Matrix free linear algebra in OOPS

Roel Stappers

[1]Norwegian Meteorological Institute
roels@met.no

26th ALADIN Wk & HIRLAM ASM
Lisbon
4–7 April 2016

- Formulations of DA and flexibility in OOPS
- Current OOPS implementation
  - `Departures`, `ControlIncrement`, `Increment4D`, `DualVectors`, `SaddlePointVector`, etc
  - `HMatrix`, `HBHtMatrix`, `HessianMatrix` `SaddlePointMatrix` etc.
- Matrix free linear algebra in OOPS

Formulations of DA and flexibility in OOPS

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

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

Formulations of DA and flexibility in OOPS

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

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

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)

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

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

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 that `class HMatrix` in OOPS maps `ControlIncrement` to `DualVector`

Current OOPS implementation

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

ControlIncrement is a container for Increment4D, ModelAuxIncrement, ObsAuxIncrement

```
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) {
```

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

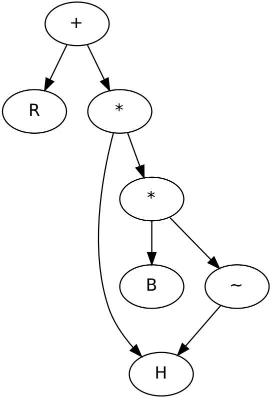
```
    void multiply(const CtrlInc_ & dx, CtrlInc_ & dz) const {
//  Setup TL terms of cost function
    PostProcessorTL<Increment_> costtl;
    JqTermTL_ * jqtl = j_.jb().initializeTL();
    costtl.enrollProcessor(jqtl);
    unsigned iq = 0;
    if (jqtl) iq = 1;
    for (unsigned jj = 0; jj < j_.nterms(); ++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

Expression tree for $R + HBH^T$

- All leaf nodes are interfaces to the Fortran code. We have implementations of the adjoint for these.
- Every linear operator knows it domain and codomain. During construction of a new linear operator (a new node), e.g. `auto BHt=B*~H;` we do a static assert (compile time) that the domain and codomain are consistent.
- Vectors are treated as linear operators. The domain is the scalar field (typically `double`). The codomain is the vector space itself.
- Construction of a new operator also constructs the adjoint.
- We have a separation between the construction of linear operators and the application of the operators to vectors.

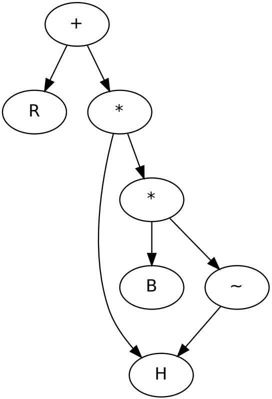
```
auto A = R + H*B*~H;
```



Expression tree for $R + HBH^T$

- All leaf nodes are interfaces to the Fortran code. We have implementations of the adjoint for these.
- Every linear operator knows it domain and codomain. During construction of a new linear operator (a new node), e.g. `auto BHt=B*~H;` we do a static assert (compile time) that the domain and codomain are consistent.
- Vectors are treated as linear operators. The domain is the scalar field (typically `double`). The codomain is the vector space itself.
- Construction of a new operator also constructs the adjoint.
- We have a separation between the construction of linear operators and the application of the operators to vectors.
- In the current mfla the generalized `HMatrix` from OOPS is used.

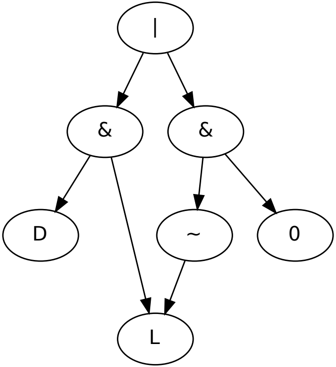
```
auto A = R + H*B*~H;
```



Expression trees for $\begin{bmatrix} D & L \\ L^T & 0 \end{bmatrix}$

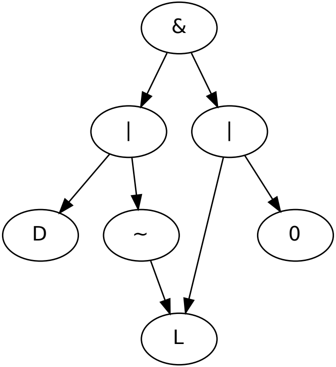
```cpp
auto S = D & L | ~L & 0;
```

$$\begin{bmatrix} \begin{bmatrix} D & L \end{bmatrix} \\ \begin{bmatrix} L^T & 0 \end{bmatrix} \end{bmatrix}$$




```cpp
auto S = (D | ~L) & (L | 0);
```

$$\begin{bmatrix} \begin{bmatrix} D \\ L^T \end{bmatrix} & \begin{bmatrix} L \\ 0 \end{bmatrix} \end{bmatrix}$$



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 $x_1, \ldots x_n$ we can construct an ensemble as

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

We can then construct new operators

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

and

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

In the first case this constructs an operator that can act on vectors x_i. In the second case we create an operator from $\mathbb{R}^n \rightarrow \mathbb{R}^n$.

Further development for mfla

- Define an interface for the NL, TL and AD for each operator to simplify unit-tests.
- Automatically generate the TL and AD code?
- Extend to nonlinear operators and States. E.g. let `ModelState` be vertical concatenation of the T, div, vor, p, q fields. Can the class can be eliminated from OOPS?
- Replace the observer design pattern (`PostProcessors`) in `HMatrix` etc. by composition of operators

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>`

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

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
- Open issues
  - Signature of the NL TL AD functions and their underlying relation
  - How to handle linearization state of operators
  - Automatically generate TL/AD code at compile time? (Adept and FADBAD++)
  - C++11 in OOPS (use of `auto`, move constructors, rvalue references). The Cray compiler support for C++11 is limited.
  - 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?

Additional slides

Linear algebra in the current OOPS system

Vectors

```
Departures          = ObsVector; // ObsVector = MODEL::ObsVector;
Increment4D         = Increment   | Increment |  ...  | Increment;
ControlIncrement    = Increment4D | ModelAuxIncrement | ObsAuxIncrement;
DualVector          = Departures  | Increment | ControlIncrement;
SaddlepointVector = ControlIncrement | DualVector;
```

Linear algebra in the current OOPS system

Vectors

```
Departures          = ObsVector; // ObsVector = MODEL::ObsVector;
Increment4D         = Increment   | Increment |  ...  | Increment;
ControlIncrement    = Increment4D | ModelAuxIncrement | ObsAuxIncrement;
DualVector          = Departures  | Increment | ControlIncrement;
SaddlepointVector = ControlIncrement | DualVector;
```

Matrices

```
HMatrix::multiply(const ControlIncrement &,DualVector);
BMatrix::multiply(const ControlIncrement &,ControlIncrement);
HtRinvHMatrix::multiply(const ControlIncrement &,ControlIncrement);
RinvMatrix::multiply(const DualVector &,DualVector);
HBHtMatrix::multiply(const DualVector &,DualVector);
HessianMatrix::multiply(const ControlIncrement &,ControlIncrement);
SaddlePointMatrix::multiply(const SaddlePointVector &,SaddlePointVector);
SaddlePointPrecondMatrix::multiplyconst SaddlePointVector &,SaddlePointVector)
EnsembleCovariance::multiply(const Increment &,Increment);
HybridCovariance::multiply(const Increment &,Increment);
ObsErrorDiag::multiply(const MODEL::ObsVector &,MODEL::ObsVector);
ModelAuxCovariance::multiply(const ModelAuxIncrement &,ModelAuxIncrement);
ObsAuxCovariance::multiply(const ObsAuxIncrement &,ObsAuxIncrement);
ObsErrorCovariance::multiply(const Departures &,Departures);
```

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.

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.
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.

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);}
```

Adjoint code

Let $\mathcal{A}_i$ be a nonlinear operator for which we recompute in the TL and AD code.
Given

$$x_3 = (\mathcal{A}_3 \circ \mathcal{A}_2 \circ \mathcal{A}_1)(x_0)$$

During the nonlinear integration each object $\mathcal{A}_i$ should store (and own) it's own linearization state. (No separate class for the linearization trajectory).
This would imply that the objects $\mathcal{A}_i$ are not fully initialized after the call to the constructor. Better let `operator()` of each object $\mathcal{A}_i$ return the TL/AD operator instead of x_i.[1]
Given

$$(A_3 \circ A_2 \circ A_1) = (\mathcal{A}_3 \circ \mathcal{A}_2 \circ \mathcal{A}_1)(x_0)$$

$$\delta x_3 = (A_3 \circ A_2 \circ A_1) * \delta x_0$$

AD

$$\delta x_0^* = (A_1^T \circ A_2^T \circ A_3^T) * \delta x_3^*$$

Each TL/AD object should be convertible to an element in the codomain such that we can still do composition of the nonlinear objects. i.e. in $(A_3 \circ A_2 \circ A_1) = (\mathcal{A}_3 \circ \mathcal{A}_2 \circ \mathcal{A}_1)(x_0)$ we don't compute $x_3 = \mathcal{A}_3(x_2)$ as it is not needed for the TL and AD only when the is an explicit request for x_3 e.g. because of ~~composition with~~ $\cancel{\mathcal{A}_4}$ ~~do we calculate~~ the value.

[1] Both the highres NL and the Lowres NL should return the TL/AD of the Lowres. What about code that reads the whole Highres traj.?

Signature of the TL and AD operators

TL	`void Htl(const X&, Y&)`
	`Htl(x,y)`
	$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I & 0 \\ H & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
		`Y Htl(const X&);`
		`auto y = Htl(x);`
		$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I \\ H \end{bmatrix} \begin{bmatrix} x \end{bmatrix}$
			`Y Htl(X&&);`
			`auto y = Htl(std::move(x));`
			`// auto y = Htl(f())`
			$\begin{bmatrix} y \end{bmatrix} = \begin{bmatrix} H \end{bmatrix} \begin{bmatrix} x \end{bmatrix}$

AD	$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} I & Ht \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
	`void Had(X& x, Y& y)`
		$\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} I & Ht \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
		`void Had(X& x, Y&& y);`
		`//X& x = Had(Y&& y);`
			$\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} Ht \end{bmatrix} \begin{bmatrix} y \end{bmatrix}$
			`X Had(Y&& y);`

- Guidelines Stroustrup
  - Return a result as a return value rather than modifying an object through an argument.
  - Use pass-by-reference only if you have to.
- 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
- Option 3) `x` is not allowed to be a deduced type[3]. Introduce unit-tests for this.
- Option 3) For consistency also Hnl should take argument by rvalue-reference.

[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] On the stack

[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

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

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.

Naive Krylov methods

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

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

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.

```
Mn = iterate(M,n); dX = Mn*dx0 //M is the single time step propagator
```