# `Fazang`: A Reverse-mode Automatic differentiation tool in Fortran

**User's Guide**
**(Version 0.0.1)**

Yi Zhang

# Contents

CHAPTER 1

# Introduction

`Fazang` is a reverse-mode automatic differentiation (AD) tool. The project is heavily influenced by `Stan/Math` [**1**], a project the author is also involved in. `Fazang` is intended to support general scientific computing in Fortran beyond Bayesian inference and Markove Chain Monte Carlo that `Stan/Math` is designed for.

User should be aware that the project is at early stage and still under development. For any questions, suggestions, and contributions, please visit the project at `https://github.com/yizhang-yiz/fazang`.

CHAPTER 2

# Quick Start

Currently **Fazang** has been tested on Linux and MacOS platform, with Fortran compiler Intel Fortran 19.0.1+ and GNU Fortran 11.2.0+.

After downloading **Fazang**, user can use **meson** to build the library.

```
git clone git@github.com:yizhang-yiz/fazang.git
cd fazang && mkdir build && cd build
meson compile
```

This generates a shared library at `build/src/`. User needs to link this library when building an application. This can be done in **meson** by setting

```
executable('app_name', files('path/to/app_file.F90'),
↪  dependencies : fazang_dep)
```

**Fazang** provides a user-facing derived type `var`. This is the type for the dependent and independent variables of which the adjoint (derivative) will be calculated.

For example, consider the log of the Gaussian distribution density with mean $\mu$ and standard deviation $\sigma$

$$f(\mu, \sigma) = \log\left(\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2\right)\right) \tag{1}$$

The following programe calculates $\frac{df}{d\mu}$ and $\frac{df}{d\sigma}$ at $y = 1.3$, $\mu = 0.5$, and $\sigma = 1.2$.

```
program log_demo
  use fazang ! load Fazang library

  implicit none

  real(rk) :: y
```

```fortran
  type(var) :: f, sigma, mu

  ! data
  y = 1.3d0

  ! independent variables
  mu = var(0.5d0)
  sigma = var(1.2d0)

  ! dependent
  f = var(-0.5d0 * log(2 * pi))
  f = f - log(sigma)
  f = f - 0.5d0 * ((y - mu) / sigma) ** 2.d0;

  ! use grad() to calculate df/d(mu) and df/d(sigma). Each var's
  ! derivative (also called adjoint) can be access through
  ↪  var%adj().

  call f%grad()
  write(*, *) "df/d(mu): ", mu%adj()
  write(*, *) "df/d(sigma): ", sigma%adj()
end program log_demo
```

CHAPTER 3

# Use Fazang

`Fazang` uses `var` type to record numerical and gradient operations. The type supports three functions

- `var%val()` : returns value
- `var%adj()` : returns derivative, henceforth referred as *adjoint*.
- `var%grad()` : takes gradient operation with respect to the current `var` variable.

## 1. Constructors

`var` can be constructed using overloaded `var` interface.

```
real(real64) :: a, b(3), c(2, 3)
real(real64) :: new_a, new_b(3), new_c(2, 3)
type(var) :: x, y(3), z(2, 3)
! ...
x = var()                    ! x%val() == 0.d0
x = var(a)                   ! x%val() == a
y = var(b)                   ! y%val() == b
z = var(c)                   ! z%val() == c
```

## 2. Assignment

`var` can be assigned from consistent `var` and `real(real64)`.

```
! ....
x = new_a                    ! x%val() == new_a
y = new_b                    ! y%val() == new_b
z = new_c                    ! z%val() == new_c
```

## 3. Gradient

Consider a variable $z$ calculated by the composition of a series of operations

$$z = f_1(z_1), \quad z_1 = f_2(z_2), \quad \dots, \quad z_{n-1} = f_n(z_n),$$

for $z_i, i = 1, \ldots, n$ we call $dz/z_i$ the *adjoint* of $z_i$, denoted by $z_i^{\text{adj}}$. According to chain rule the adjoints can be calculated recursively [**2**],

$$\begin{aligned} z^{\text{adj}} &= 1, \\ z_1^{\text{adj}} &= z^{\text{adj}} \frac{f_1}{z_1}, \\ &\ldots, \\ z_i^{\text{adj}} &= z_{i-1}^{\text{adj}} \frac{f_i}{z_i}. \end{aligned}$$

We often refer each $(f_i, z_i)$ pair as a *node*, and $z_i$ the *operand* of operation $f_i$. The above recursion through the nodes requires a way to store and visit the *callstack* of nodes. It is embodied in `Fazang` by the `var%grad()` function. When `z%grad()` is called, `z`'s adjoint is set to 1, and every other `var` variable is transversed with its adjoint updated. In order to calculate the adjoint with respect to another variable, user must `call set_zero_all_adj()` first to reset all adjoints to zero.

An alternative to invoke gradient calculation is to define the dependent as a function and feed it to `Fazang`'s `gradient` function. Take Eq.(1) for example, we can first define the function for $f(\mu, \sigma)$.

```
module func
  use fazang ! load Fazang library
  implicit none

  real(rk), parameter :: y = 1.3d0

contains
  type(var) function f(x)
    type(var), intent(in) :: x(:)
    type(var) :: mu, sigma
    mu = x(1)
    sigma = x(2)
    f = -0.5d0 * log(2 * pi) - log(sigma) - 0.5d0 * ((y - mu) /
     ↪  sigma) ** 2.d0;
  end function f

end module func
```

Then we can supply function `f` as a procedure argument.

```
program log_demo2
  use iso_c_binding
```

```
  use fazang
  use func

  implicit none

  real(real64) :: fx(3), x(2)
  x = [0.5d0, 1.2d0]

  fx = gradient(f, x)
  write(*, *) "f(x): ", fx(1)
  write(*, *) "df/d(x(1)): ", fx(2)
  write(*, *) "df/d(x(2)): ", fx(3)
end program log_demo2
```

The output of `gradient(f, x)` is an array of size `1 + size(x)`, with first component being the function value, and the rest the partial derivatives.

## 4. Functions

Numeric functions supported by `Fazang` are listed in Appendix A. All unary and binary functions are `elemental`. The binary functions allow mixed argument types, namely, either argument can be `real64` type while the other the `var` type.

CHAPTER 4

# Design

The core of any reverse-mode automatic differentiation is the data structure to store and visit the callstack. `Fazang` achieves this through two derived types, `tape` and `vari`.

## 1. `tape` data structure

A `tape` is an `int32` array emulating a stack, with an integer marker `head` pointing to the head to the current stack top.

```
type :: tape
     integer(ik) :: head = 1
     integer(ik), allocatable :: storage(:)
!...
```

Each time a new AD node is created, space in `storage` is allotted to store the node's

- value $f_i(z_i)$,
- adjoint $z_{i-1}^{\text{adj}}$,
- number of `var` operands of $f_i$,
- The `var` operands' index in the same `tape` array,
- number of `real64` operands of $f_i$,
- The `real64` operands' value.

Since a node's value, adjoint, and data operands are `real64`, they are first converted to `int32` using `transfer` function before stored in the `tape` array, so that each such a value occupies two `storage` entries. After each allotation, the `head` is moved to point to the next empty slot in the array after saving its current value to a `vari` type variable for future retrieval.

## 2. `vari` type

The `vari` type is simply a proxy of a node's storage location in the tape

```
type :: vari
  integer(ik) :: i = 0
  procedure(chain_op), pass, pointer :: chain
contains
   !....
```

where `i` is the index to the beginning of a node's storage, and the `chain` procedure encodes the node's operation $f_i$. `chain` follows an interface that describes the chain rule operation

```
abstract interface
   subroutine chain_op(this)
     import :: vari
     class(vari), intent(in) :: this
   end subroutine chain_op
end interface
```

An alternative to integer index is to a `pointer` to the according enry in the `tape` array. However, we will need to expand the `storage` when it is filled up, and `Fazang` does this by doubling the `storage` size and use `move_alloc` to restore the original values. Since there is no guarantee that `move_alloc` will keep the original memory, a pointer to the original address would be corrupted.

A `Fazang` program steps forward, a series of `vari` variables are generated, with their *values* calculated and stored. This is called a *forward pass*. The generated `vari` variables in the forward pass are stored in array `varis`. Each entry in `varis` is a dependent (operation output) of one or more previous entries.

## 3. var type

The user-facing `var` type serves as proxy to `vari`. Each `var` stores the index of a `vari` in the `varis` array.

```
type :: var
   integer(int32) :: vi
 contains
   procedure :: val
   procedure :: adj
   procedure :: grad
   procedure :: set_chain
end type var
```

After the forward pass, when adjoints are desired, we call `grad` or `gradient` procedure. This initiates a *reverse pass*, in which the `varis` array is transversed backward so that each `vari`'s `chain` procedure is called to update the operand adjoints.

```
subroutine grad(this)
  class(var), intent(in) :: this
  integer i
  call callstack % varis (this%vi) % init_dependent()
  do i = callstack % head - 1, 1, -1
     call callstack % varis(i) % chain()
  end do
end subroutine grad
```

Here `callstack` is the module variable that encapsulate `tape` and `varis` arrays.

CHAPTER 5

# Add operation functions

Adding an operation $f_i$ involves creating functions for forward pass and reverse pass. Let us first use `log` function as a simple example.

First, we create a `log_v` function for the forward pass.

```
impure elemental function log_v(v) result(s)
  type(var), intent(in) :: v
  type(var) :: s
  s = var(log(v%val()), [v])
  call s%set_chain(chain_log)
end function log_v
```

The function generates a new `var` variable `s` using a special constructor `var(value, array of operands)` which stores the value as well as the single operand `v`'s index (in the `tape storage` array). It also points `s`'s chain to a dedicated procedure `chain_log`.

```
subroutine chain_log(this)
  class(vari), intent(in) :: this
  real(rk) :: new_adj(1), val(1)
  new_adj = this%operand_adj()
  val = this%operand_val()
  new_adj(1) = new_adj(1) + this%adj() / val(1)
  call this%set_operand_adj(new_adj)
end subroutine chain_log
```

To understand this function, recall the recursion in Section 3, assume the `log` operation is node $i$, then $f_i = \log(\dot{})$ and $z_i$ is the operand `v`, and the new `var s` would be $z_{i-1}$. During the reverse pass when the node is visited, `chain_log` first retrieves current $(z_i, z_i^{\text{adj}})$ using `operand_val()` and `operand_adj()`, then updates $z_i^{\text{adj}}$ with additional

$$z_{i-1}^{\text{adj}} \frac{df_i}{dz_i} = z_{i-1}^{\text{adj}} \frac{\log(z_i)}{z_i} = \frac{z_{i-1}^{\text{adj}}}{z_i}.$$

Adding a binary operation $f_i(z_i^{(1)}, z_i^2)$ is slightly more complex, as we will need to address possibly different scenarios when $z_i^{(1)}$ and $z_i^{(2)}$ are either `var` or `real64`. Let us use overloaded division `operator(/)` as an example.

With

$$f_i(z_i^{(1)}, z_i^2) = z_i^{(1)}/z_i^{(2)}$$

we need to account for

- both $z_i^{(1)}$ and $z_i^2$ are `var`'s
- $z_i^{(1)}$ is `var`, $z_i^2$ is `real64`,
- $z_i^{(1)}$ is `real64`, $z_i^2$ is `var`,

For the first scenario, we create

```
impure elemental function div_vv(v1, v2) result(s)
  type(var), intent(in) :: v1, v2
  type(var) :: s
  s = var(v1%val() / v2%val(), [v1, v2])
  call s%set_chain(chain_div_vv)
end function div_vv
```

Similar to the `log` example, we create a new `s` with both operands stored. In the corresponding `chain` procedure, we need update the adjoints of both `v1` and `v2`.

```
subroutine chain_div_vv(this)
  class(vari), intent(in) :: this
  real(rk) :: new_adj(2), val(2)
  new_adj = this%operand_adj()
  val = this%operand_val()
  new_adj(1) = new_adj(1) + this%adj()/val(2)
  new_adj(2) = new_adj(2) - this%val() * this%adj()/val(2)
  call this%set_operand_adj(new_adj)
end subroutine chain_div_vv
```

For the second scenario, we create

```
impure elemental function div_vd(v, d) result(s)
  type(var), intent(in) :: v
  real(rk), intent(in) :: d
  type(var) :: s
  s = var(v%val() / d, [v], [d])
  call s%set_chain(chain_div_vd)
end function div_vd
```

Again we create a new `var s`. But this time we use another constructor `var(value, var operands, data operands)` to store value, `var` operand `v`, and `real64` operand `d`. In the corresponding reverse pass `chain` procedure, not only we need retrieve `var` operand `v` but also data operand `d`, as the new adjoint of $z_i^{(1)}$ is

$$z_i^{(1)\text{new adj}} = z_i^{(1)\text{old adj}} + z_{i-1}^{\text{adj}} \frac{df_i}{dz_i^{(1)}} = z_i^{(1)\text{old adj}} + z_{i-1}^{\text{adj}} \frac{1}{dz_i^{(2)}}$$

So with `v` as $z_i^{(1)}$ and `d` as $z_i^{(2)}$ we have

```
subroutine chain_div_vd(this)
  class(vari), intent(in) :: this
  real(rk) d(1), new_adj(1)
  new_adj = this%operand_adj()
  d = this%data_operand()
  new_adj(1) = new_adj(1) + this%adj() / d(1)
  call this%set_operand_adj(new_adj)
end subroutine chain_div_vd
```

The third scenario is treated similarly.

APPENDIX A

# Fazang Functions

| Function | Argument(s) | Operation |
|---|---|---|
| sin | scalar or array | same as intrinsic |
| cos | scalar or array | same as intrinsic |
| tan | scalar or array | same as intrinsic |
| asin | scalar or array | same as intrinsic |
| acos | scalar or array | same as intrinsic |
| atan | scalar or array | same as intrinsic |
| log | scalar or array | same as intrinsic |
| exp | scalar or array | same as intrinsic |
| squrt | scalar or array | same as intrinsic |
| square | scalar or array | For input x, calculate x**2 |
| inv | scalar or array | For input x, calculate 1/x |
| inv_square | scalar or array | For input x, calculate 1/x**2 |
| inv_sqrt | scalar or array | For input x, calculate 1/sqrt(x) |
| operator (+) | scalars or arrays | same as intrinsic |
| operator (-) | scalars or arrays | same as intrinsic |
| operator (*) | scalars or arrays | same as intrinsic |
| operator (/) | scalars or arrays | same as intrinsic |
| operator (**) | scalars | same as intrinsic |
| sum | 1D array | same as intrinsic |
| dot_product | 1D arrays | same as intrinsic |
| log_sum_exp | 1D array | For input x, calculate log(sum(exp((x)))) |
| matmul | 2D arrays | same as intrinsic |

# Bibliography

[1] Bob Carpenter, Matthew D. Hoffman, Marcus A. Brubaker, Daniel Lee, Peter Li, and Michael J. Betancourt. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv 1509.07164.*, 2015. `https://mc-stan.org/users/interfaces/math`.

[2] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition.* Society for Industrial and Applied Mathematic, Philadelphia, PA, second edition edition, September 2008.