aderrors: Error analysis of Monte Carlo data with Automatic Differentiation.

Alberto ramos <alberto.ramos@maths.tcd.ie>

Abstract

aderrors is a fortran 2008 library for error analysis of MC data. It uses the Γ-method together with techniques of automatic differentiation to provide a robust, efficient, portable, transparent and open source module for the analysis of Monte Carlo data [1].

Contents

1 Features

- **Exact** linear error propagation, even in iterative algorithms (i.e. error propagation in fit parameters) thanks to Automatic Differentiation.

- Handles data from **any number of ensembles** (i.e. simulations with different parameters).
- Support for **replicas** (i.e. several runs with the same simulation parameters).
- Standalone **portable** implementation without any external dependencies.
- **Fast** computation of autocorrelation functions with the FFT package [2] (included in the distribution).
- **Exact** determination of gradients and Hessians of arbitrary functions.

2 Installing and linking with the library

2.1 Installing

1. Download or clone the repository at [1].
2. Edit the `Makefile` in the `build` directory. Change according to your needs. Usually only the compiler command/options (variables `FC` and `FOPT`) have to be changed.
3. Compile the library with `gmake`.
4. Optionally build/run the test codes with `gmake test` . Executabes will be placed in the `test` directory.
5. If preferred, move the contents of the `include` and `lib` directories somewhere else.

2.2 Using the library

- Compile your programs with `-I <dir>/include`.
- Link your programs with the `-L <dir>/lib` and `-laderr` options.

3 Example usage

This is a collection of simple examples on the usage of the `module aderrors`. Note that the basic data type is `uwreal`, that is able to handle MC histories and data with errors. The error is determined by calling the method `uwerr` on the data type.

In these examples we use the `module simulator` (`test/simulator.f90`) to generate autocorrelated data.

3.1 Simple analysis of MC data

Estimating the value and statistical error of a quantity given some MC data is really very simple. The following example is available in `test/simple.f90`.

```
1 program simple
2
3   use numtypes
4   use aderrors
5   use simulator
6
```

```
  implicit none

  integer, parameter  :: nd = 20000
  type (uwreal)       :: x
  real (kind=DP)      :: data_x(nd), err, ti, tau(5), lam(5)


  ! Fill array data_x(:) with autocorrelated
  ! data from the module simulator.
  tau = (/1.0_DP, 3.0_DP, 4.0_DP, 5.0_DP, 7.34_DP/)
  lam = (/1.00_DP, 0.87_DP, 1.23_DP, 0.56_DP, 0.87_DP/)

  call gen_series(data_x,  err, ti, tau, lam, 0.3_DP)

  ! Load data_x(:) measurements in variable x. Use
  ! default settings (Stau=4, texp=0, 1 replica)
  x = data_x

  ! Perform error analysis (optimal window)
  call x%uwerr()

  ! Print results and compare with exact values
  write(*,'(1A,1I6,1A)')'** Measurements: ', nd, ' **   '
  write(*,100)'  - Gamma-method:  ', x%value(),  " +/- ", x%error(), '( tauint: ', &
       x%taui(1), " +/- ", x%dtaui(1), ')'
  write(*,100)'  - Exact:         ', 0.3_DP,     " +/- ", err, '( tauint: ', ti, " )"

100 FORMAT((2X,1A,1F8.5,1A,1F7.5,5X,1A,1F0.2,1A,1F7.5,1A))

  stop
end program simple
```

Note that we only need to load the MC data in a `uwreal` data type and call `uwerr()` to determine the mean value and error.

3.2 Complicated derived observables

Imagine that we have the MC data of some observable X. We are interested in a complicated quantity

$$f(\langle X\rangle) = \frac{\sin(\langle X\rangle)\,\cos(\langle X\rangle^2)}{1+\langle X\rangle^4}\,. \tag{1}$$

The question is what is the mean value and the error in $f(\langle X\rangle)$? The example shows how to do such a computations (`test/derived.f90`)

```
program derived

  use ISO_FORTRAN_ENV, Only : error_unit, output_unit
  use numtypes
  use aderrors
  use simulator
```

```
7
8     implicit none
9
10    integer, parameter  :: nd = 20000
11    type (uwreal)       :: x, y
12    real (kind=DP)      :: data_x(nd), err, ti, tau(5), lam(5)
13
14
15    ! Fill arrays data_x(:) with autocorrelated
16    ! data from the module simulator.
17    tau = (/1.0_DP, 3.0_DP, 4.0_DP, 5.0_DP, 7.34_DP/)
18    lam = (/1.00_DP, 0.87_DP, 1.23_DP, 0.56_DP, 0.87_DP/)
19
20    call gen_series(data_x,  err, ti, tau, lam, 0.3_DP)
21
22    ! Load data_x(:) measurements in variable x. Use
23    ! default settings (Stau=4, texp=0, 1 replica)
24    x = data_x
25
26    ! Exact, transparent error propagation
27    y = sin(x)*cos(x**2)/(1.0_DP+x**4)
28    ! Perform error analysis (optimal window)
29    call y%uwerr()
30
31    ! Print results
32    write(*,'(1A,1I6,1A)')'** Measurements: ', nd, ' **    '
33    write(*,100)'  - Gamma-method:  ', y%value(),  " +/- ", y%error(), '( tauint: ', &
34         y%taui(1), " +/- ", y%dtaui(1), ')'
35
36  100 FORMAT((2X,1A,1F8.5,1A,1F7.5,5X,1A,1F0.2,1A,1F7.5,1A))
37
38
39    stop
40  end program derived
```

This computation requires to determine the derivative of the function $f(x)$. This derivative is determined using *automatic differentiation* [3]. This method avoids the numerical evaluation of the derivative by some variant of the finite difference approach, or by some stochastic evaluation of the derivative (as is done in jacknife/bootstrap). Making a long story short, the derivative of $f(x)$ is determined exactly (up to machine precision).

3.3 A calculator with uncertainties

The `module aderrors` does not only handle full MC histories, but also regular variables with errors. Consider the next example (`test/calculator.f90`)

```
1   program calculator
2
3     use numtypes
4     use aderrors
```

```
5
6    implicit none
7
8    type (uwreal) :: x, y, z, t
9
10   x = (/1.223_DP, 0.012_DP/) ! x = 1.223 +/- 0.012
11   y = sin(2.0_DP*x)
12   z = 1.0_DP + 2.0_DP * sin(x)*cos(x)
13   t = z - y
14   call t%uwerr()
15   write(*,'(1A,1F18.16,1A,1F18.16)')'Exactly one: ', &
16        t%value(), " +/- ", t%error() ! 1.0 +/- 0.0
17
18   stop
19 end program calculator
```

Note that not only the mean value of `t` is one, but also the error is zero (i.e. the correlation between different variables is correctly taken into account).

This is a trivial example, but the `module aderrors` handles correctly and automatically the correlations between different observables from the same MC chain (i.e. proton and pion masses measured in a ensemble).

3.4 Combining MC chains

In many cases we have to estimate quantities that depend on several MC ensembles. This case is handled transparently by just identifying the ensemble where each primary observable was measured with an integer ID.

The following example (`test/multi.f90`) combines in a derived observable data from different ensembles and also a simple quantity with uncertainties (these are treated as independent ensembles).

```
1  program multi
2
3    use ISO_FORTRAN_ENV, Only : error_unit, output_unit
4    use numtypes
5    use aderrors
6    use simulator
7
8    implicit none
9
10   integer, parameter :: nd = 200000
11   real (kind=DP)     :: data_x(nd), data_y(nd), err, ti, tau(5), lam(5), texp
12   type (uwreal)      :: x, y, z, t
13   integer            :: i
14
15   ! Fill array data_x(:) with autocorrelated data
16   ! from the module simulator. Save slowest mode in texp
17   tau  = (/1.0_DP, 3.0_DP, 4.0_DP, 5.0_DP, 73.4_DP/)
18   texp = maxval(tau)
19   lam  = (/1.00_DP, 0.87_DP, 1.23_DP, 0.56_DP, 0.087_DP/)
```

```
20    call gen_series(data_x,  err, ti, tau, lam, 0.3_DP)
21
22
23    ! Fill array data_y(:) with autocorrelated
24    ! data from the module simulator.
25    tau = (/1.4_DP, 2.4_DP, 1.8_DP, 5.1_DP, 7.14_DP/)
26    lam = (/2.00_DP, 0.85_DP, 1.33_DP, 2.56_DP, 1.87_DP/)
27    call gen_series(data_y,  err, ti, tau, lam, 0.3_DP)
28
29    ! Load data_x(:) in variable x and set ensemble id to 1.
30    ! Set exponential autocorrelation time to add tail
31    x = data_x
32    call x%set_id(1)
33    call x%set_texp(texp)
34    ! Set name of ensemble 1
35    i = aderr_new_id("Ensemble X", 1)
36
37
38    ! Load data_y(:) in variable x and set ensemble id to 2
39    y = data_y
40    call y%set_id(2)
41    ! Set name of ensemble 2
42    i = aderr_new_id("Ensemble Y", 2)
43
44    ! Observable z is just a variable with error.
45    ! Treat as another ensemble with id 3
46    z = (/3.23_DP, 0.23_DP/) ! z = 3.23(23)
47    call z%set_id(3)
48    ! Set name of ensemble
49    i = aderr_new_id("Variable z with errors", 3)
50
51    ! Exact, transparent error propagation.
52    t = z*x/y - y/(z*x)
53
54    ! Use Stau=1.75 for ensemble with ID 2
55    call t%set_stau(1.75_DP,2)
56    ! Error analysis (optimal window, tails, ...)
57    call t%uwerr()
58
59    write(*,*)"Result: ", t%value(), " +/- ", t%error()
60    do i = 1, t%neid()
61       write(*,200)trim(t%error_name(i)), &
62            ' contribution to error: ', t%error_src(i)*100., '%'
63    end do
64
65  200 FORMAT(1A22,1A,5X,1F6.2,1A)
66
67
68    stop
69  end program multi
```

This code produces the flowing output

```
Result:    2.6486091631414594       +/-   0.38165436636532096
Variable z with errors contribution to error:       38.34%
            Ensemble X contribution to error:       10.64%
            Ensemble Y contribution to error:       51.01%
```

Practical lessons from this example:

- The routine `aderr_new_id`, stores in a database the ID of a MC chain and an identification string.
- Independent ensembles are identified by assigning to the primary observables an integer ensemble ID.
- By default all observables are assumed to derive from a common ensemble with id=-1.
- Variables with errors can be combined with MC measurements. One just has to assign an ensemble ID to these variables as if they were a new ensemble.

3.5 Error propagation in iterative algorithms

We know some function

$$f(x) = a\cos(b\sin(x)) \tag{2}$$

where

$$a = 0.800(56), \qquad b = 1.30(10)\,. \tag{3}$$

are observables measured on different ensembles. We want to determine the *fixed point* of f (i.e. the value x such that $f(x) = x$) with uncertainty. A typical procedure is to use the newton method to find a root of $f(x) - x$. This is implemented in the most naive way (see `test/newton.f90`)

```
1  program newton
2
3    use numtypes
4    use aderrors
5
6    implicit none
7
8    type (uwreal)  :: xant, xnew, val, der, a, b
9    integer        :: id, i
10
11   a = (/ 0.8_DP, 0.056_DP /)
12   id = aderr_new_id("Variable a", 100)
13   call a%set_id(id)
14
15   b = (/ 1.3_DP, 0.10_DP /)
16   id = aderr_new_id("Variable b", 222)
17   call b%set_id(id)
18
```

```
19    xnew = (/0.50_DP, 10.0_DP/)
20    id = aderr_new_id("Initial condition", 3674)
21    call xnew%set_id(id)
22
23    ! Newton's method. Iterate until root does not change
24    do
25      xant = xnew
26      call fnew(xant, val, der)
27      xnew = xant - val/der
28      if (abs(xnew%value()-xant%value()).lt.1.0E-10_DP) exit
29    end do
30
31    call xnew%uwerr()
32    write(*,*)"Fixed point at: ", &
33              xnew%value(), " +/- ", xnew%error()
34    do i = 1, xnew%neid()
35       write(*,200)trim(xnew%error_name(i)), &
36            ' contribution to error: ', xnew%error_src(i)*100., '%'
37    end do
38
39  200 FORMAT(1A20,1A,5X,1F6.2,1A)
40
41    stop
42  contains
43    subroutine fnew(x, f, d)
44    ! Returns the value (f) an derivative (d) of
45    ! the function f(x)-x
46
47      type (uwreal), intent (in)  :: x
48      type (uwreal), intent (out) :: f, d
49
50      f =  a*cos(b*sin(x)) - x
51      d = -a*b*sin(b*sin(x)) * cos(x) - 1.0_DP
52
53      return
54    end subroutine fnew
55  end program newton
```

Note that the implementation of Newton's method (lines 23-29) is exactly the same as what one would write *without* having to deal with uncertainties.

In this case, even the "derivative of the Newton Method" (i.e. how much the position of the root changes when changing the parameters a and b in eq. (2)), is determined "exactly" thanks to the techniques of automatic diferentiation. In particular running this code gives as output

```
 Fixed point at:   0.59617758342692495       +/-    3.2638835812954813E-002
   Initial condition contribution to error:        0.00%
          Variable a contribution to error:       66.00%
          Variable b contribution to error:       34.00%
```

The contribution to the error in the position of the root due to the error of the initial condition is zero. That is, the techniques of automatic differentiation have been able

to pick the fact that the result of the newton method does not depend on the initial condition[1].

Practical lessons from this example:

- `id`'s of MC chains **do not** need to be consecutive. One can "encode" $\beta = 6.12$, $\kappa = 0.135734$ in `id = 612135734`, for example.

3.6 Compute derivatives of arbitrary functions

Let's look with more detail the previous example. We are going to wrap the newton method into a function that has three inputs: the value of a, b and the initial condition. The `module aderrors` is able to compute derivatives of arbitrary functions (including complex functions with loops). We have to use the routine `aderr_grad` as the following example shows (`test/newton_grad.f90`)

```
1  program newton_grad
2
3    use ISO_FORTRAN_ENV, Only : error_unit, output_unit
4    use numtypes
5    use aderrors
6
7    implicit none
8
9    type (uwreal)  :: pa, pb, xini, a, b
10   real (kind=DP) :: grad(3)
11   integer        :: id
12
13   pa = (/ 0.8_DP, 0.056_DP /)
14   id = aderr_new_id("Variable a", 100)
15   call pa%set_id(id)
16
17   pb = (/ 1.3_DP, 0.10_DP /)
18   id = aderr_new_id("Variable b", 222)
19   call pb%set_id(id)
20
21   xini = (/0.50_DP, 10.0_DP/)
22   id = aderr_new_id("Initial condition", 3674)
23   call xini%set_id(id)
24
25   call aderr_grad(grad, newton, [pa, pb, xini])
26   write(*,*)'derivative of newton method respect a:    ', grad(1)
27   write(*,*)'derivative of newton method respect b:    ', grad(2)
28   write(*,*)'derivative of newton method respect xini: ', grad(3)
29
30   stop
31 contains
32
33   function newton(x)
34     type (uwreal) :: newton
```

[1]Note that initially we set an absurd error to the initial condition `xnew = 0.50 +/- 10.0`

```
35      type (uwreal), intent (in) :: x(:)
36
37      type (uwreal) :: xant, xnew, val, der
38
39      a    = x(1)
40      b    = x(2)
41      xnew = x(3)
42      do
43         xant = xnew
44         call fnew(xant, val, der)
45         xnew = xant - val/der
46         if (abs(xnew%value()-xant%value()).lt.1.0E-10_DP) exit
47      end do
48
49      newton = xnew
50
51      return
52    end function newton
53
54    subroutine fnew(x, f, d)
55    ! Returns the value (f) an derivative (d) of
56    ! the function f(x)-x
57
58      type (uwreal), intent (in)  :: x
59      type (uwreal), intent (out) :: f, d
60
61      f =  a*cos(b*sin(x)) - x
62      d = -a*b*sin(b*sin(x)) * cos(x) - 1.0_DP
63
64      return
65    end subroutine fnew
66  end program newton_grad
```

The output of the program is

```
derivative of newton method respect a:      0.47350138707933248
derivative of newton method respect b:     -0.19031322284115804
derivative of newton method respect xini:   -2.5243548967072378E-028
```

3.7 Computing the Hessian of arbitrary functions

The `module aderrors` is also able to determine the Hessian of any function to machine precision (even iterative routines). This is useful for error propagation in fits. In this example (`test/hessian.f90`) we just compute the Hessian of the function

$$f(x_1, x_2) = a(x_1 - b + ax_1^3)^2 + b\sin(x_2 - a + b^2) + x_1 x_2\,, \tag{4}$$

where a, b are some parameters with values

$$a = 0.800\,, \qquad b = 1.30\,, \tag{5}$$

and x_1, x_2 will be taken as

$$x_1 \approx -0.4319\ldots\,, \qquad x_2 \approx 4.1610\ldots\,. \tag{6}$$

(these are the values that minimize $f(x_1, x_2)$ for the values of a, b given in equation (5)). Note that the code returns not only the derivatives with respect to x_1, x_2, but also with respect to a, b.

```
program hessian

  use ISO_FORTRAN_ENV, Only : error_unit, output_unit
  use numtypes
  use constants
  use aderrors

  implicit none

  type (uwreal)  :: va(4)
  real (kind=DP) :: hess(4,4)
  integer        :: i, j

  ! X1 AND X2 AT THE MINIMA
  va(1) = (/ -0.43191853747977416_DP, 0.0_DP/)
  va(2) = (/  4.16107180379317750_DP, 0.0_DP/)

  ! CENTRAL VALUES OF (a,b) (ERRORS NOT NEEDED)
  va(3) = (/ 0.8_DP, 0.0_DP/)
  va(4) = (/ 1.3_DP, 0.0_DP/)

  ! COMPUTE THE HESSIAN
  call aderr_hess(hess, fmulti, va)
  write(*,*)'## HESSIAN ##'
  do i = 1, 4
     write(*,'(100ES15.6)')(hess(i,j), j=1, 4)
  end do
  write(*,*)'##         ##'

contains
  function fmulti(x)
    type (uwreal) :: fmulti
    type (uwreal), intent (in) :: x(:)

    fmulti = x(3)*(x(1)-x(4) + x(3)*x(1)**3)**2 + x(4)*sin(x(2) - x(3) + x(4)**2) &
         + x(1)*x(2)

    return
  end function fmulti
end program hessian
```

This little code produces, as output

```
 ## HESSIAN ##
   9.312301E+00   1.000000E+00  -6.996564E+00  -2.316366E+00
   1.000000E+00   1.226151E+00  -1.226151E+00   3.520238E+00
```

```
  -6.996564E+00  -1.226151E+00   1.815519E+00   2.014424E-01
  -2.316366E+00   3.520238E+00   2.014424E-01   1.248029E+01
 ##         ##
```

4 Documentation

4.1 `type (uwreal)`

4.1.1 Components

The data type has no externally accessible components.

4.1.2 Methods that operate on all `uwreal` types

We can operate on any **`uwreal`** data type with the following methods.

`value()`: Function. Returns (**`double precision`**) the estimate of the observable.

`error()`: Function. Returns (**`double precision`**) the error on the estimate of the observable.

`derror()`: Function. Returns (**`double precision`**) the statistical error of the error on the estimate of the observable.

`neid()`: Function. Returns (**`integer`**) the number of ensembles contributing to the error.

`eid(n)`: Function. Returns (**`integer`**) the ensemble ID number **`n`** contributing to the error.

`taui(n)`: Function. Returns (**`double precision`**) τ_{int} corresponding to ensemble number **`n`** contributing to the error.

`dtaui(n)`: Function. Returns (**`double precision`**) $\delta\tau_{\mathrm{int}}$ corresponding to ensemble number **`n`** contributing to the error.

`window(n)`: Function. Returns (**`integer`**) the summation window chosen for ensemble ID number **`n`**.

This example shows the calls to the different accesible components

```
1   type (uwreal) :: obs
2
3   ! Always call the method uwerr()
4   ! to perform error analysis
5   call obs%uwerr()
6   write(*,*)'Observable:   ', obs%value(),  " +/- ", obs%error(), &
7      "error of the error: ", obs%derror()
8   do i = 1, obs%neid()
9   write(*,*)'Ensemble ', obs%eid(i), &
10    ' tau int: ', obs%taui(i), " +/- ", obs%dtaui(i), &
11    " (window: ", obs%window(i), ")"
12  end do
```

`uwerr()`: Subroutine. Performs the error analysis. Only after calling this method we can expect that `error()`, `derror()`, `taui(n)`, `dtaui(n)`, `window(n)` will return the proper values.

`error_src(n)`: Function. Returns the contribution to the sum of errors in quadrature of the n^{th} MC chain contributing to the error.

`error_name(n)`: Function. Returns the name stored in the database that corresponds to the n^{th} MC chain contributing to the error.

This code snippet shows the call to these methods.

```
  ...
  type (uwreal) :: z

  ! Perform error analysis (tails, optimal window,...)
  call z%uwerr()

  write(*,*)'Value:   ', z%value(),  " +/- ", z%error()
  do i = 1, z%neid()
  write(*,'(3X,1A,1X,1A,3X,1F0.2,"%")')&
    'Contribution to error from', z%error_name(i), &
    100.0_DP*z%error_src(i)
  end do
```

`set_stau(x, [id])`: Set the parameter S_τ to the ensemble with ID `id` to `x`. If `id` is not present, the default value (1) is used.

`set_dsig(x, [id])`: Set where to attach the tail for analysis with $\tau_{\exp} \neq 0$ to x. This is measured in units of error in the normalized autocorrelation function. (i.e. `x`=2.0 attach the tail at a point where the value of $\rho(t)$ is two times larger than the error).

`write_bdio(fb, uid)`: Subroutine. Saves the observable in a BDIO record with user identification `uid`. `fb` must be of type `BDIO` and in write state.

`read_bdio(fb)`: Subroutine. Reads the observable from a BDIO record. `fb` must be of type `BDIO`, in read state, and positioned in the record that one want's to read.

`print_hist([ifn])`: Subroutine. Prints the history of the observable as well as the normalized autocorrelation functions and integrated autocorrelation times with errors. This is simple text that can be processed to produce plots. Note that for derived observables there is a history, a normalized autocorrelation function and integrated autocorrelation time for each MC ID.

This example shows the call to the `print_hist` method.

```
  type (uwreal) :: z
  integer       :: iflog

  call z%uwerr()

  open(newunit=iflog, file="sal.out")
  call z%print_hist(iflog)
  close(iflog)
```

4.1.3 Methods that operate on primary observables

These methods operate only on primary observables[2].

`set_id(n)`: Set the ID of the MC chain to `n`. Different ID's are considered statistically independent data, while the same ID is treated in a fully correlated way. (Default is -1)

`set_replica(ivrep)`: Set the replica vector to `ivrep(:)`. This is an integer vector with the size of each replica. Example:

```
1    ! Load 1000 measurements
2    obs = data(1:1000)
3
4    ! In three replicas, two of 100 measurements, one
5    ! of 800 measurements
6    call obs%set_replica((/100, 100, 800/))
```

Note that the sum over the replica vector has to match the number of measurements. (Default is that all measurements belong to the same replica).

`set_texp(tau)`: Set $\tau_{\rm exp}$ to `tau`. This is used to add a tail to the autocorrelation function. (Default is no tail)

IMPORTANT: This code assumes that observables measured in the same ensemble have been measured in exactly the same set of configurations. Therefore all primary observables with the same ensemble ID **must have the same** replica vector.

Similarly, $\tau_{\rm exp}$ is a property of the simulation, and therefore **must be the same on all primary observables measured on the same ensemble ID**. Where to attach the tail for each observable is controlled with the routine `set_dsig`.

The replica vector, $\tau_{\rm exp}$ and ensembles ID's are inherited in derived observables from the primary ones. This is the reason why they cannot be called on derived observables.

4.2 Externally accessible functions

4.2.1 `aderr_new_id(name, [id])`

Stores `name` in the database and associate it with ID `id`. If `id` is not present use a new unique `id`.

`name`: Type `character (len=*)` (input). Name associated with ID `id`.

`id`: Type `integer` (input). If present ID of the MC chain to be associated with `name`.

Output: Type integer. Returns the ID associated with `name` (if `id` is present, this is it).

4.2.2 `aderr_id_exists(id)`

Checks if `id` is associated with a name in the database.

`id`: Type `integer` (input). id to check if exists in the database.

Output: Type `logical`. `T` if the ID is associated with a name in the database. `F` otherwise.

[2] A primary observable is an observable that depend on just one MC id. These are usually the measurements

4.3 Externally accessible subroutines

4.3.1 `adset_default_stau(x)`

Set the default value for the parameter S_τ to `x` (double precision). This value affects the default settings for all primary and derived observables. (Default value is $S_\tau = 4$).

4.3.2 `adset_default_dsig(x)`

Set the default value for the parameter that decides where to attach the tail to the autocorrelation function to `x` (double precision). This value affects the default settings for all primary and derived observables with $\tau_{\rm exp} \neq 0$. (Default value is 1.5).

4.3.3 `aderr_list_ids([ifp])`

Prints the ID and the name of the ID stored in the database in file unit `ifp` if present, to standard output otherwise.

`ifp`: Type `integer` (input).

4.3.4 `addobs(x, a, fn)`

This routine propagates the error from the variables `a` trough the function `fn(a)` into `x`. This has to be understood as in

$$\texttt{x} = \texttt{fn}(\texttt{a}) \tag{7}$$

The routine comes in two flavors: single variable or multiple variables. In the single variable version `x` is a scalar and `fn` is a function with the following interface

```
interface
  function fn(a)
    type (uwreal) :: fn
    type (uwreal), intent (in) :: a(:)
  end function fn
end interface
```

The multiple variable version `x(:)` is a one dimensional array and `fn` has the following interface

```
interface
  subroutine fn(r,a)
    USE numtypes
    import :: uwreal
    type (uwreal), intent (inout) :: r(:)
    type (uwreal), intent (in)    :: a(:)
  end subroutine fn
end interface
```

where `r(:)` has to be understood as the result of the function.

`x/x(:)`: Type `uwreal` (input-output). The result of the error propagation.

`a(:)`: Type `uwreal` (input). The input variables.

`fn`: The routine where errors need to be propagated.

4.3.5 `uwdobs(x, a, fn)`

This routine is completely analogous to `addobs`, except that the differentiation is performed by finite differences (**not** with AD techniques). This is mainly legacy code or for situations where one has a black-box function that does not admit type `uwreal` as input/output and errors need to be propagated.

This routine propagates the error from the variables `a` trough the function `fn(a)` into `x`. This has to be understood as in

$$\mathtt{x} = \mathtt{fn}(\mathtt{a}) \tag{8}$$

The routine comes in two flavors: single variable or multiple variables. In the single variable version `x` is a scalar and `fn` is a function with the following interface

```
interface
  function fn(a)
    real (kind=DP) :: fn
    real (kind=DP), intent (in) :: a(:)
  end function fn
end interface
```

The multiple variable version `x(:)` is a one dimensional array and `fn` has the following interface

```
interface
  subroutine fn(r,a)
    USE numtypes
    import :: uwreal
    real (kind=DP), intent (inout) :: r(:)
    real (kind=DP), intent (in)    :: a(:)
  end subroutine fn
end interface
```

where `r(:)` has to be understood as the result of the function.

`x/x(:)`: Type `real (kind=DP)` (input-output). The result of the error propagation.

`a(:)`: Type `real (kind=DP)` (input). The input variables.

`fn`: The routine where errors need to be propagated.

4.3.6 `aderr_grad(grad, f, x)`

Determines the gradient of the function `f` at the point `x`. The routine comes in two flavors: single variable or multiple variables. In the single variable version `grad(:)` is a double precision vector and `f` is a function with the following interface

```
interface
   function f(x)
     type (uwreal), intent (in) :: x(:)
     type (uwreal) :: f
   end function f
end interface
```

while in the multiple variable version `grad(:,:)` is a douple precision two dimensional array and `f` has the interface

```
interface
   subroutine f(r, x)
     type (uwreal), intent (in)    :: x(:)
     type (uwreal), intent (inout) :: r(:)
   end subroutine f
end interface
```

`grad(:)/grad(:,:):` Type `real (kind=DP)` (input-output). The gradient of the function. In the single variable version the notation is obvious

$$\texttt{grad(j)} = \frac{\partial f}{\partial x_j}\,, \tag{9}$$

while in the multiple variable version we have

$$\texttt{grad(i,j)} = \frac{\partial f_i}{\partial x_j}\,. \tag{10}$$

`f:` The routine whose gradient we want to compute.

`x(:):` Type `uwreal` (input). The point at which the derivative has to be computed.

4.3.7 `aderr_hess(hess, f, x)`

Determines the Hessian of the function `f` at the point `x`. The routine comes in two flavors: single variable or multiple variables. In the single variable version `hess(:,:)` is a double precision matrix and `f` is a function with the following interface

```
interface
   function f(x)
     type (uwreal), intent (in) :: x(:)
     type (uwreal) :: f
   end function f
end interface
```

while in the multiple variable version `hess(:,:,:)` is a douple precision three dimensional array and `f` has the interface

```
interface
   subroutine f(r, x)
     type (uwreal), intent (in)    :: x(:)
     type (uwreal), intent (inout) :: r(:)
   end subroutine f
end interface
```

`hess(:,:)/hess(:,:,:):` Type `real (kind=DP)` (input-output). The hessian of the function. In the single variable version we have

$$\texttt{hess(i,j)} = \frac{\partial f}{\partial x_i \partial x_j}\,. \tag{11}$$

while in the multiple variable version this has to be understood as in

$$\texttt{hess}(\texttt{k},\texttt{i},\texttt{j}) = \frac{\partial f_k}{\partial x_i \partial x_j}\,. \tag{12}$$

`f:` The routine whose gradient we want to compute.

`x(:):` Type `uwreal` (input). The point at which the derivative has to be computed.

4.3.8 `aderr_cov(mat,p)`

Determines the convariance matrix among the observables `p(:)`. The general expression for the covariance matrix is

`mat(:,:):` Type `real (kind=DP)` (input-output). After calling the routine, the covariance ammong the observables `p(:)`.

`p(:):` Type `uwreal` (input). A vector of observables whose covariance is to be determined.

5 Additional modules

In the distribution (`src/misc`) there are a few additional modules for managing the precision of real and complex data types, the definition of common mathematical constants, time and date manipulation, manipulation of Fourier series and FFT (thanks to the FFT package [2]), non-numerical routines (manipulation of command-line arguments and data checksum), and `BDIO` i/o [4].

These modules are not needed for error analysis and are only needed internally. They are provided without documentation.

5.1 `module numtypes`

Includes the definition of simple precision (`SP`), and double precision (`DP`).

5.2 `module constants`

Includes the definition of the simple and double precision mathematical constants of table 1.

5.3 `module time`

Date and time manipulation.

5.4 `module fourier`

Manipulation of Fourier series and FFT (thanks to the FFT package [2]).

5.5 `module nonnumeric`

This module includes command line argument manipulation, data checksum and sorting/searching routines.

5.6 `module BDIO`

This module provides a `fortran` library for BDIO i/o [4].

SP Name	DP Name	Value
PI_SP	PI_DP	π
TWOPI_SP	TWOPI_DP	2π
HALFPI_SP	HALFPI_DP	$\pi/2$
UNITIMAG_SPC	UNITIMAG_DPC	$\imath$
PI_IMAG_SPC	PI_IMAG_DPC	$\pi\imath$
TWOPI_IMAG_SPC	TWOPI_IMAG_DPC	$2\pi\imath$
HALFPI_IMAG_SPC	HALFPI_IMAG_SDC	$\imath\,\pi/2$
SR2_SP	SR2_DP	$\sqrt{2}$
SR3_SP	SR3_DP	$\sqrt{3}$
SRe_SP	SRe_DP	$\sqrt{e}$
SRpi_SP	SRpi_DP	$\sqrt{\pi}$
LG102_SP	LG102_DP	$\log_{10} 2$
LG103_SP	LG103_DP	$\log_{10} 3$
LG10e_SP	LG10e_DP	$\log_{10} e$
LG10pi_SP	LG10pi_DP	$\log_{10} \pi$
LGe2_SP	LGe2_DP	$\log_e 2$
LGe3_SP	LGe3_DP	$\log_e 3$
LGe10_SP	LGe10_DP	$\log_e 10$
GEULER_SP	GEULER_DP	$\gamma(= 0.5772\ldots)$

Table 1: Mathematical constants defined in the `module constants`.

6 FAQ

6.1 Why `fortran`?

The main reason is **portability**. I am not sure how many cores, what OS or architecture we will use in 20 years, but I am sure that this code will work.

6.2 Why automatic differentiation?

The most subtle issue in the practical implementation of the theory of linear error propagation is the computation of derivatives. Numerical diferentiation is an ill defined problem.

Automatic differentiation is exact up to machine precision, and way easier to implement (and faster) than symbolic differentiation. In the opinion of the author is **the solution** to the problem.

6.3 Is there any serious limitation of the code?

The only serious limitation that I can imagine is to handle gaps in MC histories (i.e. different observables of the same MC chain determined with different statistics).

Of course this is not a limitation in principle, and can be solved with a limited effort.

References

[1] Ramos, Alberto, "`aderrors`: Error analysis of monte carlo data with automatic differentiation," 2018. https://gitlab.ift.uam-csic.es/alberto/aderrors.

[2] Ooura, Takuya, "FFT package," 1996.
http://www.kurims.kyoto-u.ac.jp/~ooura/fft.html.

[3] Wikipedia contributors, "Automatic differentiation — Wikipedia, the free encyclopedia," 2018.
https://en.wikipedia.org/wiki/Automatic_differentiation. [Online; accessed 22-June-2018].

[4] Tomasz Korzec and Hubert Simma, "binary data i/o," 2014.
http://www.bdio.org/.