()`

```
template<class G, class H>
class Composition {
 public:
  typedef typename H::domain_type   domain_type;
  typedef typename G::codomain_type codomain_type;
  Composition(const G & g,const H & h) : _g(g), _h(h) { }
  auto operator()(domain_type x) const {
    return g(h(x));
  }
  auto derivative(domain_type x) const {
    return TL<G>(_g,_h(x))*TL<H>(_h,x);
  }
 private:
  const G _g;
  const H _h;
};
```

Restructuring the Hessian

$$
\left\|
\left[\begin{array}{ccccc}
\mathrm{I} & & & & \\
-\mathrm{M}_1 & \mathrm{I} & & & \\
 & \ddots & \ddots & & \\
 & & -\mathrm{M}_{N-1} & & \mathrm{I} \\
\hline
\mathrm{H}_0 & & & & \\
 & \mathrm{H}_1 & & & \\
 & & \ddots & & \\
 & & & & \mathrm{H}_{N-1}
\end{array}\right]
\begin{bmatrix} \delta x_0 \\ \delta x_1 \\ \vdots \\ \delta x_{N-1} \end{bmatrix}
-
\left[\begin{array}{c} b_0 \\ b_1 \\ \vdots \\ b_{N-1} \\ \hline d_0 \\ d_1 \\ \vdots \\ d_{N-1} \end{array}\right]
\right\|_{\mathrm{diag}(D^{-1},R^{-1})}
\tag{1}
$$

Reordering the rows gives

$$
\left\|
\begin{bmatrix}
\mathrm{I} & & & & & \\
\mathrm{H}_0 & & & & & \\
-\mathrm{M}_1 & \mathrm{I} & & & & \\
 & \mathrm{H}_1 & & & & \\
 & -\mathrm{M}_2 & \mathrm{I} & & & \\
 & & \mathrm{H}_2 & & & \\
 & & \ddots & \ddots & & \\
 & & & -\mathrm{M}_{N-1} & & \mathrm{I} \\
 & & & & & \mathrm{H}_{N-1}
\end{bmatrix}
\begin{bmatrix} \delta x_0 \\ \delta x_1 \\ \vdots \\ \delta x_{N-1} \end{bmatrix}
-
\begin{bmatrix} b_0 \\ d_0 \\ b_1 \\ d_1 \\ \vdots \\ b_{N-1} \\ d_{N-1} \end{bmatrix}
\right\|_{X2}
\tag{2}
$$

GOM+ control variable (flexibility of the OOPS code)

GOM+-arrays as control variable. Interpolation as a strong constraint

The weak constraint 4D-VAR cost function can be written as (4D-vector notation)

$$J(\mathbf{x}) = \frac{1}{2}\|\mathcal{H}(\mathcal{V}(\mathbf{x})) - \mathbf{y}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathcal{L}(\mathbf{x}) - \mathbf{p}^b\|^2_{\mathbf{D}^{-1}} = \tilde{J}_o(\mathbf{x}) + J_q(\mathbf{x})$$

Here $\mathcal{V}(\mathbf{x})$ is the mapping from 4D model states to 4D-GOM+-arrays.

GOM+-arrays as control variable. Interpolation as a strong constraint

The weak constraint 4D-VAR cost function can be written as (4D-vector notation)

$$J(\mathbf{x}) = \frac{1}{2}\|\mathcal{H}(\mathcal{V}(\mathbf{x})) - \mathbf{y}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathcal{L}(\mathbf{x}) - \mathbf{p}^b\|^2_{\mathbf{D}^{-1}} = \tilde{J}_o(\mathbf{x}) + J_q(\mathbf{x})$$

Here $\mathcal{V}(\mathbf{x})$ is the mapping from 4D model states to 4D-GOM+-arrays.

$$J(\mathbf{x}, \mathbf{z}) = \frac{1}{2}\|\mathcal{H}(\mathbf{z}) - \mathbf{y}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathcal{L}(\mathbf{x}) - \mathbf{p}^b\|^2_{\mathbf{D}^{-1}} = J_o(z) + J_q(x)$$

subject to $\mathcal{V}(\mathbf{x}) - \mathbf{z} = \mathbf{0}$.

GOM+-arrays as control variable. Interpolation as a strong constraint

The weak constraint 4D-VAR cost function can be written as (4D-vector notation)

$$J(\mathbf{x}) = \frac{1}{2}\|\mathcal{H}(\mathcal{V}(\mathbf{x})) - \mathbf{y}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathcal{L}(\mathbf{x}) - \mathbf{p}^b\|^2_{\mathbf{D}^{-1}} = \tilde{J}_o(\mathbf{x}) + J_q(\mathbf{x})$$

Here $\mathcal{V}(\mathbf{x})$ is the mapping from 4D model states to 4D-GOM+-arrays.

$$J(\mathbf{x}, \mathbf{z}) = \frac{1}{2}\|\mathcal{H}(\mathbf{z}) - \mathbf{y}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathcal{L}(\mathbf{x}) - \mathbf{p}^b\|^2_{\mathbf{D}^{-1}} = J_o(z) + J_q(x)$$

subject to $\mathcal{V}(\mathbf{x}) - \mathbf{z} = \mathbf{0}$.

Incremental formulation with GOM+-arrays as control variable

$$\left(\mathbf{I} + \mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\right)\delta\mathbf{z} = \mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d} + \mathbf{V}\mathbf{L}^{-1}\mathbf{b} \tag{3}$$

- This a similar to a 1D-VAR retrieval but with $\mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T$ as the background error covariance in GOM+-space. Note

  $$\delta\mathbf{x} = \mathbf{L}^{-1}\mathbf{b} - \mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T\mathbf{H}^T\mathbf{R}^{-1}(\mathbf{H}\delta\mathbf{z} - \mathbf{d})$$

  No need for a seperate 4D-VAR to assimilate the retrievals.
- Note that $\mathbf{H}$ and $\mathbf{H}^T$ are linearized around a guess $\mathbf{z}^g$ and we could update the linearization trajectories for the obs op without running the nonlinear model. E.g. we could update the linearization trajectory for J_o more often if it is expected that nonlinearities in $\mathcal{H}$ are more important than those in $\mathcal{L}$. This looks similar to the double inner loop implementation at the UK Met Office.

GOM+-arrays as control variable. Interpolation as a weak constraint

Incremental weak constraint 4D-VAR cost function with weak constraint interpolation

$$J(\delta\mathbf{x}, \delta\mathbf{z}) = \frac{1}{2}\|\mathbf{H}\delta\mathbf{z} - \mathbf{d}\|^2_{\mathbf{R}^{-1}} + \frac{1}{2}\|\mathbf{L}\delta\mathbf{x} - \mathbf{b}\|^2_{\mathbf{D}^{-1}} + \frac{1}{2}\|\mathbf{V}\delta\mathbf{x} - \delta\mathbf{z}\|^2_{\mathbf{T}^{-1}} \tag{4}$$

$$(\mathbf{I} + (\mathbf{T} + \mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T)\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\delta\mathbf{z} = \mathbf{V}\mathbf{L}^{-1}\mathbf{b} + (\mathbf{T} + \mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T)\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d} \tag{5}$$

Showing that the weak constraint formulation is obtained from the strong constraint formulation (eq (3)) by replacing the background error covariance in GOM-space $\mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T$ by $\mathbf{T} + \mathbf{V}\mathbf{L}^{-1}\mathbf{D}\mathbf{L}^{-T}\mathbf{V}^T$

Alternatively we can formulate the problem in block matrix form

$$\begin{bmatrix} \mathbf{T}^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H} & -\mathbf{T}^{-1}\mathbf{V} \\ -\mathbf{V}^T\mathbf{T}^{-1} & \mathbf{L}^T\mathbf{D}^{-1}\mathbf{L} + \mathbf{V}^T\mathbf{T}^{-1}\mathbf{V} \end{bmatrix} \begin{bmatrix} \delta\mathbf{z} \\ \delta\mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{H}^T\mathbf{R}^{-1}\mathbf{d} \\ \mathbf{L}^T\mathbf{D}^{-1}\mathbf{b} \end{bmatrix}$$

For fixed δx the top block row is a 1D-VAR retrieval (parallel). But we avoid here using the background twice

High resolution adjoint in gradient

4D-VAR ($\mathbf{x}^b = \mathbf{x}^g$)

$$(\mathbf{I} + \mathbf{B}\mathbf{M}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{M})\delta\mathbf{x}_0 = \mathbf{B}\mathbf{M}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

3D-FGAT

$$(\mathbf{I} + \mathbf{B}\mathbf{I}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{I})\delta\mathbf{x}_{T/2} = \mathbf{B}\mathbf{I}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

3D-FGAT (with 4D-VAR gradient)

$$(\mathbf{I} + \mathbf{B}\mathbf{I}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{I})\delta\mathbf{x}_0 = \mathbf{B}\mathbf{M}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}$$

- For this we need flexibility (ease of composition, addition etc) on the low-level objects instead of "high level" objects like a `CostFunction`

Varbc

Varbc

$$J(\delta x, \delta\beta) = \frac{1}{2} \left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} - \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{H} & \mathbf{P} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\beta \end{bmatrix} \right\|^2_{\tilde{\mathbf{B}}^{-1}}$$

with $\tilde{\mathbf{B}}^{-1} = \mathrm{diag}(\mathbf{D}^{-1}, \mathbf{B}_\beta^{-1}, \mathbf{R}^{-1})$

Varbc

$$J(\delta x, \delta\beta) = \frac{1}{2}\left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} - \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{H} & \mathbf{P} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\beta \end{bmatrix} \right\|^2_{\tilde{\mathbf{B}}^{-1}}$$

with $\tilde{\mathbf{B}}^{-1} = \mathrm{diag}(\mathbf{D}^{-1}, \mathbf{B}_\beta^{-1}, \mathbf{R}^{-1})$

$$J(\delta\mathbf{x}, \delta\beta) = \frac{1}{2}\|\mathbf{b} - \mathbf{L}\delta\mathbf{x}\|^2_{\mathbf{D}^{-1}} + \frac{1}{2}\|\mathbf{c} - \delta\beta\|^2_{\mathbf{B}_\beta^{-1}} + \frac{1}{2}\|\mathbf{d} - \mathbf{H}\delta\mathbf{x} - \mathbf{P}\delta\beta\|^2_{\mathbf{R}^{-1}}$$

We need to interface Fortran subroutines for $\mathbf{P}$ and $\mathbf{B}_\beta$ and introduce a new type for β and $\delta\beta$. Can we separate the varbc code from hop?