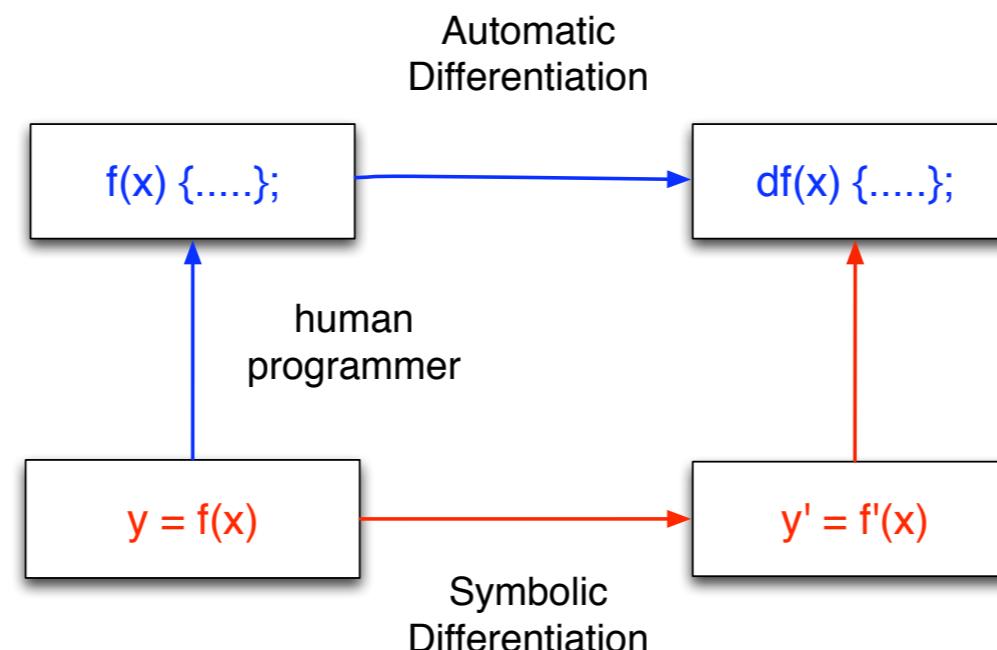


# ADIC2 : A component source transformation system for the differentiation of C/C++

Sri Hari Krishna Narayanan in collaboration with  
Paul Hovland  
Andrew Lyons (past member)  
Boyana Norris  
Jean Utke  
Beata Winnicka (past member)  
Mathematics & Computer Science Division  
Argonne National Laboratory

# AD in a Nutshell

- › Technique for computing analytic derivatives of programs (millions of loc)
- › Derivatives used in optimization, nonlinear PDEs, sensitivity analysis, inverse problems, etc.
- › AD = analytic differentiation of elementary functions + propagation by chain rule
  - Every programming language provides a limited number of elementary mathematical functions
  - Thus, every function computed by a program may be viewed as the composition of these so-called intrinsic functions
  - Derivatives for the intrinsic functions are known and can be combined using the chain rule of differential calculus
- › Associativity of the chain rule leads to two main modes: forward and reverse
- › Can be implemented using source transformation or operator overloading



# Modes of AD

- ▶ Forward Mode
  - Propagates derivative vectors, often denoted  $\nabla u$  or  $g_u$
  - Derivative vector  $\nabla u$  contains derivatives of  $u$  with respect to independent variables
  - Time and storage proportional to vector length (# indeps)
- ▶ Reverse Mode
  - Propagates adjoints, denoted  $\bar{u}$  or  $u_{\bar{u}}$
  - Adjoint  $\bar{u}$  contains derivatives of dependent variables with respect to  $u$
  - Propagation starts with dependent variables—must reverse flow of computation
  - Time proportional to adjoint vector length (# dependents)
  - Storage proportional to number of operations
  - Because of this limitation, often applied to subprograms

# Which mode to use?

- ▶ Use forward mode when
  - # independents is very small
  - Only a directional derivative ( $Jv$ ) is needed
  - Reverse mode is not tractable
- ▶ Use reverse mode when
  - # dependents is very small
  - Only  $J^T v$  is needed

# Ways of Implementing AD

- ▶ Operator Overloading
  - Use language features to implement differentiation rules or to generate trace (“tape”) of computation
  - Implementation can be very simple
  - Difficult to go beyond one operation/statement at a time in doing AD
  - Potential overhead due to compiler-generated temporaries,  
e.g.  $w = x^*y^*z \rightarrow tmp = x^*y; w = tmp^*z.$
  - Examples: ADOL-C, ADMAT, SACADO
- ▶ Source Transformation
  - Requires significant (compiler) infrastructure
  - More flexibility in exploiting chain rule associativity
  - Examples: ADIFOR, ADIC, OpenAD, TAF, TAPENADE

# Operator Overloading: simple example (implementation)

```
class a_double{
private:
    double value, grad;
public:
/* constructors */
a_double(double v=0.0, double g=0.0){value=v; grad = g;}

/* operators */
friend a_double operator+(const a_double &g1, const a_double &g2) {
    return a_double(g1.value+g2.value,g1.grad+g2.grad);
}
friend a_double operator-(const a_double &g1, const a_double &g2) {
    return a_double(g1.value-g2.value,g1.grad-g2.grad);
}
friend a_double operator*(const a_double &g1, const a_double &g2) {
    return a_double(g1.value*g2.value,g2.value*g1.grad+g1.value*g2.grad);
}
friend a_double sin(const a_double &g1) {
    return a_double(sin(g1.value),cos(g1.value)*g1.grad);
}
friend a_double cos(const a_double &g1) {
    return a_double(cos(g1.value),-sin(g1.value)*g1.grad);
}
// ...
```

# Operator Overloading: simple example (use)

```
#include <math.h>

void func(double *f, double x, double y){
    double a,b;

    if (x > y) {
        a = cos(x);
        b = sin(y)*y*y;
    } else {
        a = x*sin(x)/y;
        b = exp(y);
    }
    *f = exp(a*b);
}
```

```
#include <math.h>
#include "adouble.hxx"

void func(a_double *f, a_double x, a_double y){
    a_double a,b;

    if (x > y) {
        a = cos(x);
        b = sin(y)*y*y;
    } else {
        a = x*sin(x)/y;
        b = exp(y);
    }
    *f = exp(a*b);
}
```

# Source Transformation: simple example

C        Generated by TAPENADE            (INRIA, Tropics team)

C ...

```
SUBROUTINE FUNC_D(f, fd, x, xd, y, yd)
DOUBLE PRECISION f, fd, x, xd, y, yd
DOUBLE PRECISION a, ad, arg1, arg1d, b, bd
INTRINSIC COS, EXP, SIN
```

C

```
IF (x .GT. y) THEN
  ad = -(xd*SIN(x))
  a = COS(x)
  bd = yd*COS(y)*y*y + SIN(y)*(yd*y+y*yd)
  b = SIN(y)*y*y
ELSE
  ad = ((xd*SIN(x)+x*xd*COS(x))*y-x*SIN(x)*yd)/y**2
  a = x*SIN(x)/y
  bd = yd*EXP(y)
  b = EXP(y)
END IF
arg1d = ad*b + a*bd
arg1 = a*b
fd = arg1d*EXP(arg1)
f = EXP(arg1)
RETURN
END
```

# TAPENADE reverse mode: simple example

C Generated by TAPENADE (INRIA, Tropics team)  
C Version 2.0.6 - (Id: 1.14 vmp Stable - Fri Sep 5 14:08:23 MEST 2003)  
C ...

```
SUBROUTINE FUNC_B(f, fb, x, xb, y, yb)
DOUBLE PRECISION f, fb, x, xb, y, yb
DOUBLE PRECISION a, ab, arg1, arg1b, b, bb
INTEGER branch
INTRINSIC COS, EXP, SIN
```

```
C
IF (x .GT. y) THEN
  a = COS(x)
  b = SIN(y)*y*y
  CALL PUSHINTEGER4(0)
ELSE
  a = x*SIN(x)/y
  b = EXP(y)
  CALL PUSHINTEGER4(1)
END IF
arg1 = a*b
f = EXP(arg1)
arg1b = EXP(arg1)*fb
ab = b*arg1b
bb = a*arg1b
CALL POPINTEGER4(branch)
IF (branch .LT. 1) THEN
  yb = yb + (SIN(y)*y+y*SIN(y)+y*y*COS(y))*bb
  xb = xb - SIN(x)*ab
ELSE
  yb = yb + EXP(y)*bb - x*SIN(x)*ab/y**2
  xb = xb + (x*COS(x)/y+SIN(x)/y)*ab
END IF
fb = 0.D0
END
```

# Tools

- Fortran 95
- C/C++
- Fortran 77
- MATLAB
- Other languages: Ada, Python, ...
- More tools at <http://www.autodiff.org/>



# Tools: Fortran 95

- ▶ TAF (FastOpt)
  - Commercial tool
  - Support for (almost) all of Fortran 95
  - Used extensively in geophysical sciences applications
- ▶ Tapenade (INRIA)
  - Support for many Fortran 95 features
  - Developed by a team with extensive compiler experience
- ▶ OpenAD/F (Argonne/UChicago/Rice)
  - Support for many Fortran 95 features
  - Developed by a team with expertise in combinatorial algorithms, compilers, software engineering, and numerical analysis
  - Development driven by climate model & astrophysics code
- ▶ All three: forward and reverse; source transformation

## Tools: C/C++

- ▶ ADOL-C (Dresden)
  - Mature tool
  - Support for all of C++
  - Operator overloading; forward and reverse modes
- ▶ ADIC 2 (Argonne/UChicago)
  - Support for all of C, some C++ : All of C/C++ planned
  - Source transformation; forward mode, reverse mode
- ▶ TAPENADE (INRIA)
  - Source transformation; Support for C
- ▶ SACADO:
  - Operator overloading; forward and reverse modes
- ▶ TAC++ (FastOpt)
  - Commercial tool (under development)
  - Support for much of C/C++
  - Source transformation; forward and reverse modes

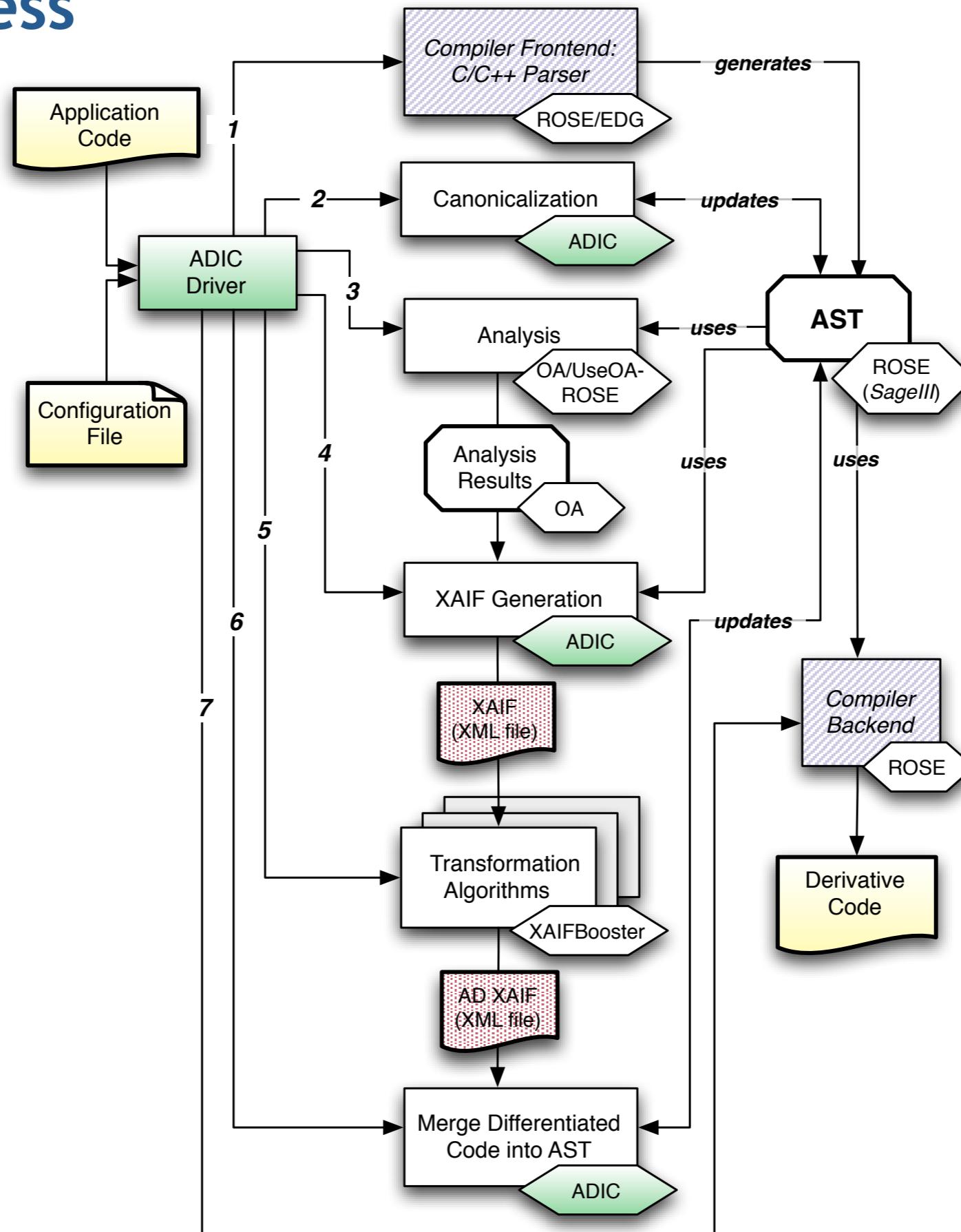
## Tools: Fortran 77

- ADIFOR (Rice/Argonne)
  - Mature and very robust tool
  - Support for all of Fortran 77
  - Forward and (adequate) reverse modes
  - Hundreds of users; ~150 citations

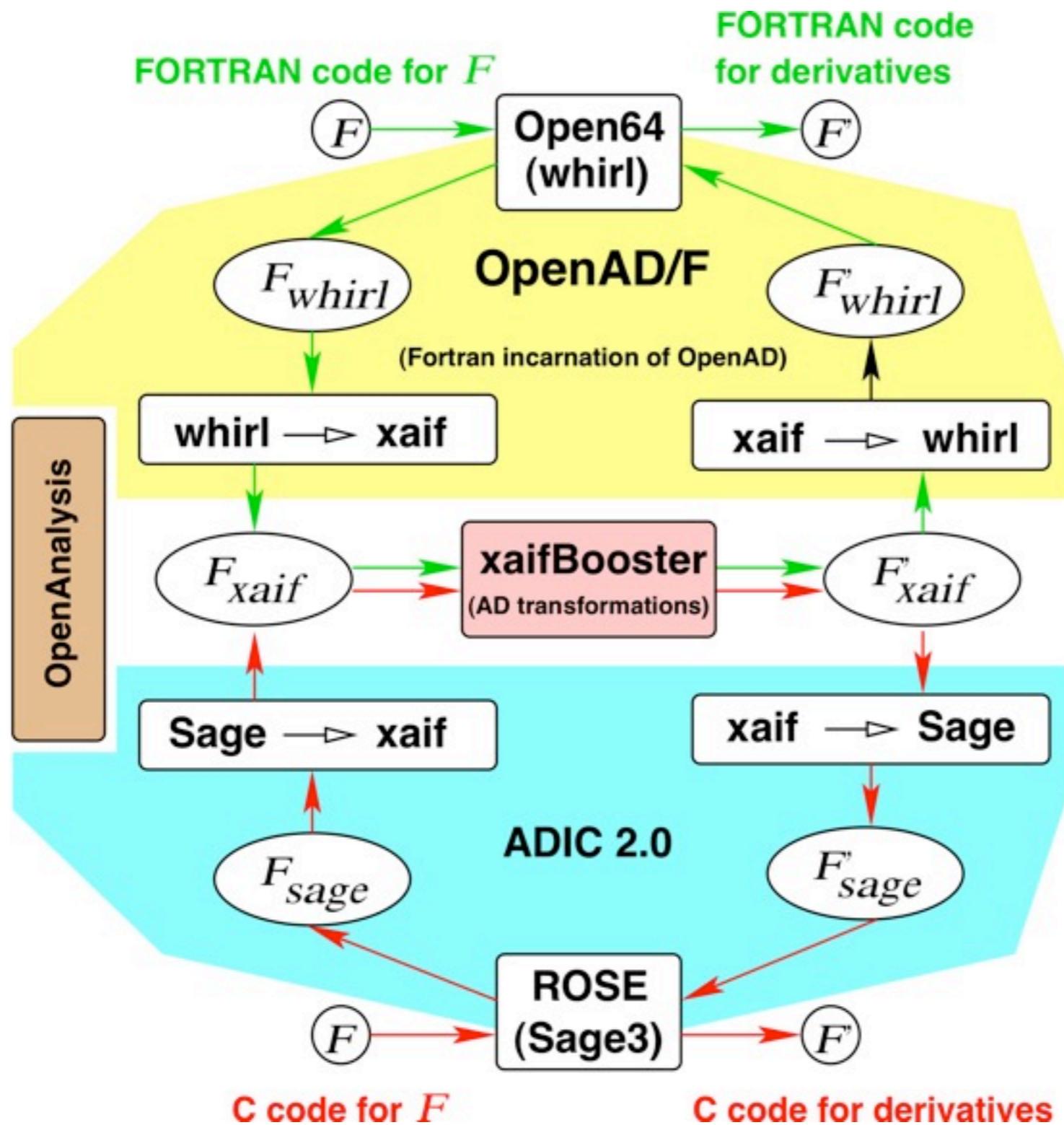
## Tools: MATLAB

- AdiMat (Aachen): source transformation
- MAD (Cranfield/TOMLAB): operator overloading
- Various research prototypes

# ADIC Process



# Shared OpenAD/ADIC system architecture



# ADIC Usage



DOE ACTS Workshop 2010

# ADIC Usage

ANSI-C or  
C++ Code

Configuration  
File



## **ADIC Usage**

### **[INACTIVE\_FUNCTIONS]**

**exit**

**creat**

**open**

**close**

### **[INTRINSIC\_FUNCTIONS]**

**log**

**sqrt**

**cos**

**asin**

### **[INACTIVE\_TYPES]**

**int**



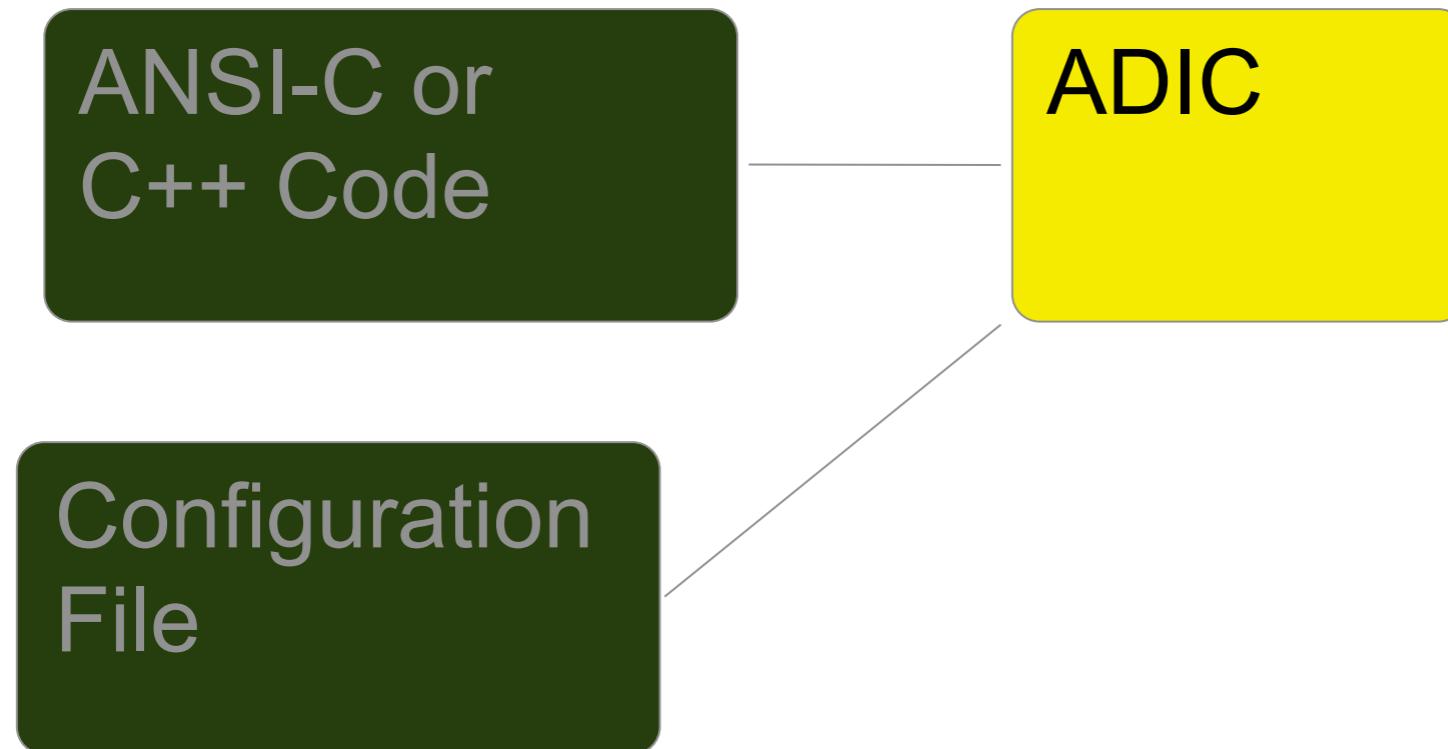
# ADIC Usage

ANSI-C or  
C++ Code

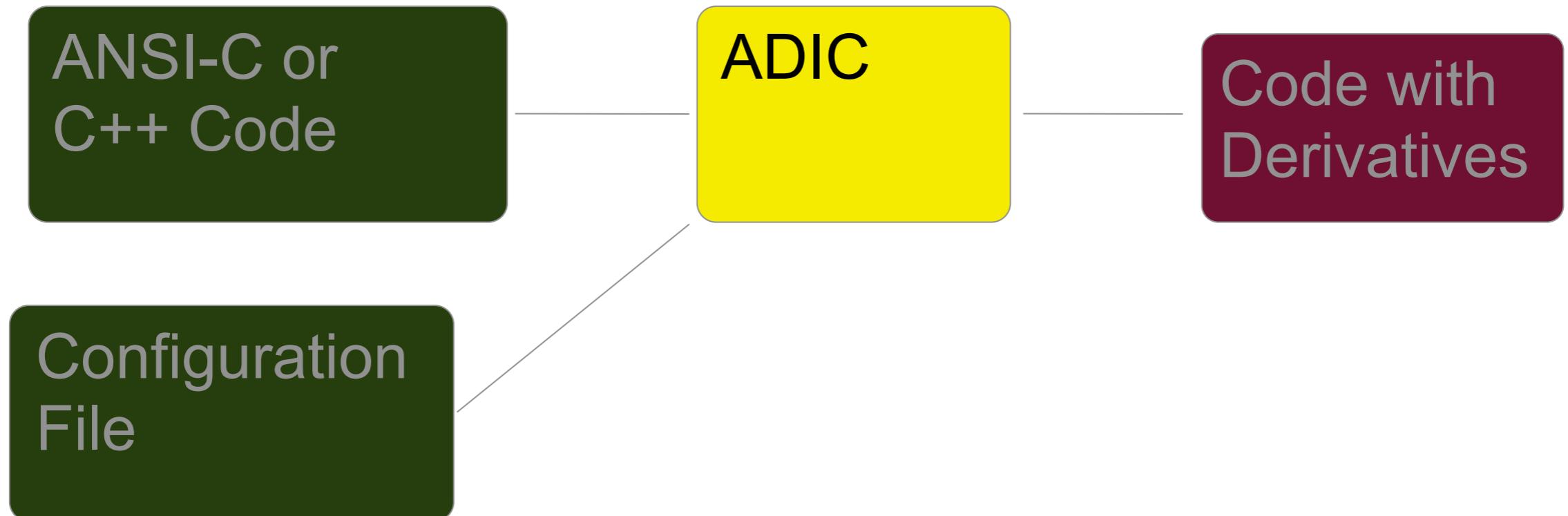
Configuration  
File



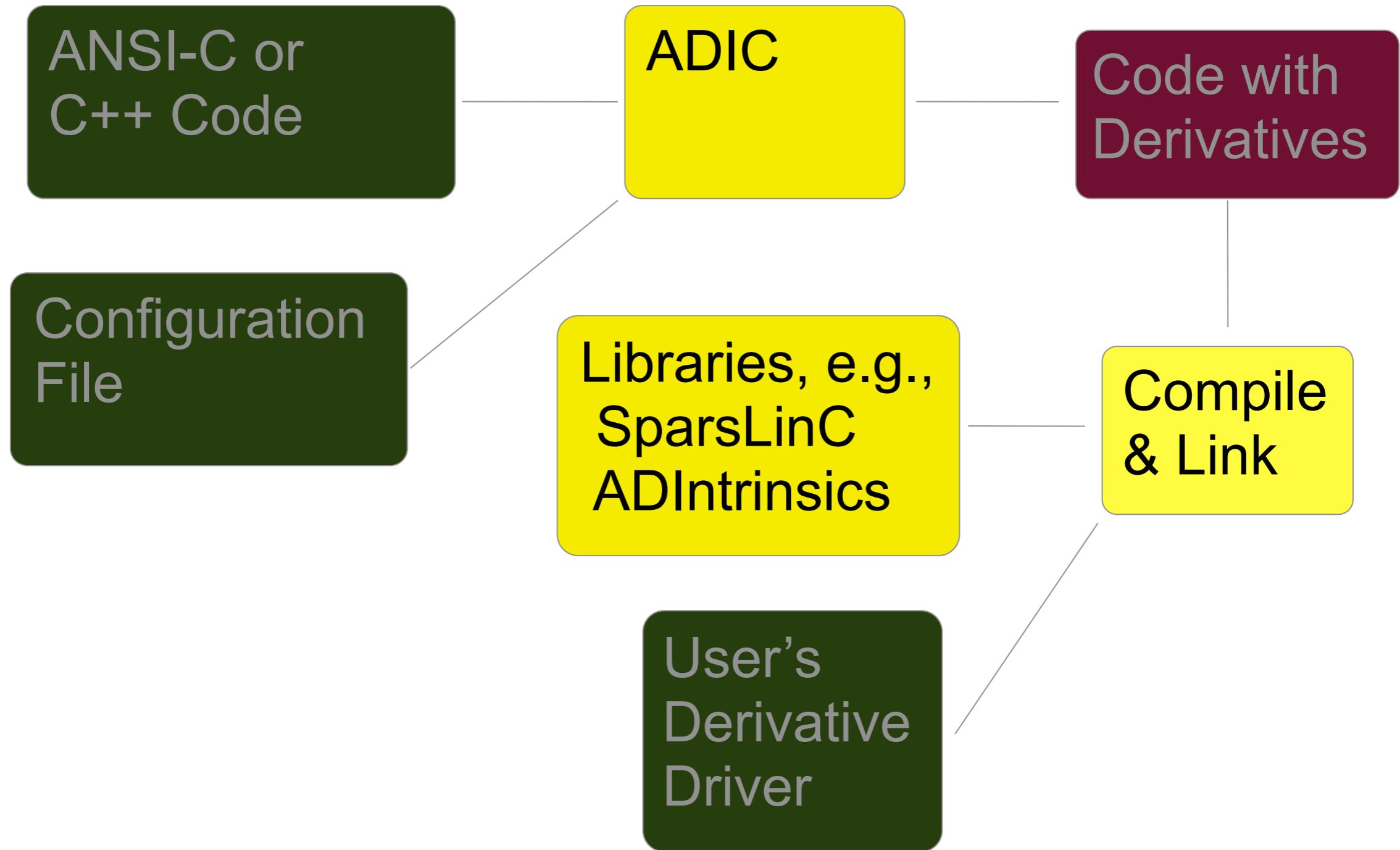
# ADIC Usage



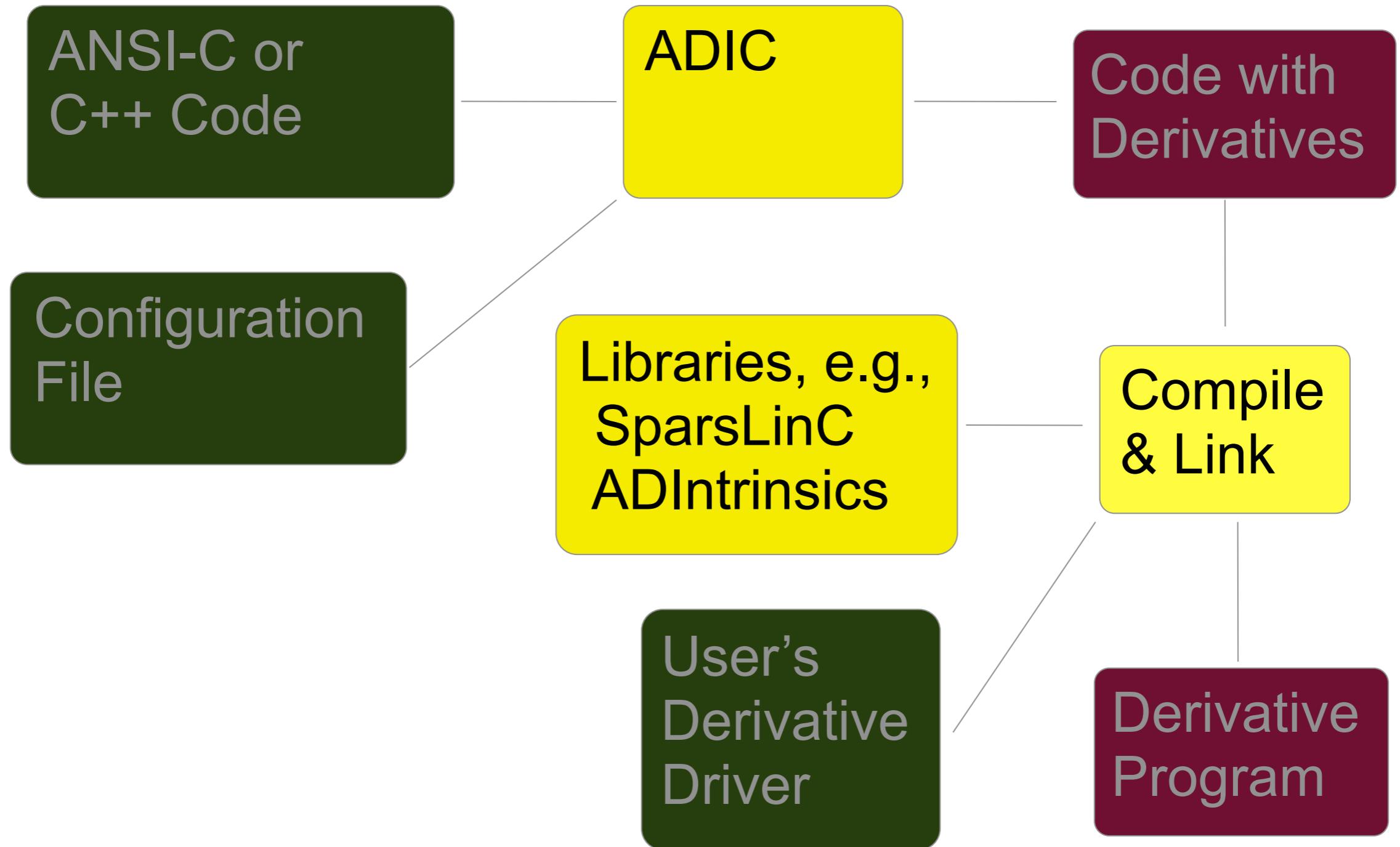
# ADIC Usage



# ADIC Usage



# ADIC Usage



# Forward mode ADIC-Generated Code: Interpretation

```
void mini1(double *y, double x)
{
    *y = x + sin(x * x);
}
```

- ▶ Output will include the original code as well as derivate code
- ▶ Types of ‘active’ variable will have changed
- ▶ Macros are used to manipulate the derivative values

```
typedef struct {
    double val;
    double grad[ADIC_GRADVEC_LENGTH];
} DERIV_TYPE;
```

DERIV\_val(y): value of program variable y  
DERIV\_grad(y): derivative object associated with y

```
#define ADIC_SetDeriv(__adic_src, __adic_tgt)
#define ADIC_IncDeriv(__adic_src, __adic_tgt)
#define ADIC_Sax(__adic_ca, __adic_src, __adic_tgt)
#define ADIC_Saxpy(__adic_ca, __adic_src, __adic_tgt)
```

# Reverse mode ADIC-Generated Code: Interpretation

```
void mini1(double *y, double x)
{
    *y = x + sin(x * x);
}
```

- ▶ Output will contain 3 cases
  - Original, taping and adjoint
- ▶ Taping case pushes values onto the stack which are popped by the adjoint case

```
void ad_mini1(DERIV_TYPE *y, DERIV_TYPE *x)
{
    if ((our_rev_mode.plain) == 1) {
        // ...Original function code
    }
    else if ((our_rev_mode.tape) == 1) {
        // ...Taping case
    }
    else if ((our_rev_mode.adjoint) == 1) {
        // ... Adjoint case
    }
}
```

# ADIC Driver Creation - Forward

```
void mini1(double *y, double x)
{
    *y = x + sin(x * x);
}
```

```
int main()
{
    double x = 0.5, y;
    DERIV_TYPE ad_x, ad_y;
    int i;

    /* Initialization Macro */
    ADIC_Init();

    /* Set the AD mode */
    ADIC_SetForwardMode();

    /* Specify the independent variable */
    ADIC_SetIndep(ad_x);

    /* Specify the dependent variable */
    ADIC_SetDep(ad_y);

    /* Done with specifications*/
    ADIC_SetIndepDone();

    /*Initialize the value of the independent variable ad_x */
    DERIV_val(ad_x) = x;

    /* Invoke AD function */
    ad_mini1(&ad_y, &ad_x);

    /* Extract the gradient */
    double temp_adj;
    temp_adj = 0.0;
    for (i = 0; i < ADIC_GRADVEC_LENGTH; i++) {
        temp_adj += DERIV_grad(ad_y)[i];
    }

    /* Optional output */

    /* End Macro */
    ADIC_Finalize();

    return 0;
}
```

# ADIC Driver Creation - Reverse

```
void mini1(double *y, double x)
{
    *y = x + sin(x * x);
}
```

```
int main()
{
    double x = 0.5, y;
    DERIV_TYPE ad_x, ad_y;
    int i;

    /* Initialization Macro */
    ADIC_Init();
    __ADIC_TapeInit();

    /* Set the AD mode */
    ADIC_SetReverseMode();

    /* Specify the independent variable */
    ADIC_SetIndep(ad_x);

    /* Specify the dependent variable */
    ADIC_SetDep(ad_y);

    /* Done with specifications*/
    ADIC_SetIndepDone();

    /*Initialize the value of the independent variable ad_x */
    DERIV_val(ad_x) = x;

    /* Invoke AD function in taping mode (forward sweep)*/
    our_rev_mode.tape = 1;
    ad_mini1(&ad_y, &ad_x);

    /* Invoke AD function in the adjoint mode (reverse sweep) */
    our_rev_mode.tape = 0;
    our_rev_mode.adjacent = 1;
    ad_mini1(&ad_y, &ad_x);

    /* Extract the gradient */
    double temp_adj;
    temp_adj = 0.0;
    for (i = 0; i < ADIC_GRADVEC_LENGTH; i++) {
        temp_adj += DERIV_grad(ad_x)[i];
    }

    /* End Macro */
    ADIC_Finalize();
    return 0;
}
```

# ADIC Driver Creation - Forward mode

```
void mini1(double *y, double *x)
{
    int i;
    for (i = 0; i < 2; i=i+1) {
        y[i] = x[i] + sin(x[i]*x[i]);
    }
}
```

```
int main()
{
    double x[ARRAY_SIZE], y[ARRAY_SIZE];
    DERIV_TYPE ad_x[ARRAY_SIZE], ad_y[ARRAY_SIZE];
    int i,j;

    for (i = 0; i < ARRAY_SIZE; i++){
        x[i] = 0.5;
    }
    /* Initialization Macro */
    ADIC_Init();
    /* Set the AD mode */
    ADIC_SetForwardMode();

    /* Specify the dependent variable */
    ADIC_SetDepArray(ad_y, ARRAY_SIZE);
    /* Specify the independent variable */
    ADIC_SetIndepArray(ad_x, ARRAY_SIZE);
    /* Done with specifications*/
    ADIC_SetIndepDone();

    // Initialize the value of the independent variable ad_x
    for (i = 0; i < ARRAY_SIZE; i++){
        DERIV_val(ad_x[i])=x[i];
    }

    // Invoke AD function
    ad_mini1(ad_y, ad_x);

    /* Extract the gradient */
    double temp_adj;
    for (i = 0; i < ARRAY_SIZE; i++) {
        temp_adj = 0.0;
        for (j = 0; j < ADIC_GRADVEC_LENGTH; j++) {
            temp_adj += DERIV_grad(ad_y[i])[j];
        }
        /* Optional output */
        printf("AD result is: \t\t[%lf]\t", temp_adj);
    }

    /* End Macro */
    ADIC_Finalize();

    return 0;
}
```

# ADIC Driver Creation - Reverse mode

```
void mini1(double *y, double *x)
{
    int i;
    for (i = 0; i < 2; i=i+1) {
        y[i] = x[i] + sin(x[i]*x[i]);
    }
}
```

```
int main()
{
    double x[ARRAY_SIZE], y[ARRAY_SIZE];
    DERIV_TYPE ad_x[ARRAY_SIZE], ad_y[ARRAY_SIZE];
    int i,j;

    for (i = 0; i < ARRAY_SIZE; i++){
        x[i] = 0.5;
    }
    /* Initialization Macro */
    ADIC_Init();
    __ADIC_TapeInit();
    /* Set the AD mode */
    ADIC_SetReverseMode();
    /* Specify the dependent variable */
    ADIC_SetDepArray(ad_y, ARRAY_SIZE);
    /* Specify the independent variable */
    ADIC_SetIndepArray(ad_x, ARRAY_SIZE);
    /* Done with specifications*/
    ADIC_SetIndepDone();

    // Initialize the value of the independent variable ad_x
    for (i = 0; i < ARRAY_SIZE; i++){
        DERIV_val(ad_x[i]) =x[i];
    }

    /* Invoke AD function in taping mode (forward sweep)*/
    our_rev_mode.tape = 1;
    ad_mini1(ad_y, ad_x);
    /* Invoke AD function in the adjoint mode (reverse sweep) */
    our_rev_mode.tape = 0;
    our_rev_mode.adj = 1;
    ad_mini1(ad_y, ad_x);

    double temp_adj;
    for (i = 0; i <ARRAY_SIZE; i++) {
        temp_adj = 0.0;
        for (j = 0; j <ADIC_GRADVEC_LENGTH; j++) {
            temp_adj += DERIV_grad(ad_x[i])[j];
        }
        printf("AD result is: \t\t[%lf]\t", temp_adj);
    }

    ADIC_Finalize();

    return 0;
}
```

# What is feasible & practical

- › Jacobian matrices are often sparse
- › The forward mode of AD computes  $J \times S$ , where  $S$  is usually an identity matrix or a vector
- › Key point: forward mode computes  $JS$  at cost proportional to number of columns in  $S$ ; reverse mode computes  $J^T W$  at cost proportional to number of columns in  $W$
- › Jacobians of functions with small number (1–1000) of independent variables (forward mode,  $S=I$ )
- › Jacobians of functions with small number (1–100) of dependent variables (reverse/adjoint mode,  $S=I$ )
- › Very (extremely) large, but (very) sparse Jacobians and Hessians (forward mode plus coloring)
- › Jacobian-vector products (forward mode)
- › Transposed-Jacobian-vector products (adjoint mode)
- › Hessian-vector products (forward + adjoint modes)
- › Large, dense Jacobian matrices that are effectively sparse or effectively low rank (e.g., see Abdel-Khalik et al., AD2008)

# Issues with Black Box Differentiation

- Source code may not be available or may be difficult to work with
- Simulation may not be (chain rule) differentiable
  - Feedback due to adaptive algorithms
  - Nondifferentiable functions
  - Noisy functions
  - Convergence rates
  - Etc.
- Accurate derivatives may not be needed (FD might be cheaper)
- Differentiation and discretization do not commute

# Difficulties encountered in application of ADIFOR 2.0

- ▶ Dubious programming techniques:
  - Type mismatches in actual & declared parameters
- ▶ Bugs:
  - inconsistent number of arguments in subroutine calls
- ▶ Not conforming to Fortran77 standard
  - while statement in one subroutine
- ▶ ADIFOR2.0 limitations:
  - I/O statements containing function invocations

# Points of Nondifferentiability

- ▶ Due to intrinsic functions
  - Several intrinsic functions are defined at points where their derivatives are not, e.g.:
    - $\text{abs}(x)$ ,  $\sqrt{x}$  at  $x=0$
    - $\max(x,y)$  at  $x=y$
  - Requirements:
    - Record/report exceptions
    - Optionally, continue computation using some generalized gradient
  - ADIFOR/ADIC approach
    - User-selected reporting mechanism
    - User-defined generalized gradients, e.g.:
      - [1.0,0.0] for  $\max(x,0)$
      - [0.5,0.5] for  $\max(x,y)$
    - Various ways of handling
      - Verbose reports (file, line, type of exception)
      - Terse summary (like IEEE flags)
      - Ignore
- ▶ Due to conditional branches
  - May be able to handle using trust regions

# Implicitly Defined Functions

- Implicitly defined functions often computed using iterative methods
- Function and derivatives may converge at different rates
- Derivative may not be “accurate” if iteration halted upon function convergence
- Solutions:
  - Tighten function convergence criteria
  - Add derivative convergence to stopping criteria
  - Compute derivatives directly, e.g.  $A \nabla x = \nabla b$

# Addressing Limitations in Black Box AD

- Detect points of nondifferentiability
  - proceed with a subgradient
  - currently supported for intrinsic functions, but not conditional statements
- Exploit mathematics to avoid differentiating through an adaptive algorithm
- Modify termination criterion for implicitly defined functions
  - Tighten tolerance
  - Add derivatives to termination test (preferred)
- There are some potential “gotchas” when applying AD in a black box fashion
- Some care should be taken to ensure that the desired quantity is computed
- There are usually workarounds

# Conclusions & Future Work

- ADIC2 release expected very shortly.
- Working with several researchers' applications.
- Greater C++ handling will be incrementally added.
- Checkpointing for reverse mode is needed.
- Hessians are not currently handled.

## For More Information

- Andreas Griewank, Evaluating Derivatives, SIAM, 2000.
- Griewank, “On Automatic Differentiation”; this and other technical reports available online at: [http://www.mcs.anl.gov/autodiff/tech\\_reports.html](http://www.mcs.anl.gov/autodiff/tech_reports.html)
- AD in general: <http://www.mcs.anl.gov/autodiff/>, <http://www.autodiff.org/>
- ADIFOR: <http://www.mcs.anl.gov/adifor/>
- ADIC: <http://www.mcs.anl.gov/adic/>
- OpenAD: <http://www.mcs.anl.gov/openad/>
- Other tools: <http://www.autodiff.org/>
- E-mail: [hovland@mcs.anl.gov](mailto:hovland@mcs.anl.gov); [snarayan@mcs.anl.gov](mailto:snarayan@mcs.anl.gov); [norris@mcs.anl.gov](mailto:norris@mcs.anl.gov) ;  
[utke@mcs.anl.gov](mailto:utke@mcs.anl.gov);