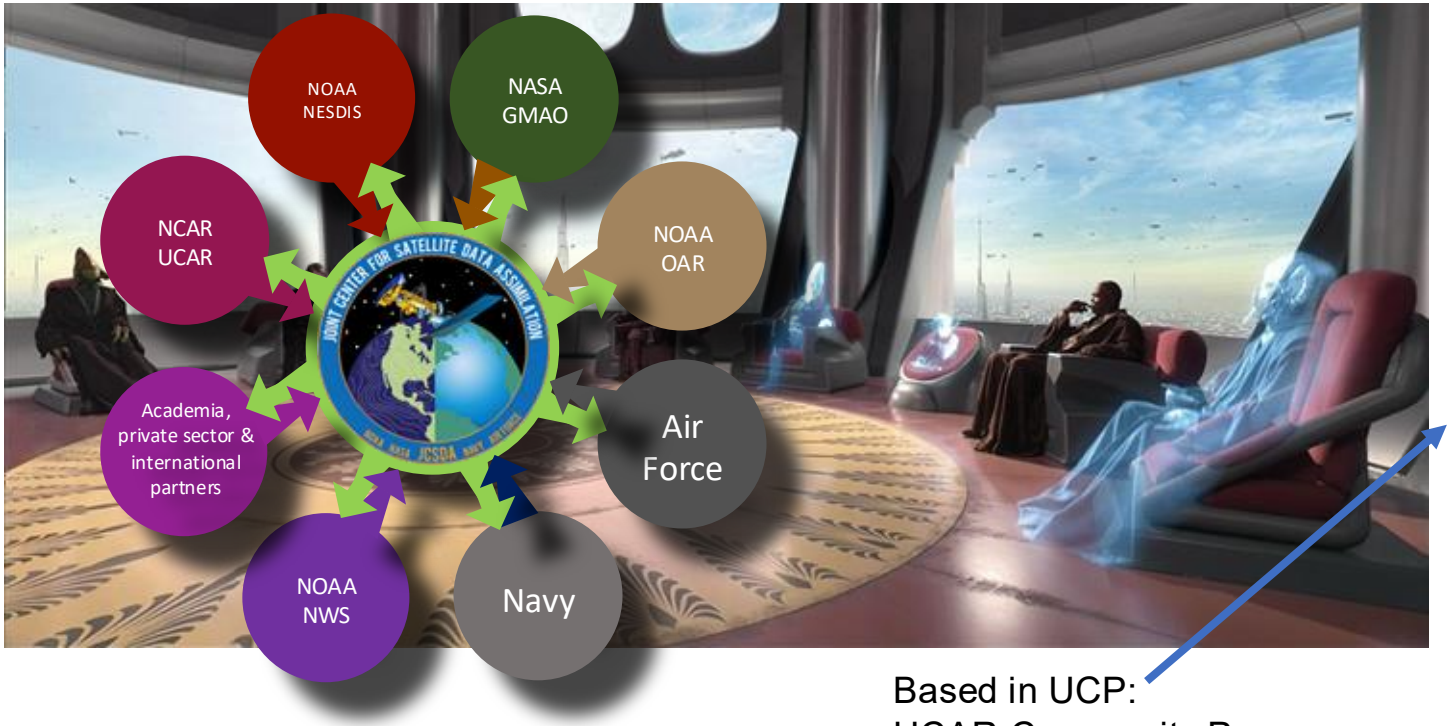
Introduction to JEDI

MPAS-JEDI Tutorial
Aug 13, 2026 – Foothills Lab, Boulder CO

Nate Crossette

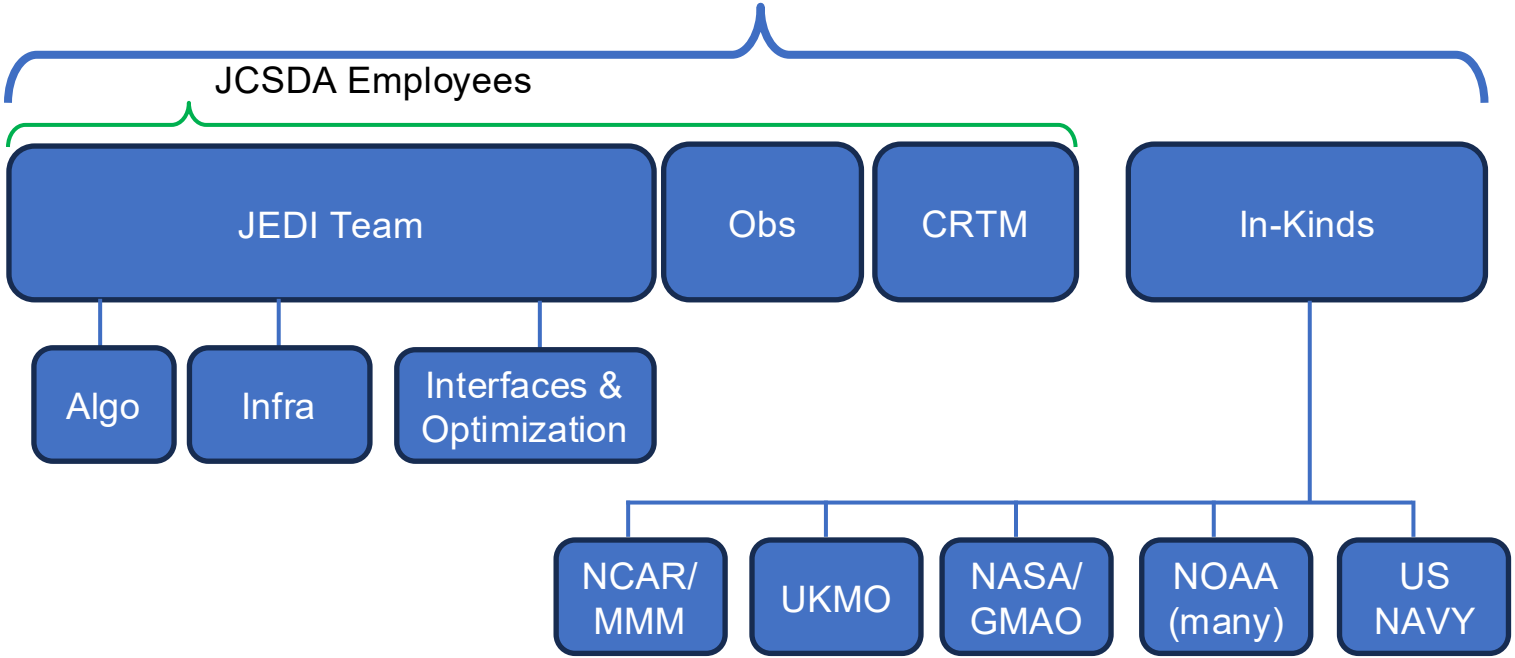
The JCSDA



Based in UCP:
UCAR Community Programs

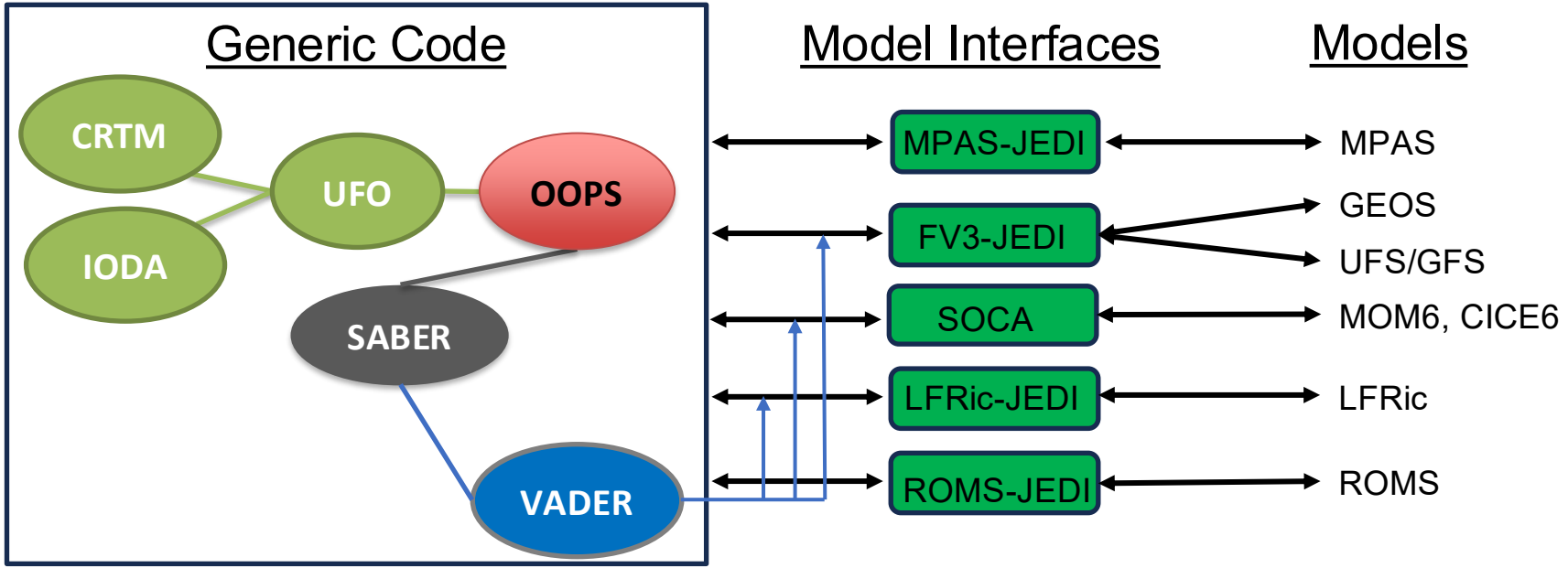
The People of the JCSDA

Joint Center for Satellite Data Assimilation



Joint Effort for Data assimilation Integration

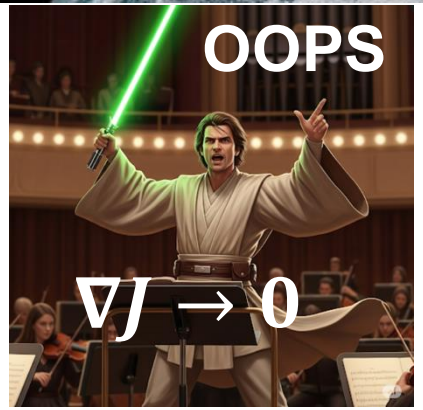
- Generic (model-agnostic), unified data assimilation framework for **Research AND Operations**
  - JEDI is run with toy models (Lorenz95/QG) up to Earth system coupled models



JEDI Components

- OOPS: Object Oriented Prediction System
  - Generic & Abstract data assimilation algorithms
- VADER: VAriable DErivation Repository
  - Generic variable changes
- SABER: System Agnostic Background Error Representation
  - Generic background error covariance models (B-matrix)
- UFO: Unified Forward Operator
  - Collection of model-agnostic observation operators, filters, QC capabilities, etc.
- CRTM: Community Radiative Transfer Model
  - Simulation of satellite radiances
- IODA: Interface for Observation Data Access
  - Performs I/O and provides generic containers for observations

JEDI Components



conductor/
director/
orchestrator

B-Matrix Models

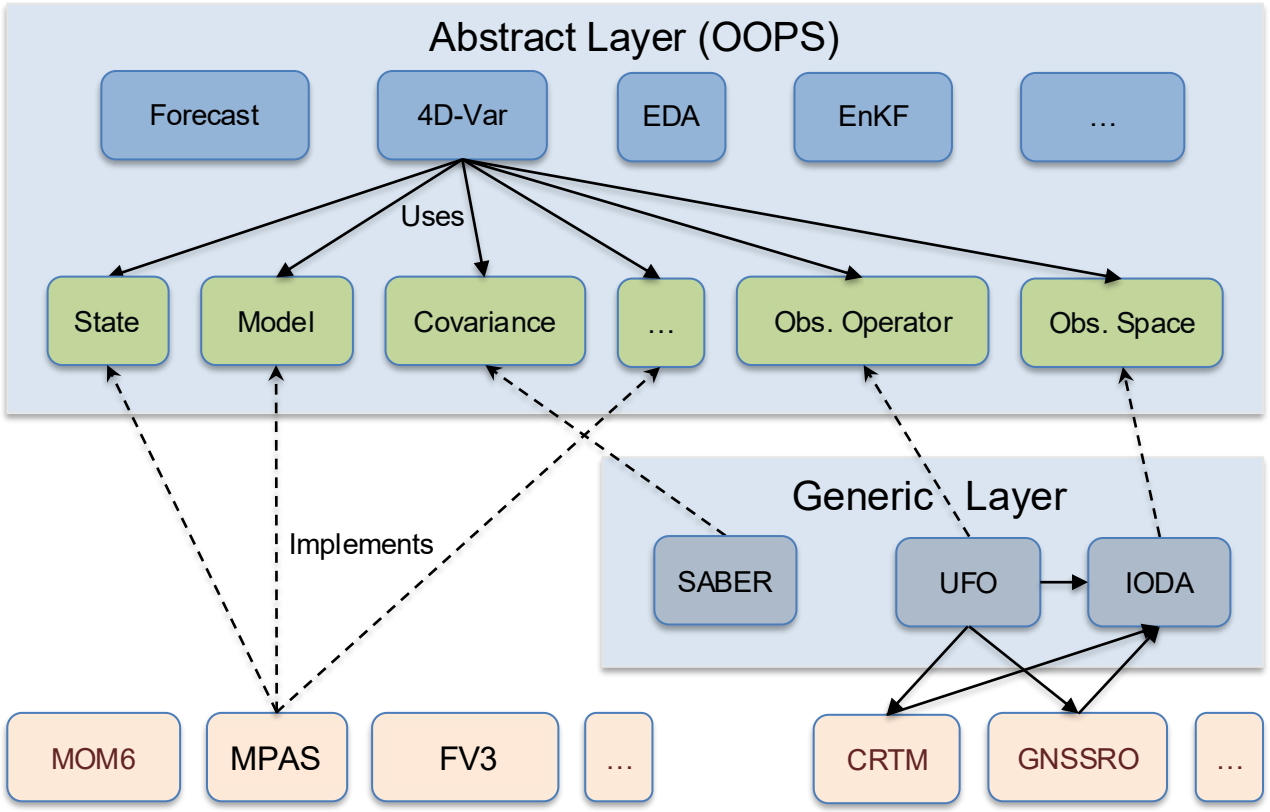
Generic Obs: Errors; Operators (including CRTM); Filters; etc.

$$J(x) = \frac{1}{2}(x - x_b)^T B^{-1}(x - x_b) + \frac{1}{2}\left(y - H(x)\right)^T R^{-1}\left(y - H(x)\right)$$

Variable transforms (as generic 'recipes')

Standardized Obs data format and obs data containers

JEDI - Abstraction and Genericity



Model-agnostic Abstract Interface *(outlines what a model interface needs to implement)*

Model-agnostic components implementing part of the OOPS interface.

How is JEDI code 'Generic'?

DA in one sentence:

Update a model **state** using new observations accounting for errors: obs, background, (model).

$$J = J_b(x - x_b) + J_o\big(y - H(x)\big)$$

In J_b, JEDI uses `atlas::FieldSet` as generic data structure for state x (including model grid geometry)

In J_o, JEDI uses `ioda::ObsVector` as generic data structure (including Obs Locations)

atlas — From ECMWF

A library for Numerical Weather Prediction and Climate Modelling

Atlas is an open source library providing grids, mesh generation, and parallel data structures targetting Numerical Weather Prediction or Climate Model developments.

The `FieldSet` and `ObsVector` are 'workhorse' objects in JEDI algorithms

ndarray is to numpy; as atlas::FieldSet/ioda::ObsVector is to JEDI

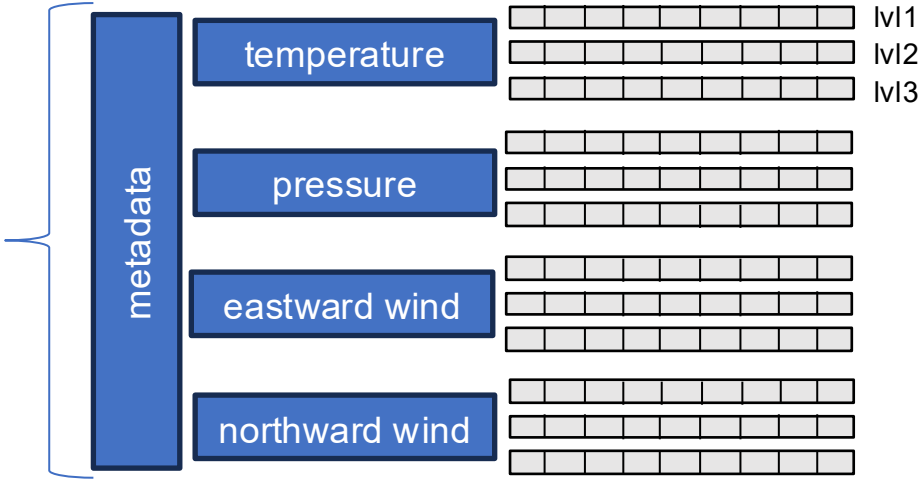
https://github.com/ecmwf/atlas;
https://sites.ecmwf.int/docs/atlas/

How is JEDI Code 'Generic'?

```
class FieldSet3D : public util::Serializable,
                   public util::Printable {
public:
  /// Constructors
  FieldSet3D(const util::DateTime &, const eckit::mpi::Comm &); //empty fieldSet
  FieldSet3D(const FieldSet3D &); //deep copy of other fieldSet

  /// Accessors
  const Variables & variables() const;
  const util::DateTime validTime() const {return validTime_;}
  const atlas::FieldSet & fieldSet() const {return fset_;}
  atlas::FieldSet & fieldSet() {return fset_;}

  /// Get individual fields from inside fieldSet
  atlas::Field & operator[](const int & fieldIndex) {return fset_[fieldIndex];}
  atlas::Field & operator[](const std::string & fieldName) {return fset_[fieldName];}
  atlas::Field & operator[](const Variable & var) {return fset_[var.name()];}

  /// Arithmetic operations
  void zero();
  FieldSet3D & operator+=(const FieldSet3D &);
  FieldSet3D & operator*=(const FieldSet3D &);
  FieldSet3D & operator*=(const double &);
  double dot_product_with(const FieldSet3D &, const Variables &) const;
  double norm(const Variables &) const;
  void sqrt();

  /// Utilities
  // iterators, add/remove fields, etc...

private:
  oops::Variables currentVariables() const;
  void print(std::ostream &) const override;

  atlas::FieldSet fset_;
  const util::DateTime validTime_;
  const eckit::mpi::Comm & comm_;
  std::string name_;
  mutable oops::Variables vars_;
};
```

JEDI wrapper for `atlas::FieldSet`

- Access variables/fieldnames
- Access individual field
- Perform operations



How is JEDI code 'Generic'?

```
template <typename MODEL>
class State : public util::Printable,
              public util::Serializable,
              private util::ObjectCounter<State<MODEL> > {
  typedef typename MODEL::State            State_;
  typedef oops::Geometry<MODEL>            Geometry_;

 public:
  /// Constructors/Destructor
  State(const Geometry_ & resol, const Variables & vars, const util::DateTime & time);
  ...
  /// Assignment operator
  State & operator =(const State &);
  /// Accessors
  State_ & state() {if (fset_) {fset_->clear();} return *state_;}
  const State_ & state() const {return *state_;}
  ...
  /// Accessor to variables associated with this State
  const Variables & variables() const;
  /// Class Methods
      ...

  ///  IMPORTANT METHODS ::
  void toFieldSet(atlas::FieldSet &) const;
  void fromFieldSet(const atlas::FieldSet &);

 private:
  std::unique_ptr<State_> state_;
  size_t ID_;
  void print(std::ostream &) const override;

 protected:
  mutable std::unique_ptr<FieldSet3D> fset_;
};
```

Model-facing classes in OOPS are class templates

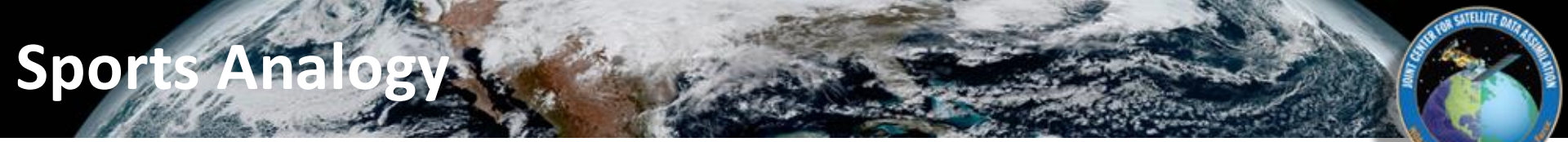
Methods are implemented on generic concepts of `MODEL::state` and `MODEL::geometry`

This 'interface' layer contains (wraps) a pointer to the model-specific "state"

Model-specific definitions are written in model interfaces and ***fill*** the class template

→ ex, the model-specific `to/fromFieldSet` methods are implemented in model interfaces

Sports Analogy



A tournament is a sport-agnostic event. A tournament can be organized by a tournament committee with only an **abstract** idea of a "*game*".

Plug in a specific ***sport****/model* into a tournament (accomplished with a ***sport****/model* interface), and the organizers (OOPS) will be able to run the event.

The tournament actions (e.g. PlayGame, Call Penalty) equivalents in JEDI (e.g. HofX, forecast, Variational) are `OOPS::Application`**s**

Class Templates in C++

A class template IS NOT A CLASS. It is a RECIPE for making a CLASS

```
template <typename SPORT>
class Tournament{
typedef SPORT::Team   Team_;
typedef SPORT::Match  MATCH;
 public:
  Tournament(config);
  void runTournament(){
    while(tournament){
      // run tournament ...
      MATCH::playMatch(team1, team2)
    }
  }
private:
  std::set<Team_> competitors;
};
```

```
// ======================
//   Baseball-Interface
// ======================

struct BaseballTraits {
  // Baseball type traits
  typedef Baseball::BaseballMatch Match;
  typedef Baseball::Team          Team;
};

class BaseballMatch {
  // implementation of Baseball match
  static void playMatch();
};

class Team {
  // Baseball specfic implmentation
};
```

The equivalent MPAS-JEDI `Traits.h` 'connects the wires' and ***specific*** what get filled into the generic OOPS `MODEL::<THING>`

Template Instantiation (in Model Interfaces)

The (compile time) process of ‘filling’ a template to create a true class

MPAS-JEDI Variational Application

```
/*
 * (C) Copyright 2017 UCAR
 *
 * This software is licensed under the terms of the Apache Licence Version 2.0
 * which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
 */

#include "oops/generic/instantiateModelFactory.h"
#include "oops/runs/Run.h"
#include "oops/runs/Variational.h"

#include "saber/oops/instantiateCovarFactory.h"
#include "saber/oops/instantiateLocalizationFactory.h"

#include "ufo/instantiateObsErrorFactory.h"
#include "ufo/instantiateObsFilterFactory.h"
#include "ufo/ObsTraits.h"

#include "mpasjedi/Traits.h"

int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  oops::instantiateModelFactory<mpas::Traits>();
  saber::instantiateCovarFactory<mpas::Traits>();
  saber::instantiateLocalizationFactory<mpas::Traits>();
  ufo::instantiateObsErrorFactory();
  ufo::instantiateObsFilterFactory();
  oops::Variational<mpas::Traits, ufo::ObsTraits> var;
  return run.execute(var);
}
```

FV3-JEDI Variational Application

```
/*
 * (C) Copyright 2017-2020 UCAR
 *
 * This software is licensed under the terms of the Apache Licence Version 2.0
 * which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
 */

#include "fv3jedi/Utilities/Traits.h"
#include "oops/generic/instantiateModelFactory.h"
#include "oops/runs/Run.h"
#include "saber/oops/instantiateCovarFactory.h"
#include "ufo/instantiateObsErrorFactory.h"
#include "ufo/instantiateObsFilterFactory.h"
#include "ufo/ObsTraits.h"

#include "oops/runs/Variational.h"

int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  saber::instantiateCovarFactory<fv3jedi::Traits>();
  ufo::instantiateObsErrorFactory();
  ufo::instantiateObsFilterFactory();
  oops::instantiateModelFactory<fv3jedi::Traits>();
  oops::Variational<fv3jedi::Traits, ufo::ObsTraits> var;
  return run.execute(var);
}
```

Template Instantiation (in Model Interfaces)

The (compile time) process of ‘filling’ a template to create a true class

MPAS-JEDI Variational Application

```
/*
 * (C) Copyright 2017 UCAR
 *
 * This software is licensed under the terms of the Apache Licence Version 2.0
 * which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
 */

#include "oops/generic/instantiateModelFactory.h"
#include "oops/runs/Run.h"
#include "oops/runs/Variational.h"

#include "saber/oops/instantiateCovarFactory.h"
#include "saber/oops/instantiateLocalizationFactory.h"

#include "ufo/instantiateObsErrorFactory.h"
#include "ufo/instantiateObsFilterFactory.h"
#include "ufo/ObsTraits.h"

#include "mpasjedi/Traits.h"

int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  oops::instantiateModelFactory<mpas::Traits>();
  saber::instantiateCovarFactory<mpas::Traits>();
  sa
  uf
  uf
  oo
  re
}
```

FV3-JEDI Variational Application

```
/*
 * (C) Copyright 2017-2020 UCAR
 *
 * This software is licensed under the terms of the Apache Licence Version 2.0
 * which can be obtained at http://www.apache.org/licenses/LICENSE-2.0.
 */

#include "fv3jedi/Utilities/Traits.h"
#include "oops/generic/instantiateModelFactory.h"
#include "oops/runs/Run.h"
#include "saber/oops/instantiateCovarFactory.h"
#include "ufo/instantiateObsErrorFactory.h"
#include "ufo/instantiateObsFilterFactory.h"
#include "ufo/ObsTraits.h"

#include "oops/runs/Variational.h"

int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  saber::instantiateCovarFactory<fv3jedi::Traits>();
  ufo::instantiateObsErrorFactory();
```

Only define ‘how to do DA’ once, so for each model have the model-interface to connect the wires.

var;

OOPS Orchestration

DA ‘tasks’ are defined as subclasses of `OOPS::Application`

```
class Application : public util::Printable {
 public:
  explicit Application(const eckit::mpi::Comm & comm) : comm_(comm) {}
  virtual ~Application() {}
  virtual int execute(const eckit::Configuration &, bool validate) const = 0;
```

What the application does is defined in the `execute()` method.

```
template <typename MODEL, typename OBS> class Variational : public Application {
  typedef Increment<MODEL>           Increment_;
  typedef State<MODEL>               State_;
  typedef Model<MODEL>                 Model_;
  typedef ModelAuxControl<MODEL>       ModelAux_;

 public:
// -----------------------------------------------------------------------------
  explicit Variational(const eckit::mpi::Comm & comm = oops::mpi::world()) : Application(comm) {
    instantiateCostFactory<MODEL, OBS>();
    instantiateCovarFactory<MODEL>();
    ...
  }
  int execute(const eckit::Configuration & fullConfig, bool validate) const override {
    Log::trace() << "Variational: execute start" << std::endl;
```

```
oops/src/
 |- test/
 |- oops/
      |- assimilation/
      |- base/
      |- generic/
      |- interface/
--> |- runs/
           Application.h
           Variational.h
           ...
      ...
```

YAML Ain't Markup Language

Experiments are setup in yaml files

YAML files are read by eckit and written into `eckit::Configuration` object that is passed to OOPS.

```yaml
cost function:
  cost type: 3D-Var
  time window:
    begin: '2018-04-14T21:00:00Z'
    length: PT6H
  geometry:
    nml_file: "./Data/480km/namelist.atmosphere_2018041500"
    streams_file: "./Data/480km/streams.atmosphere"
  analysis variables: [list of an_vars]
  background:
    state variables: [List of vars to read from state file]
    filename: "./Data/480km/bg/restart.2018-04-15_00.00.00.nc"
    date: &analysisdate '2018-04-15T00:00:00Z'
  background error:
    covariance model: SABER
    saber central block:
      saber block name: ID
      active variables: [sub-set of analysis vars]
    saber outer blocks:
    - saber block name: StdDev
      ...
  observations:
    observers:
    - obs space:
        name: MY_OBS
        ...
            obsfile: <path to MY_OBS file>
        ...
variational:
  minimizer:
    algorithm: DRPCG
  iterations:
  - geometry:
      nml_file: "./Data/480km/namelist.atmosphere_2018041500"
      streams_file: "./Data/480km/streams.atmosphere"
    ninner: '10'
```

```
template <typename MODEL, typename OBS> class Variational : public Application {
  typedef Increment<MODEL>          Increment_;
  typedef State<MODEL>              State_;
  typedef Model<MODEL>              Model_;
  typedef ModelAuxControl<MODEL>    ModelAux_;

 public:
// -----------------------------------------------------------------------------
  explicit Variational(const eckit::mpi::Comm & comm = oops::mpi::world()) : Application(comm) {
    instantiateCostFactory<MODEL, OBS>();
    instantiateCovarFactory<MODEL>();
    ...
  }
  int execute(const eckit::Configuration & fullConfig, bool validate) const override {
    Log::trace() << "Variational: execute start" << std::endl;
```

Telling OOPS what to do

Inside OOPS, the `eckit::Configuration` will look like a nested python dictionary

```
{ 'cost function':
  { 'cost type': '3D-Var',
   'time window':
    { 'begin': '2018-04-14T21:00:00Z', 'length': 'PT6H' },
   'geometry':
    { 'nml_file': './Data/480km/namelist.atmosphere_2018041500',
      'streams_file': './Data/480km/streams.atmosphere' },
   'analysis variables': ['list of an_vars'],
   'background':
    { 'state variables': ['List of vars to read from state file'],
      'filename': './Data/480km/bg/restart.2018-04-15_00.00.00.nc',
      'date': '2018-04-15T00:00:00Z' },
   'background error':
    { 'covariance model': 'SABER',
      'saber central block':
       { 'saber block name': 'ID',
         'active variables': ['sub-set of analysis vars'] },
       'saber outer blocks': [ // start list of outer blocks
         {'saber block name':
          'StdDev': -> {**StdDev BLOCK PARAMETERS**} },
          ...] //end list of outer blocks },
   'observations':
    { 'observers': [ //start list of observers
      { 'obs space':
        { 'name': 'MY_OBS',
          ...
          'obsfile': '<path to MY_OBS file>' } }
               ] //end list of observers }
  },
  'variational':
   { 'minimizer':
    { 'algorithm': 'DRPCG' },
     'iterations': [ //begin list of outer loop settings
      { 'geometry':
        { 'nml_file': './Data/480km/namelist.atmosphere_2018041500',
          'streams_file': './Data/480km/streams.atmosphere' },
        'ninner': '10',
        'gradient norm reduction': '1e-10' }
       ] //end list of outer loop settings }
}// end config
```

Telling OOPS what to do

Inside OOPS, the `eckit::Configuration` will look like a nested python dictionary

OOPS will read parts out of this and put them into an: `eckit::LocalConfiguration`

```
{ 'cost function':
  'cost type': '3D-Var',
  'time window':
   { 'begin': '2018-04-14T21:00:00Z', 'length': 'PT6H' },
  'geometry':
   { 'nml_file': './Data/480km/namelist.atmosphere_2018041500',
     'streams_file': './Data/480km/streams.atmosphere' },
  'analysis variables': ['list of an_vars'],
  'background':
   { 'state variables': ['List of vars to read from state file'],
     'filename': './Data/480km/bg/restart.2018-04-15_00.00.00.nc',
     'date': '2018-04-15T00:00:00Z' },
  'background error':
  { 'covariance model': 'SABER',
    'saber central block':
     { 'saber block name': 'ID',
       'active variables': ['sub-set of analysis vars'] },
    'saber outer blocks': [ // start list of outer blocks
      {'saber block name':
        'StdDev': -> {**StdDev BLOCK PARAMETERS**} },
       ...] //end list of outer blocks },
  'observations':
   { 'observers': [ //start list of observers
     { 'obs space':
       { 'name': 'MY_OBS',
         ...
         'obsfile': '<path to MY_OBS file>' } }
           ] //end list of observers }
}
```

```
//  Setup cost function
    eckit::LocalConfiguration cfConf(fullConfig, "cost function");
    std::unique_ptr<CostFunction<MODEL, OBS>>
      J(CostFactory<MODEL, OBS>::create(cfConf, this->getComm()));
```

TANGENT – Abstract Factories

```
//  Setup cost function
    eckit::LocalConfiguration cfConf(fullConfig, "cost function");
    std::unique_ptr<CostFunction<MODEL, OBS>>
      J(CostFactory<MODEL, OBS>::create(cfConf, this->getComm()));
```

Once you build JEDI, you can run a wide variety of applications (HofX, B-matrix training, 3dVar, EnKF, …) with a wide variety of input yaml configurations.

Abstract factories are an important feature enabling this flexibility

Runtime Polymorphism with Factories

```
//  Setup cost function
    eckit::LocalConfiguration cfConf(fullConfig, "cost function");
    std::unique_ptr<CostFunction<MODEL, OBS>>
      J(CostFactory<MODEL, OBS>::create(cfConf, this->getComm()));
```

Abstract factory design pattern:

- *Allows automatic generation of generic class of objects (derived from a common base class)* ***<u>at runtime</u>***

Only need to compile one executable, which supports flexible execution based on input configuration

- e.g., can swap out cost-function, B-Matrix, OBS, etc within SAME application
- i.e., there is ONE variational application (not one per cost-function)

What does OOPS do with the config?

```
//  Setup cost function
    eckit::LocalConfiguration cfConf(fullConfig, "cost function");
    std::unique_ptr<CostFunction<MODEL, OBS>>
      J(CostFactory<MODEL, OBS>::create(cfConf, this->getComm()));
```

The cost function factory will build the Jb and Jo terms (including the B and R matrix models) specified in the yaml config…

Most likely by passing the `background error` sub-config to SABER and the `observations` sub-config UFO

```
{ 'cost function':
  { 'cost type': '3D-Var', ...},
   'background error':
    { 'covariance model': 'SABER', ...}
   'observations':
    { 'observers':
      [ list of observers ] }
 }, // end "cost function" section
 ...
}// end config
```

What happens to the Config in SABER (and UFO)?

We've now crossed from the "Abstract" Layer to the "Generic" Layer of JEDI

In the Generic layer, the `eckit::Configuration` gets turned into `Parameters` which:

- wrap and organize the config
- 'know' more about what specific components want/need in the config and can do some error checking
- have recently been completely removed from OOPS (still in SABER/UFO)

```
background error:
  covariance model: SABER
  covariance type: hybrid
  run components recursively: true
  components:
    # Component 1 (ensemble)
  - weight:
        value: 1.0
    covariance:
        covariance type: ensemble
        ensemble:
          < ensemble config >
        localization:
            saber central block:
              saber block name: spectral analytical filter
                < central block parameters >
            saber outer blocks:
            [ list of outer blocks for localization ]
        saber outer blocks:
          < optional outer-outer block >
    # Component 2 (static/parametric)
  - weight:
        value: 1.0
    covariance:
      covariance model: SABER
      saber central block:
        saber block name: BUMP_NICAS
          < block parameters >
      saber outer blocks:
      - saber block name: StdDev
          < block parameters >
```

SABER Parameters

This says ‘build a SABER ErrorCovariance”
(*as opposed to an oops or model-specific one*)

```
class ErrorCovarianceParameters : public ErrorCovarianceParametersBase {
  OOPS_CONCRETE_PARAMETERS(ErrorCovarianceParameters,
                           ErrorCovarianceParametersBase)

 public:
  /// ModelSpaceCovarianceBase class parameters

  // Covariance model
  oops::OptionalParameter<std::string> covarianceModel{"covariance model", this};
```

```
  virtual ~ErrorCovarianceToolbox() {}
// -----------------------------------------------------------------
  int execute(const eckit::Configuration & fullConfig)
    // Deserialize parameters
    ErrorCovarianceToolboxParameters params;
    params.deserialize(fullConfig);
```

```
// ACCESSING SPECIFIC PARAMETER VALUE:
const auto & covType = params.covarianceModel.value();
```

```
background error:
  covariance model: SABER
  covariance type: hybrid
  run components recursively: true
  components:
    # Component 1 (ensemble)
  - weight:
      value: 1.0
    covariance:
      covariance type: ensemble
      ensemble:
       < ensemble config >
      localization:
        saber central block:
          saber block name: spectral analytical filter
           < central block parameters >
        saber outer blocks:
        [ list of outer blocks for localization ]
      saber outer blocks:
       < optional outer-outer block >
    # Component 2 (static/parametric)
  - weight:
      value: 1.0
    covariance:
      covariance model: SABER
      saber central block:
        saber block name: BUMP_NICAS
         < block parameters >
      saber outer blocks:
      - saber block name: StdDev
         < block parameters >
```

SABER Generic STDDEV Block Parameters

Error Checking: will throw (runtime error) if value zero or less

```yaml
saber outer blocks:
- saber block name: StdDev
  stddev scale factor: 2.0
  read:
    atlas file:
      filepath: testdata/stddev_data
```

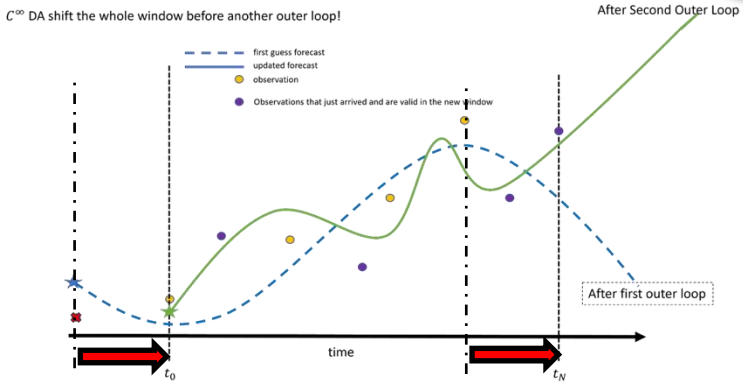
```
class StdDevParameters : public SaberBlockParametersBase {
  OOPS_CONCRETE_PARAMETERS(StdDevParameters, SaberBlockParametersBase)

 public:
  // Read parameters
  oops::OptionalParameter<StdDevReadParameters> readParams{"read", this};

  // Calibration of block parameters
  oops::OptionalParameter<StdDevWriteParameters> calibrationParams{"calibration", this};

  // Scaling parameter
  oops::Parameter<double> scaleFactorParam{"stddev scale factor",
                                           "multiplicative factor applied to StdDev block",
                                           1.0, this, {oops::exclusiveMinConstraint(0.)}};

  oops::Variables mandatoryActiveVars() const override {return oops::Variables();}

  oops::Parameter<eckit::LocalConfiguration> scaling{"standard deviations",
                                                     eckit::LocalConfiguration(), this};
};
```

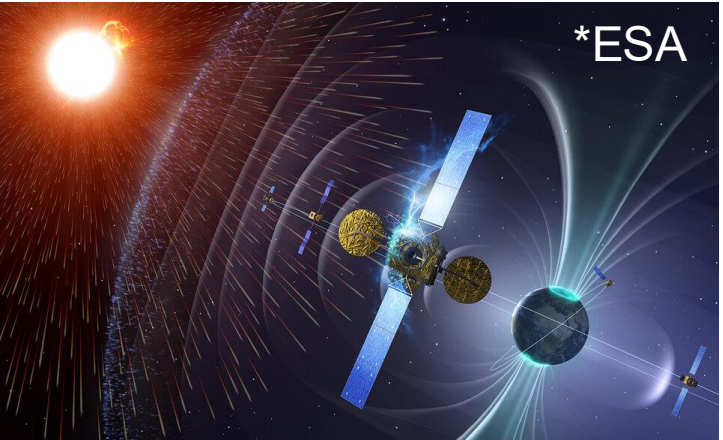
Upshot:

Looking at the `Parameters` classes for the JEDI component you are working with maybe helpful!

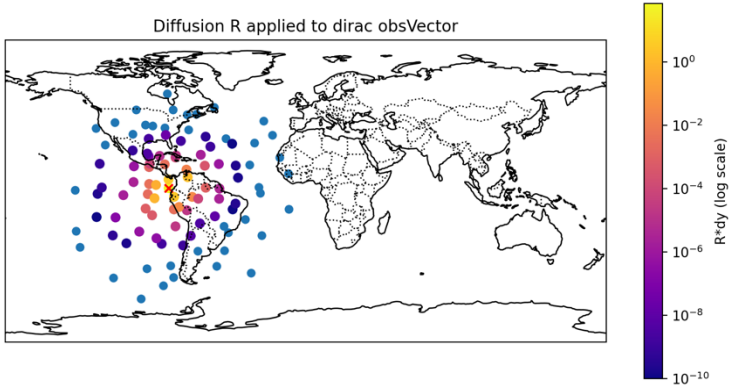
Current JEDI Efforts

- Transition to Operations
- New OBS data container: OSDF (IODA)
- Continuous DA (OOPS)
- Diffusion-based Correlated R (UFO)
- Scale-Dependent B-Matix (SABER)
- Space Weather (Model Interfaces)





COMING SOON:
`mist` – Helper Library for model interfaces



JEDI Capabilities & Features

More on this today & tomorrow

Models:

- Lorenz95, QG (native toy models)
- FV3-JEDI
- MPAS-JEDI
- SOCA
- PyIRI-JEDI

DA Flavors:

- 3DVar/3D-FGAT
- 4DVar (-Weak Constraint)
- 4DEnVar
- EnKF (LETKF, GETKF)
- EDA

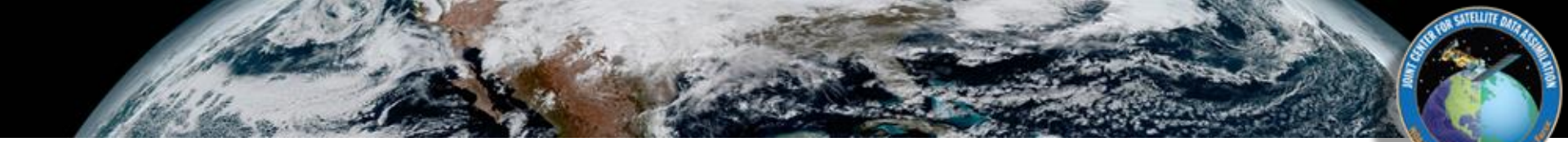
Background Error Models (SABER):

- BUMP
- Explicit Diffusion
- Spectral Filtering
- GSI Interface (limited)

UFO:

- Wide collection of Obs Operators (remote & in-situ)
- Bias correction
- Quality Control

More on this later from Francois, Ivette & BJ



Questions?

JEDI Documentation: https://jointcenterforsatellitedataassimilation-jedi-docs.readthedocs-hosted.com/en/latest/index.html
JEDI Forum: https://forums.jcsda.org/ (requires account to post/comment)
Github: https://github.com/JCSDA (public)