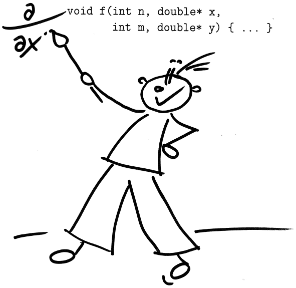


Adjoint Code Design Patterns

... applied to Monte Carlo Simulation

Uwe Naumann and Johannes Lotz

RWTH Aachen University, Aachen, Germany

Outline

Motivation

Adjoint Code Design Patterns

- Sample Scenario
- Concept
- Implementation with `dco/c++`

Further Details

- Late Recording
- Ensembles
- Evolutions
- Nonlinear Systems

Conclusion

Outline

Motivation

Adjoint Code Design Patterns

- Sample Scenario
- Concept
- Implementation with `dco/c++`

Further Details

- Late Recording
- Ensembles
- Evolutions
- Nonlinear Systems

Conclusion

$$\boldsymbol{R}^m \ni \mathrm{r} \equiv R(\mathrm{y}(\mathrm{x}), \mathrm{x}) = 0$$

symbolic adjoint

$$\mathrm{r}_{(1)} \cdot \frac{dr}{dx} = 0$$

$$\underbrace{\underbrace{\mathrm{r}_{(1)} \cdot \frac{dR}{dy}}_{=\mathrm{y}_{(1)} \in \boldsymbol{R}^{1\times m}} \cdot \frac{dy}{dx} + \mathrm{r}_{(1)} \cdot \frac{\partial R}{\partial x}}_{=\mathrm{x}_{(1)} \in \boldsymbol{R}^{1\times n}} = 0$$

[augmented] primal

$$\mathrm{r} \rightarrow 0$$

[approximate] algorithmic tangent

$$\mathrm{y}^{(1)} = F' \cdot \mathrm{x}^{(1)} \left[\approx \frac{F(\mathrm{x} + \mathrm{x}^{(1)} \cdot h) - F(\mathrm{x})}{h} \right]$$

$$\mathrm{r}_{(1)} \cdot \frac{dR}{dy} = \mathrm{y}_{(1)}$$

$$\mathrm{x}_{(1)} = -\mathrm{r}_{(1)} \cdot \frac{\partial R}{\partial x}$$

$$\mathrm{y} = F(\mathrm{x});\ \boldsymbol{R}^n \rightarrow \boldsymbol{R}^m;\ [G]$$

algorithmic adjoint

$$\mathrm{x}_{(1)} = \mathrm{y}_{(1)} \cdot F'$$

???

$$\mathrm{y}_{(1)} \cdot \mathrm{y}^{(1)} = \mathrm{x}_{(1)} \cdot \mathrm{x}^{(1)}$$

The adjoint of a program $y = v_q := F(x = v_0)$ computes

$$V_{0(1)} = \underbrace{X_{(1)}}_{\in R^{l \times n}} := \underbrace{Y_{(1)}}_{\in R^{l \times m}} \cdot F'(x) = \left(\ldots \left(V_{q(1)} \cdot F'_q \right) \ldots \cdot F'_1 \right)$$

assuming availability of adjoint elemental functions (elemental adjoints)

$$V_{i-1(1)} := V_{i(1)} \cdot F'_i(v_{i-1})$$

for $i = q, \ldots, 1$ ($\rightarrow$ reversal of data flow).

The minimum requirement for adjoint AD (AAD) is the implementation of adjoint versions of the intrinsic operations $(+, *, \ldots)$ and functions $(\sin, \exp, \ldots)$ of the given programming language.

Their naive combination yields algorithmic adjoint programs, which may turn out infeasible for various reasons. Hierarchies in granularity and mathematical semantics must be exploited in "real world" AAD.

An elemental adjoint $F_{i(1)}$ comprises both data and instructions necessary for evaluating $V_{i-1(1)} := V_{i(1)} \cdot F_i'(\mathrm{v}_{i-1})$.

An adjoint program $F_{(1)}$ is a partially ordered sequence of evaluations of elemental adjoints.

An appropriately augmented version of the given implementation of F (the forward (augmented primal) section $\overrightarrow{F}_{(1)}$ of the adjoint program) is executed to record data required for the evaluation of

$$V_{i-1(1)} := F_{i(1)}(\mathrm{v}_{i-1}, V_{i(1)}) \equiv V_{i(1)} \cdot F_i'(\mathrm{v}_{i-1}) \quad \text{for } i = q, \ldots, 1$$

by the reverse (adjoint) section $\overleftarrow{F}_{(1)}$ of the adjoint program.

The tape of the adjoint program is a (partially ordered) concatenation of the tapes of the elemental adjoints. Basic AAD records the entire tape homogeneously based on elemental algorithmic adjoints.

Let $F_{k(1)}$ not be implemented by basic AAD.

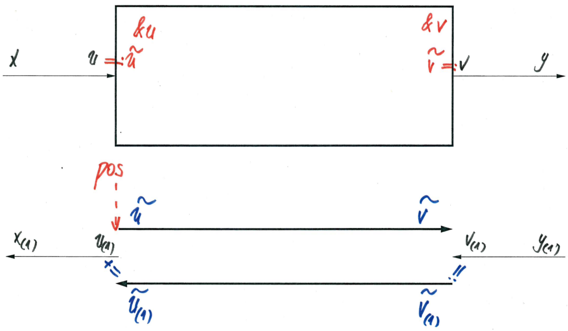
A gap is induced in the tape of the adjoint program

$$X_{(1)} = V_{0(1)} := (\ldots((\ldots(Y_{(1)} \cdot F'_q) \cdot \ldots \cdot F'_k(\mathrm{v}_{k-1})) \cdot F'_{k-1}) \cdot \ldots \cdot F'_1)$$

to be filled by a custom version of $F_{k(1)}$.

For example, checkpointing methods decrease the maximum tape size by storing v_{k-1} in the forward section followed by the evaluation of the primal F_k and postponing the generation of the tape for ${F_{(1)}}_k$ to the reverse section of $F_{(1)}$.

Further examples include the implementation of symbolic adjoint elementals, preaccumulation and approximation of Jacobians of local black boxes by finite differences.



An adjoint plugin for $v = F_k(u)$ consists of the augmented primal $v = \overrightarrow{F}_{(1)_k}(u)$ and the adjoint $U_{(1)} \mathrel{+}= \overleftarrow{F}_{(1)_k}(u, V_{(1)})$.

"In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations."

[sourcemaking.com]

- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. (*Gang of Four*)

An adjoint code design pattern is a general, reusable solution to a commonly occurring problem in adjoint code generation. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to deal with widely used reoccurring patterns in numerical simulation software in the context of AAD.

Implementations of an adjoint code design pattern yield adjoint plugins for integration into the adjoint context, e.g. and w.l.o.g., generated with dco/c++.

- U. Naumann: Adjoint code design patterns. ACM Transactions on Mathematical Software (TOMS) 45 (3), 1-32, 2019.
- U. Naumann, J. du Toit: Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance. Journal of Computational Finance 21 (4), 2018.

Outline

1. Calibration

$$\min_{x \in \mathbf{R}^{n_x}} f(x(p), p); \;\; f = \|F\|_2^2 : \mathbf{R}^{n_x} \times \mathbf{R}^{n_p} \to \mathbf{R}; \;\; F : \mathbf{R}^{n_x} \times \mathbf{R}^{n_p} \to \mathbf{R}^m$$

2. [Monte Carlo] Ensemble

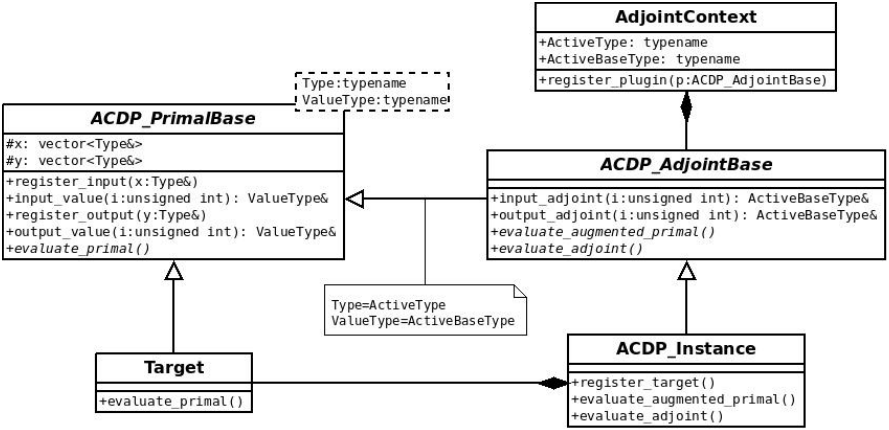
$$\frac{1}{k} \sum_{j=1}^{k} F(x, p_j); \;\; F : \mathbf{R}^{n_x} \times \mathbf{R}^{n_p} \to \mathbf{R}^m$$

3. Evolution

$$\underbrace{F(\ldots F}_{k \text{ times}}(x, p) \ldots); \;\; F : \mathbf{R}^{n_x} \times \mathbf{R}^{n_p} \to \mathbf{R}^{n_x}$$

Adjoint Code Design Patterns

Base Pattern



Implementation with dco/c++

dco/c++/etui Algorithms

- dco/c++
- dco/c++/etui
  easy to use interface

Implementation with dco/c++

dco/c++/etui Algorithms

- dco/c++
  - an AAD tool that works on C++ intrinsic functions
  - it supports a callback mechanism for writing more complex intrinsics
  - the callback mechanism is part of the low level interface
- dco/c++/etui
  easy to use interface

Implementation with dco/c++

dco/c++/etui Algorithms

- dco/c++
  - an AAD tool that works on C++ intrinsic functions
  - it supports a callback mechanism for writing more complex intrinsics
  - the callback mechanism is part of the low level interface
- dco/c++/etui
  easy to use interface
  - Drivers
  - Algorithms

Implementation with dco/c++

dco/c++/etui Algorithms

- dco/c++
  - an AAD tool that works on C++ intrinsic functions
  - it supports a callback mechanism for writing more complex intrinsics
  - the callback mechanism is part of the low level interface
- dco/c++/etui
  easy to use interface
  - Drivers
    - ease the writing of drivers
    - reduces lines of code (esp. higher-order)
    - increase efficiency
  - Algorithms

Implementation with dco/c++

dco/c++/etui Algorithms

- dco/c++
  - an AAD tool that works on C++ intrinsic functions
  - it supports a callback mechanism for writing more complex intrinsics
  - the callback mechanism is part of the low level interface
- dco/c++/etui
  easy to use interface
  - Drivers
    - ease the writing of drivers
    - reduces lines of code (esp. higher-order)
    - increase efficiency
  - Algorithms
    - implementation of ACDPs for dco/c++
    - reduces lines of code (esp. checkpointing)
    - high-level interface for exploiting reoccurring patterns (feasibility)

- dco/c++
  - an AAD tool that works on C++ intrinsic functions
  - it supports a callback mechanism for writing more complex intrinsics
  - the callback mechanism is part of the low level interface
- dco/c++/etui
  easy to use interface
  - Drivers
    - ease the writing of drivers
    - reduces lines of code (esp. higher-order)
    - increase efficiency
  - Algorithms
    - implementation of ACDPs for dco/c++
    - reduces lines of code (esp. checkpointing)
    - high-level interface for exploiting reoccurring patterns (feasibility)
- dco/c++/etui still in early development phase

- lines of code for simple example (the one Viktor showed last week)

	primal	overload (gradient)	pathwise (early prop.)	pathwise (checkpointing)	overload (Hessian)
dco/c++ plain	45	60	62	105	67
dco/c++/etui Drivers	47	50	—	—	50
dco/c++/etui Algorithms and Drivers	52	55	57	56	55

- without dco/c++/etui
  - code size increases with complexity of adjoint algorithm
  - code size increases with complexity of driver
- with dco/c++/etui
  - code size almost independent of adjoint algorithm and driver

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)

dco/c++/etui Drivers

Overview

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)
- the mode for computing derivatives can be chosen explicitly (e.g. tangent-over-adjoint for the Hessian)

dco/c++/etui Drivers

Overview

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)
- the mode for computing derivatives can be chosen explicitly (e.g. tangent-over-adjoint for the Hessian)
- drivers can be nested (compute Jacobian of a code which itself computes e.g. gradient with the dco/c++/etui drivers)

dco/c++/etui Drivers

Overview

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)
- the mode for computing derivatives can be chosen explicitly (e.g. tangent-over-adjoint for the Hessian)
- drivers can be nested (compute Jacobian of a code which itself computes e.g. gradient with the `dco/c++/etui` drivers)
- statistics can be collected (run time / memory usage)

dco/c++/etui Drivers

Overview

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)
- the mode for computing derivatives can be chosen explicitly (e.g. tangent-over-adjoint for the Hessian)
- drivers can be nested (compute Jacobian of a code which itself computes e.g. gradient with the `dco/c++/etui` drivers)
- statistics can be collected (run time / memory usage)
- generic problem definition with arbitrary number and type of parameters

- written in C++17
- currently supported drivers are
  - primal
  - tangent and adjoint
  - gradient and Jacobian (first-order)
  - Hessian (second-order)
- the mode for computing derivatives can be chosen explicitly (e.g. tangent-over-adjoint for the Hessian)
- drivers can be nested (compute Jacobian of a code which itself computes e.g. gradient with the `dco/c++/etui` drivers)
- statistics can be collected (run time / memory usage)
- generic problem definition with arbitrary number and type of parameters
- two levels of abstraction available (higher-level shown on next slide)

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
```

dco/c++/etui Drivers

Example

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
```

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
//** compute gradient with dco/c++ adjoint mode by default
auto grad  = E.gradient();
```

dco/c++/etui Drivers

Example

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
//** compute gradient with dco/c++ adjoint mode by default
auto grad  = E.gradient();
//** compute Hessian with dco/c++ tangent vector over adjoint mode
auto hess  = E.hessian<dco::ga1s<dco::gt1v<double, 5>::type>::type>();
```

dco/c++/etui Drivers

Example

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
//** compute gradient with dco/c++ adjoint mode by default
auto grad  = E.gradient();
//** compute Hessian with dco/c++ tangent vector over adjoint mode
auto hess  = E.hessian<dco::ga1s<dco::gt1v<double, 5>::type>::type>();
```

- defining problem `f`: generic lambda or templated functor

dco/c++/etui Drivers

Example

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
//** compute gradient with dco/c++ adjoint mode by default
auto grad  = E.gradient();
//** compute Hessian with dco/c++ tangent vector over adjoint mode
auto hess  = E.hessian<dco::ga1s<dco::gt1v<double, 5>::type>::type>();
```

- defining problem `f`: generic lambda or templated functor

```
//** generic lambda
auto f = [](auto & x, auto & y) { /* ... code ... */ };
```

- drivers via etui object

```
//** create etui object (stores references to in- and outputs)
double x(2.0), y;
auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(y), f);
//** run primal with given in- and outputs
E.primal();
//** compute gradient with dco/c++ adjoint mode by default
auto grad  = E.gradient();
//** compute Hessian with dco/c++ tangent vector over adjoint mode
auto hess  = E.hessian<dco::ga1s<dco::gt1v<double, 5>::type>::type>();
```

- defining problem `f`: generic lambda or templated functor

```
//** generic lambda
auto f = [](auto & x, auto & y) { /* ... code ... */ };

//** templated functor
struct F {
  template <typename T> void operator()(T & x, T & y)
       { /* ... code ... */ };
} f;
```

dco/c++/etui Drivers

More Complex Example

```
std::vector<Asset<double>> assets;
Curve<double> rate;
Matrix2D<double> Corr;
double finalMaturity;
BasketOption option;
int numPaths, numEulerSteps;
std::array<double,2> price_and_stdev;

auto f = [](auto &assets, auto &rate, auto &Corr, auto &finalMaturity,
            auto &price_and_stdev, auto &option, auto &numPaths,
            auto &numEulerSteps) {
              price_and_stdev = priceOption(option, assets, rate,
                                            Corr, numPaths, finalMaturity,
                                            numEulerSteps);
            };

auto E = dco::make_etui(
             dco::etui::in(assets, rate, Corr, finalMaturity),
             dco::etui::out(price_and_stdev),
             dco::etui::user_data(option, numPaths, numEulerSteps),
             f);

auto grad = E.gradient( [](auto &price_and_stdev) {
                            return price_and_stdev[0];
                        }
                      );
```

- written in C++17 as well
- currently supported design patterns
  - late recording
  - ensembles
  - evolutions
  - nonlinear solvers
  - more will be added in the future

- written in C++17 as well
- currently supported design patterns
  - late recording
  - ensembles
  - evolutions
  - nonlinear solvers
  - more will be added in the future
- works with and without `dco/c++/etui` drivers

- written in C++17 as well
- currently supported design patterns
  - late recording
  - ensembles
  - evolutions
  - nonlinear solvers
  - more will be added in the future
- works with and without `dco/c++/etui` drivers
- similarly generic in terms of number and type of parameters as the drivers

- algorithms are executed by

```
dco::etui::execute( dco::etui::in(...), dco::etui::out(...), f );
```

where `f` is the problem definition

- algorithms are executed by

```
dco::etui::execute( dco::etui::in(...), dco::etui::out(...), f );
```

  where `f` is the problem definition

- algorithms require different set of callbacks; general structure:

```
//** pseudo code
struct F : dco::etui::ALGORITHM {
  template <typename...>
        void CALLBACK1 (IN_T..., OUT_T..., UD_T...) { /* code */ }
  template <typename...>
        void CALLBACK2 (IN_T..., OUT_T..., UD_T...) { /* code */ }
};
```

  where (again) function templates or generic lambda definitions can be used

- algorithms are executed by

```
dco::etui::execute( dco::etui::in(...), dco::etui::out(...), f );
```

  where `f` is the problem definition

- algorithms require different set of callbacks; general structure:

```
//** pseudo code
struct F : dco::etui::ALGORITHM {
  template <typename...>
       void CALLBACK1 (IN_T..., OUT_T..., UD_T...) { /* code */ }
  template <typename...>
       void CALLBACK2 (IN_T..., OUT_T..., UD_T...) { /* code */ }
};
```

  where (again) function templates or generic lambda definitions can be used

- there are no restrictions on `F` other than
  callbacks callable with parameters and moveable

- implements loop with mutually independent iterations (like `std::for_each`)

dco/c++/etui Algorithms

Ensembles (I)

- implements loop with mutually independent iterations (like `std::for_each`)
- checkpointing and pathwise adjoints if on `dco::ga1[s|v][m]<...>::type`

dco/c++/etui Algorithms

Ensembles (I)

- implements loop with mutually independent iterations (like `std::for_each`)
- checkpointing and pathwise adjoints if on `dco::ga1[s|v][m]<...>::type`
- the problem definition is

```
struct F : dco::etui::ensemble</* loop index type */> {
  //** inherit constructors
  using ensemble::ensemble;
  //** loop body; gets all parameters and in addition loop index (i)
  static constexpr auto body  =
              [](auto& x, auto& y, int i) { /* code */ };
};
```

- implements loop with mutually independent iterations (like `std::for_each`)
- checkpointing and pathwise adjoints if on `dco::ga1[s|v][m]<...>::type`
- the problem definition is

```
struct F : dco::etui::ensemble</* loop index type */> {
  //** inherit constructors
  using ensemble::ensemble;
  //** loop body; gets all parameters and in addition loop index (i)
  static constexpr auto body  =
              [](auto& x, auto& y, int i) { /* code */ };
};
```

- it has the following constructor

```
ensemble(index_t const& lb, index_t const& ub);
```

`lb`: lower bound, `ub`: upper bound

dco/c++/etui Algorithms

Ensembles (II)

- the algorithm has the following modes:
  - `overload`:
    - default; plain overloading (record everything)
  - `pathwise`:
    - write a checkpoint during recording
    - pathwise adjoints during interpretation
  - `pathwise_early_propagation`:
    - propagate adjoints directly during recording
    - adjoints of the path outputs need to be known already
    - avoid checkpoint and second path evaluation
- the possible modes can be switched at run time

```
F f(0,n);
f.mode( f.pathwise );
dco::etui::execute( dco::etui::in(...), dco::etui::out(...), f );
```

Application to Simple Example

Driver (without dco/c++/etui)

```
int main() {
  size_t n = 4, num_mcpath = 10;

  //** initialize random numbers
  std::vector<double> r(num_mcpath);
  for (size_t i = 0; i < num_mcpath; i++)
    r[i] = static_cast<double>(rand()) / RAND_MAX;

  //** initialize parameters
  std::vector<double> x(n);
  for (size_t i = 0; i < n; i++) {
    if (n < 7)  x[i] = static_cast<double>(i)+1;
    else        x[i] = 1.00001;
  }

  //** run primal
  double res;


  double time = primal(x, res, r, num_mcpath);

  std::cout << "res  = " << res << std::endl;
  std::cout << "time = " << time << std::endl;
  return 0;
}
```

Application to Simple Example

Driver (with dco/c++/etui)

```
int main() {
  size_t n = 4, num_mcpath = 10;

  //** initialize random numbers
  std::vector<double> r(num_mcpath);
  for (size_t i = 0; i < num_mcpath; i++)
    r[i] = static_cast<double>(rand()) / RAND_MAX;

  //** initialize parameters
  std::vector<double> x(n);
  for (size_t i = 0; i < n; i++) {
    if (n < 7)  x[i] = static_cast<double>(i)+1;
    else        x[i] = 1.00001;
  }

  //** create etui-object and run primal
  double res;
  auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(res),
                          dco::etui::user_data(r, num_mcpath), F());
  E.primal();

  std::cout << "res  = " << res << std::endl;
  std::cout << E.statistics() << std::endl;
  return 0;
}
```

Application to Simple Example

Driver (with dco/c++/etui)

```
int main() {
  size_t n = 4, num_mcpath = 10;

  //** initialize random numbers
  std::vector<double> r(num_mcpath);
  for (size_t i = 0; i < num_mcpath; i++)
    r[i] = static_cast<double>(rand()) / RAND_MAX;

  //** initialize parameters
  std::vector<double> x(n);
  for (size_t i = 0; i < n; i++) {
    if (n < 7)  x[i] = static_cast<double>(i)+1;
    else        x[i] = 1.00001;
  }

  //** create etui-object and run primal
  double res;
  auto E = dco::make_etui(dco::etui::in(x), dco::etui::out(res),
                          dco::etui::user_data(r, num_mcpath), F());
  E.primal();
  auto grad = E.gradient();
  std::cout << "res   = " << res << std::endl;
  std::cout << E.statistics() << std::endl;
  return 0;
}
```

Application to Simple Example

Problem Definition (without dco/c++/etui)

```
template <typename T>
  void primal    (std::vector<T>       const& x, T &res,
                   std::vector<double> const& r, int num_mcpath) {
    T sum = 0.0;




    //** compute paths
    for (size_t i = 0; i < num_mcpath; i++) {
      f(x, sum, r, i);
    }


    res = sum / num_mcpath;
    res = pow(res, 2);

}
```

```
struct F {
template <typename T>
  void operator()(std::vector<T>      const& x, T &res,
                  std::vector<double> const& r, int num_mcpath) const {
    T sum = 0.0;

    //** declare / initialize problem definition
    auto m = mc_t(0, num_mcpath);


    //** execute algorithm
    dco::etui::execute(dco::etui::in(x),
                       dco::etui::out(sum),
                       dco::etui::user_data(dco::etui::omit_checkpoint(r)),
                       m);

    res = sum / num_mcpath;
    res = pow(res, 2);
  }
};
```

Application to Simple Example

Problem Definition (with dco/c++/etui)

```
struct F {
template <typename T>
  void operator()(std::vector<T>       const& x, T &res,
                  std::vector<double> const& r, int num_mcpath) const {
    T sum = 0.0;

    //** declare / initialize problem definition
    auto m = mc_t(0, num_mcpath);
    m.mode(m.pathwise);

    //** execute algorithm
    dco::etui::execute(dco::etui::in(x),
                       dco::etui::out(sum),
                       dco::etui::user_data(dco::etui::omit_checkpoint(r)),
                       m);

    res = sum / num_mcpath;
    res = pow(res, 2);
  }
};
```

```cpp


template <typename T>
  void f   (std::vector<T> const& x, T &sum, std::vector<double> const& r, int p) {
    size_t n = x.size();
    T y;
    for (size_t i = 0; i < n; i++) {
      if (i == 0) { y  = sin(x[i] * r[p]) * cos(1.0 + r[p]); }
      else        { y *= 0.3 + x[i] * sin(1.0 + r[p]);    }
    }
    sum += cos(y);

}
```

Application to Simple Example

Ensemble (with dco/c++/etui)

```
struct mc_t : dco::etui::ensemble<int> {
  using ensemble::ensemble;
  template <typename T>
    void body(std::vector<T> const& x, T &sum, std::vector<double> const& r, int p) const {
      size_t n = x.size();
      T y;
      for (size_t i = 0; i < n; i++) {
        if (i == 0) { y  = sin(x[i] * r[p]) * cos(1.0 + r[p]); }
        else        { y *= 0.3 + x[i] * sin(1.0 + r[p]);    }
      }
      sum += cos(y);
    }
};
```

dco/c++/etui State and Outlook

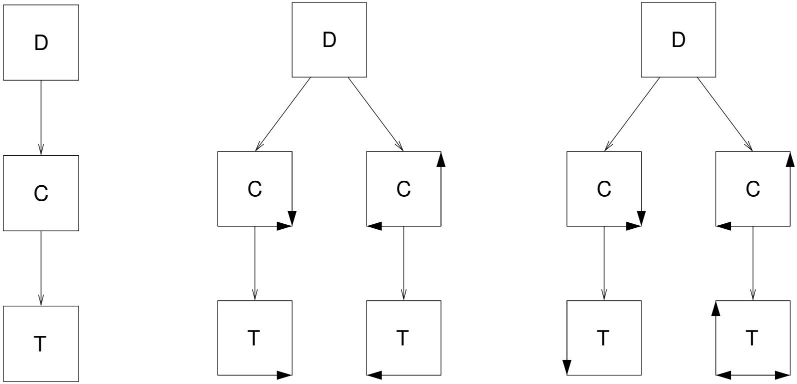
...

- ongoing development; eagerly seeking evaluators
- `dco/c++/etui` not yet part of `dco/c++` package
- independent of `dco/c++` version; should run with released package
- in the future
  - add more patterns
  - automatic switch to optimal mode (in drivers)
  - parallelism
  - lot of technical issues to work on (compile time, error messages, ...)

Outline

Late Recording

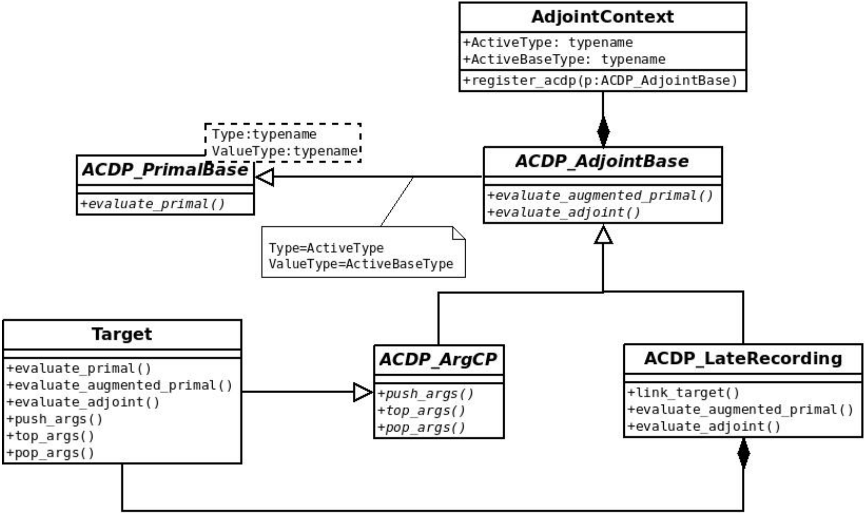
Argument Checkpointing



Call Tree | Early Recording | Late Recording

Late Recording

Adjoint Code Design Pattern



Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

naive adjoint

$$y_1 = \overrightarrow{F}_{(1)}(x, p_1)$$

$$y_2 = \overrightarrow{F}_{(1)}(x, p_2)$$

$$y = \frac{1}{2} \cdot (y_1 + y_2)$$

$$y_{2(1)} = y_{1(1)} = \frac{1}{2} \cdot y_{(1)}$$

$$\begin{pmatrix} x_{(1)} \\ p_{2(1)} \end{pmatrix} += \overleftarrow{F}_{(1)}(x, p_2, y_{2(1)})$$

$$\begin{pmatrix} x_{(1)} \\ p_{1(1)} \end{pmatrix} += \overleftarrow{F}_{(1)}(x, p_1, y_{1(1)})$$

pathwise adjoint

$$y_{2(1)} = y_{1(1)} = \frac{1}{2} \cdot y_{(1)}$$

$$y_1 = \overrightarrow{F}_{(1)}(x, p_1)$$

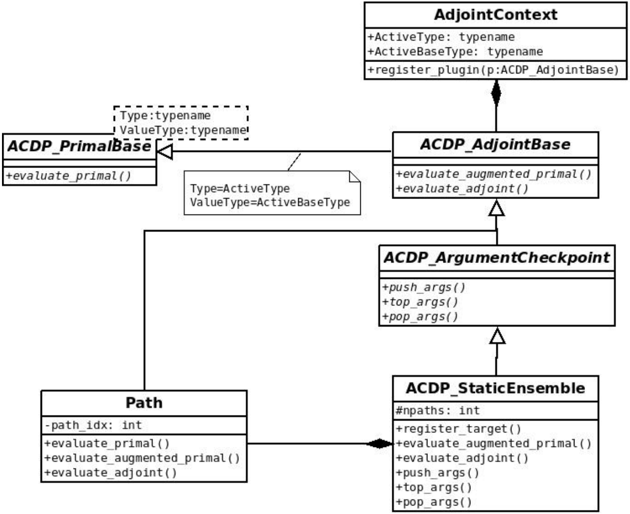
$$\begin{pmatrix} x_{(1)} \\ p_{2(1)} \end{pmatrix} += \overleftarrow{F}_{(1)}(x, p_2, y_{2(1)})$$

$$y_2 = \overrightarrow{F}_{(1)}(x, p_2)$$

$$\begin{pmatrix} x_{(1)} \\ p_{2(1)} \end{pmatrix} += \overleftarrow{F}_{(1)}(x, p_2, y_{2(1)})$$

$$y = \frac{1}{2} \cdot (y_1 + y_2)$$

Adjoint Code Design Pattern



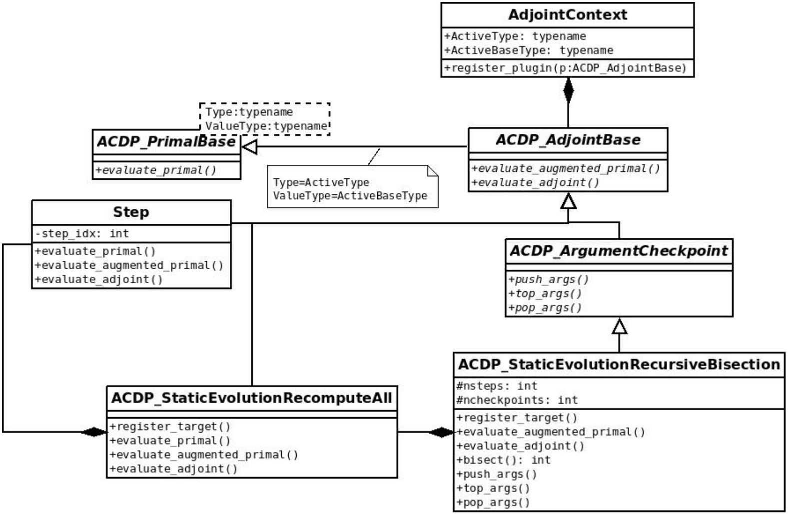
The minimal reevaluation cost of a reversal of an evolution $[f, t]$, $t > f$ with $c > 1$ checkpoints is equal to

$$C(f,t,c) = \min_{f<s\leq t}\left(\sum_{i=f}^{s} C_i + C(s,t,c-1) + C(f,s-1,c)\right)$$

for given step costs C_i, $i = f, \ldots t$ and

$$C(f,f,c) = 0 \quad \text{and} \quad C(f,t,1) = \sum_{i=f}^{t-1} \sum_{j=f+1}^{i} C_j \ .$$

- A. Griewank: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and Software, 1 (1), 35–54, 1992.



AdjointContext
+ActiveType: typename
+ActiveBaseType: typename
+register_plugin(p:ACDP_AdjointBase)
Type:typename
ValueType:typename
ACDP_PrimalBase
+evaluate_primal()
ACDP_AdjointBase
+evaluate_augmented_primal()
+evaluate_adjoint()
Type=ActiveType
ValueType=ActiveBaseType
Step
-step_idx: int
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()
ACDP_ArgumentCheckpoint
+push_args()
+top_args()
+pop_args()
ACDP_StaticEvolutionRecursiveBisection
#nsteps: int
#ncheckpoints: int
+register_target()
+evaluate_augmented_primal()
+evaluate_adjoint()
+bisect(): int
+push_args()
+top_args()
+pop_args()
ACDP_StaticEvolutionRecomputeAll
+register_target()
+evaluate_primal()
+evaluate_augmented_primal()
+evaluate_adjoint()

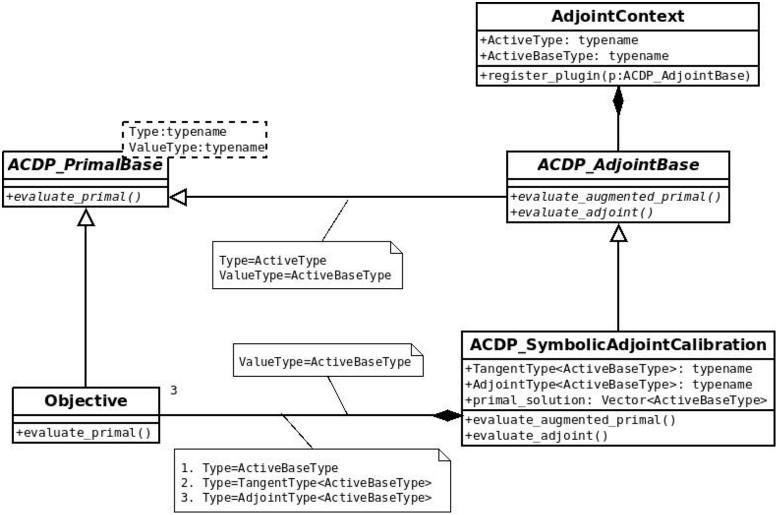
- nonlinear system: $F(x,p)=0 \Rightarrow x(p)$

$$p_{(1)} := -\frac{\partial F}{\partial p}^T \cdot \underbrace{\frac{dF}{dx}^{-T} \cdot x_{(1)}}_{z_{(1)}} .$$

- calibration: $\frac{df}{dx}(x,p)=0 \Rightarrow x(p)$

$$p_{(1)} := -\frac{\partial f^2}{\partial x \partial p}^T \cdot \underbrace{\frac{df^2}{dx^2}^{-1} \cdot x_{(1)}}_{z_{(1)}} .$$

- U. Naumann, J. Lotz, K. Leppkes, M. Towara: Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. ACM Transactions on Mathematical Software (TOMS) 41 (4), 1-21, 2015.


AdjointContext
+ActiveType: typename
+ActiveBaseType: typename
+register_plugin(p:ACDP_AdjointBase)
Type:typename
ValueType:typename
ACDP_PrimalBase
+evaluate_primal()
ACDP_AdjointBase
+evaluate_augmented_primal()
+evaluate_adjoint()
Type=ActiveType
ValueType=ActiveBaseType
ValueType=ActiveBaseType
ACDP_SymbolicAdjointCalibration
+TangentType<ActiveBaseType>: typename
+AdjointType<ActiveBaseType>: typename
+primal_solution: Vector<ActiveBaseType>
+evaluate_augmented_primal()
+evaluate_adjoint()
Objective
+evaluate_primal()
3
1. Type=ActiveBaseType
2. Type=TangentType<ActiveBaseType>
3. Type=AdjointType<ActiveBaseType>

Outline

	ERT (s)	RSS (mb)	UCI (%)
primal	0.3	4	-
central finite differences	60.1	4	-
tangent	63.0	4	-
adjoint (store-all)	1.1	577	-
adjoint (EarlyForwardFiniteDifferences, ncs=100)	29.7	5	48
adjoint (EarlyTangentPreaccumulation, ncs=100)	59.6	5	47
adjoint (LateRecording, ncs=100)	1.2	96	45
adjoint (RecursiveBisection, ncp=10)	2.6	5	37
adjoint (optimal RecursiveBisection, ncp=10)	2.3	5	32
adjoint (SymbolicAdjointLS, dense)	7.4	5772	27
adjoint (SymbolicAdjointNLS, sparse)	0.8	37	23

Example: Burgers equation (nx=100; nt=1000) as in U. N.: Adjoint code design patterns.

ERT: elapsed run time in seconds
RSS: resident set size in megabytes
UCI: user code index in percent of total source code

ncs: number of consecutive steps
ncp: number of checkpoints