Group Meeting
Introduction to numerical optimization and derivatives

Doug Shi-Dong

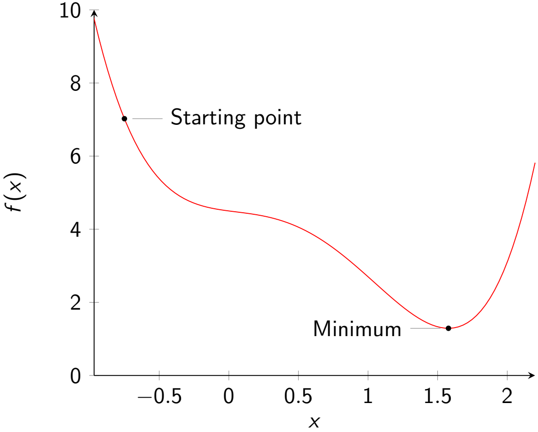
Computational Aerodynamics Group
Department of Mechanical Engineering
McGill University

24 March 2021

Problem statement

- Minimize an objective function $f : \mathbb{R}^n \to \mathbb{R}$

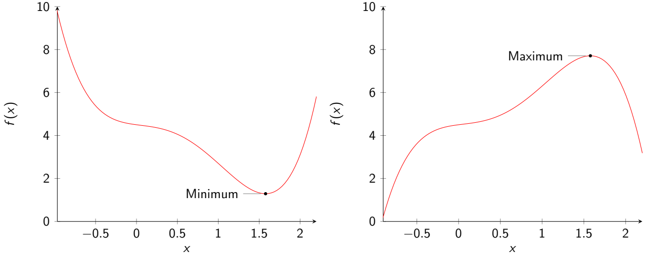
$$\min_{x} f(x)$$



Minimum

- What defines a minimum?
- First-order necessary condition

$$\frac{\mathrm{d}f}{\mathrm{d}x} = 0 \tag{1}$$

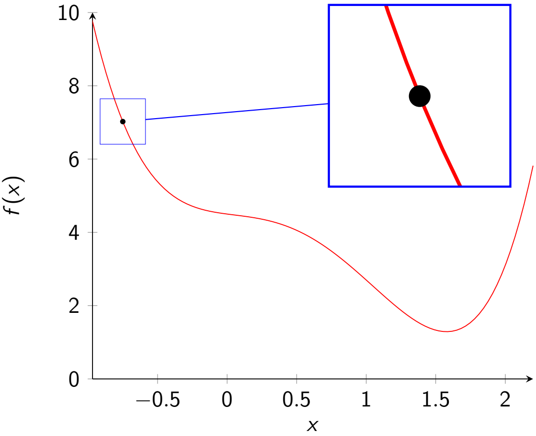


- Second-order sufficient condition

$$v^T \frac{\mathrm{d}^2 f}{\mathrm{d}x^2} v > 0 \tag{2}$$

Direction of descent

- We don't know the objective landscape
- We only know "local" information



Gradient descent

- Evaluate gradient the current design and step in its negative direction

$$\mathsf{x}^{n+1} = \mathsf{x}^{n} - \frac{\mathrm{d}f}{\mathrm{dx}} \tag{3}$$

- Gradient is the local slope, therefore, must step carefully

$$\mathsf{x}^{n+1} = \mathsf{x}^{n} - \eta\frac{\mathrm{d}f}{\mathrm{dx}} \tag{4}$$

  where η is also known as the step length, or learning rate (for ML).

- However, as we approach our minimum $\frac{\mathrm{d}f}{\mathrm{dx}} \to 0$, which means that we take smaller and smaller steps

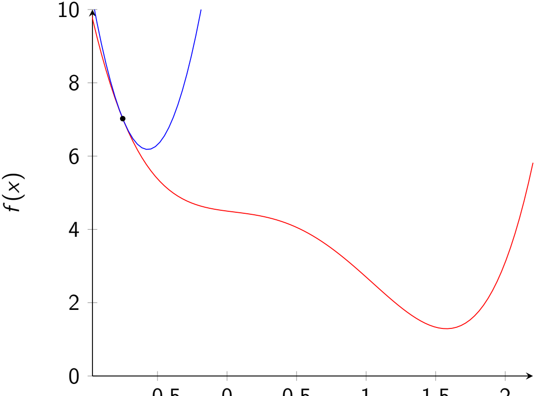
Newton step

- Model f as a quadratic function

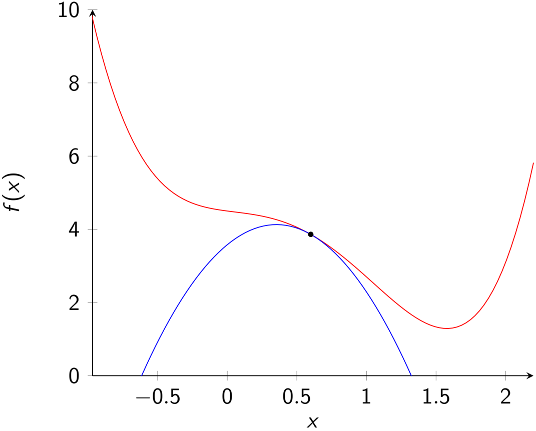
$$f \approx (\Delta x)^T \frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(\Delta x) + \frac{\mathrm{d}f}{\mathrm{d}x}(\Delta x) + f \tag{5}$$

- Easy to find minimum of a quadratic function, "simply" solve for (Δx)

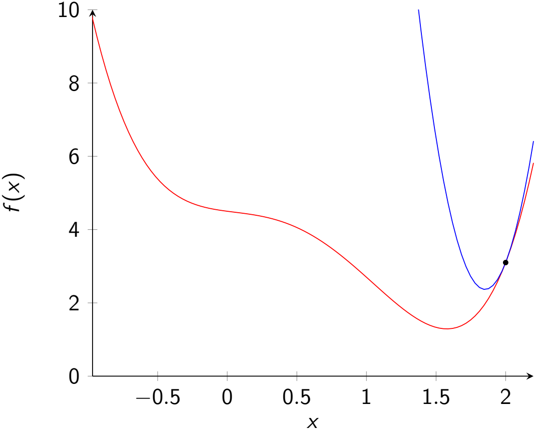
$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(\Delta x) = -\frac{\mathrm{d}f}{\mathrm{d}x} \tag{6}$$



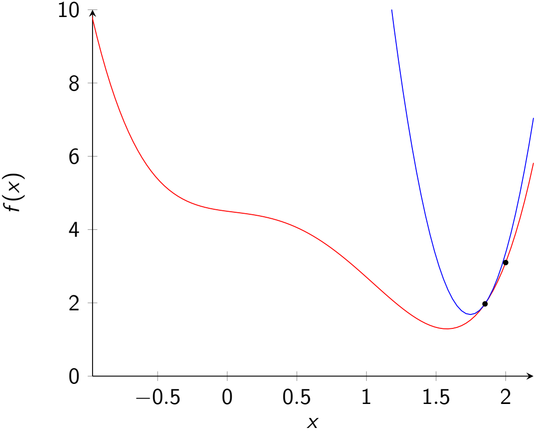
Newton step



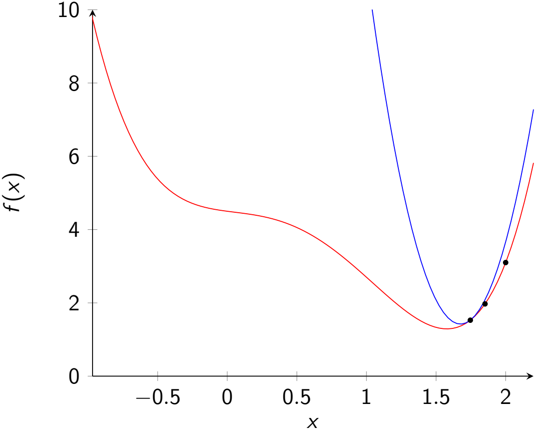
Newton step



Newton step



Newton step



Derivatives

- Compute derivatives various ways
  - Analytical
  - Finite differences
  - Automatic differentiation

	Analytical	Finite-Differences	Automatic Differentiation
Easy to implement	NO	YES	YES
Sustainable code	NO	YES	YES
Exact	YES	NO	YES
Scalable	YES	NO	YES

Analytical derivatives

- Good for small functions or to assemble blocks of derivatives
- Tedious and error-prone

	Analytical
Easy to implement	NO
Sustainable code	NO
Exact	YES
Scalable	YES

Analytical derivatives

```
dde1dtdr1dr1, dde1dtdu1dr1, dde1dtdp1dr1, dde1dtdr2dr1, dde1dtdu2dr1, dde1dtdp2dr1,
dde1dtdr1du1, dde1dtdu1du1, dde1dtdp1du1, dde1dtdr2du1, dde1dtdu2du1, dde1dtdp2du1,
dde1dtdr1dp1, dde1dtdu1dp1, dde1dtdp1dp1, dde1dtdr2dp1, dde1dtdu2dp1, dde1dtdp2dp1,
dde1dtdr1dr2, dde1dtdu1dr2, dde1dtdp1dr2, dde1dtdr2dr2, dde1dtdu2dr2, dde1dtdp2dr2,
dde1dtdr1du2, dde1dtdu1du2, dde1dtdp1du2, dde1dtdr2du2, dde1dtdu2du2, dde1dtdp2du2,
dde1dtdr1dp2, dde1dtdu1dp2, dde1dtdp1dp2, dde1dtdr2dp2, dde1dtdu2dp2, dde1dtdp2dp2;

dde1dtdr1dr1 = (2.0 * Cv * ddp1dtdr1dr1
               + R * (pow(du1dt + u1, 2.0) * ddr1dtdr1dr1
               + 2.0 * (2.0 * (du1dt + u1) * (1 + dr1dtdr1) * du1dtdr1
               + (dr1dt + r1) * pow(du1dtdr1, 2.0)
               + (dr1dt + r1) * (du1dt + u1) * ddu1dtdr1dr1)))
               / (2.0 * R);
dde1dtdu1dr1 = (2.0 * du1dt * R
               + 2.0 * du1dt * R * du1dtdu1
               + 2.0 * R * u1 * du1dtdu1
               + 2.0 * R * (du1dt + u1) * dr1dtdr1 * (1 + du1dtdu1)
               + 2.0 * R * du1dtdr1 * ((du1dt + u1) * dr1dtdu1
               + (dr1dt + r1) * (1 + du1dtdu1))
               + 2.0 * Cv * ddp1dtdr1du1
               + pow(du1dt, 2.0) * R * ddr1dtdr1du1
               + 2.0 * du1dt * R * u1 * ddr1dtdr1du1
               + R * pow(u1, 2.0) * ddr1dtdr1du1
               + 2.0 * dr1dt * du1dt * R * ddu1dtdr1du1
               + 2.0 * du1dt * R * r1 * ddu1dtdr1du1
               + 2.0 * dr1dt * R * u1 * ddu1dtdr1du1
               + 2.0 * R * r1 * u1 * ddu1dtdr1du1)
               / (2.0 * R);
dde1dtdp1dr1 = (2.0 * R * (du1dt + u1) * dr1dtdp1 * du1dtdr1
               + 2.0 * R * du1dtdp1 * (du1dt + u1 + (du1dt + u1) * dr1dtdr1
               + (dr1dt + r1) * du1dtdr1)
               + 2.0 * Cv * ddp1dtdp1dr1
               + pow(du1dt, 2.0) * R * ddr1dtdp1dr1
               + 2.0 * du1dt * R * u1 * ddr1dtdp1dr1
               + R * pow(u1, 2.0) * ddr1dtdp1dr1
               + 2.0 * dr1dt * du1dt * R * ddu1dtdp1dr1
               + 2.0 * du1dt * R * r1 * ddu1dtdp1dr1
               + 2.0 * dr1dt * R * u1 * ddu1dtdp1dr1
               + 2.0 * R * r1 * u1 * ddu1dtdp1dr1)
               / (2.0 * R);
dde1dtdr2dr1 = (2.0 * R * (du1dt + u1) * dr1dtdr2 * du1dtdr1
               + 2.0 * R * (du1dt + u1
               + (du1dt + u1) * dr1dtdr1
               + (dr1dt + r1) * du1dtdr1) * du1dtdr2
               + 2.0 * Cv * ddp1dtdr1dr2
               + pow(du1dt, 2.0) * R * ddr1dtdr1dr2
               + 2.0 * du1dt * R * u1 * ddr1dtdr1dr2
               + R * pow(u1, 2.0) * ddr1dtdr1dr2
               + 2.0 * dr1dt * du1dt * R * ddu1dtdr1dr2
               + 2.0 * du1dt * R * r1 * ddu1dtdr1dr2
               + 2.0 * dr1dt * R * u1 * ddu1dtdr1dr2
               + 2.0 * R * r1 * u1 * ddu1dtdr1dr2)
               / (2.0 * R);
dde1dtdu2dr1 = (2.0 * R * (du1dt + u1) * dr1dtdu2 * du1dtdr1
               + 2.0 * R * (du1dt + u1
               + (du1dt + u1) * dr1dtdr1
               + (dr1dt + r1) * du1dtdr1) * du1dtdu2
               + 2.0 * Cv * ddp1dtdr1du2
               + pow(du1dt, 2.0) * R * ddr1dtdr1du2
               + 2.0 * du1dt * R * u1 * ddr1dtdr1du2
               + R * pow(u1, 2.0) * ddr1dtdr1du2
               + 2.0 * dr1dt * du1dt * R * ddu1dtdr1du2
               + 2.0 * du1dt * R * r1 * ddu1dtdr1du2
               + 2.0 * dr1dt * R * u1 * ddu1dtdr1du2
               + 2.0 * R * r1 * u1 * ddu1dtdr1du2)
               / (2.0 * R);
dde1dtdp2dr1 = (2.0 * R * (du1dt + u1) * dr1dtdp2 * du1dtdr1
               + 2.0 * R * du1dtdp2 * (du1dt + u1 + (du1dt + u1) * dr1dtdr1
               + (dr1dt + r1) * du1dtdr1)
               + 2.0 * Cv * ddp1dtdp2dr1
               + pow(du1dt, 2.0) * R * ddr1dtdp2dr1
               + 2.0 * du1dt * R * u1 * ddr1dtdp2dr1
               + R * pow(u1, 2.0) * ddr1dtdp2dr1
               + 2.0 * dr1dt * du1dt * R * ddu1dtdp2dr1
               + 2.0 * du1dt * R * r1 * ddu1dtdp2dr1
               + 2.0 * dr1dt * R * u1 * ddu1dtdp2dr1
               + 2.0 * R * r1 * u1 * ddu1dtdp2dr1)
               / (2.0 * R);
```

Figure: 100 out of 2800 lines of boundary conditions for quasi-1D Euler

Finite differences derivatives

- Need to choose a good perturbation
- Need as many function evaluations as design variables

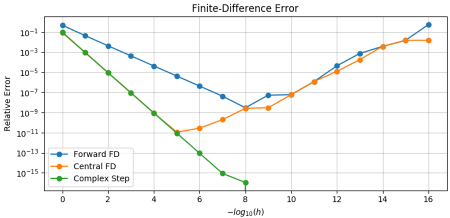


Figure: Finite difference error

	Analytical	Finite-Differences
Easy to implement	NO	YES
Sustainable code	NO	YES
Exact	YES	NO
Scalable	YES	NO

Automatic differentiation derivatives

- It's automatic right?

	Analytical	Finite-Differences	Automatic Differentiation
Easy to implement	NO	YES	YES
Sustainable code	NO	YES	YES
Exact	YES	NO	YES
Scalable	YES	NO	YES

Automatic differentiation derivatives

- It's automatic right? Yes. But no.
- Needs code planification and a bit of software engineering

	Analytical	Finite-Differences	Automatic Differentiation
Easy to implement	NO	YES	YES (but no)
Sustainable code	NO	YES	YES
Exact	YES	NO	YES
Scalable	YES	NO	YES (if done correctly)

C++ templates

```cpp
double function(double input)
{
    double result = std::sin(input) * input;
    return result;
}
int main() {
    double x = 3.0;
    std::cout << function(x) << std::endl;
}
```

C++ templates

```
double function(double input)
{
    double result = std::sin(input) * input;
    return result;
}
int main() {
    double x = 3.0;
    std::cout << function(x) << std::endl;

    std::complex<double> y(3.0, 1.0);
    std::cout << function(y) << std::endl;
}
```

```
intro_to_templates.cpp: In function 'int main()':
intro_to_templates.cpp:10:27: error: cannot convert 'std::complex<double>' to 'double'
   10 |     std::cout << function(y) << std::endl;
      |                           ^
      |                           |
      |                           std::complex<double>
intro_to_templates.cpp:3:24: note:   initializing argument 1 of 'double function(double)'
    3 | double function(double input)
      |                 ~~~~~~~^~~~~
```

Figure: Resulting compilation error

C++ templates

```
template<typename realtype>
realtype function(realtype input)
{
    realtype result = std::sin(input) * input;
    return result;
}
int main() {
    double x = 3.0;
    std::cout << function<double>(x) << std::endl;

    std::complex<double> y(3.0, 1.0);
    std::cout << function<std::complex<double>>(y) << std::endl;
}
```

C++ templates used for operator overloading

```
template<typename realtype>
double function(double input)
{
    double result = std::sin(input) * input;
    return result;
}
std::complex<double> function(std::complex<double> input)
{
    std::complex<double> result = input*input;
    return result;
}
int main() {
    double x = 3.0;
    std::cout << function(x) << std::endl;

    std::complex<double> y(3.0, 1.0);
    std::cout << function(y) << std::endl;
}
```

Automatic differentiation

- Automatic differentiation define their own type such as
  - Sacado::Fad::DFad<double> for Sacado
  - codi::RealForward for CoDiPack.

```
template<typename realtype>
realtype function(realtype input)
{
    realtype result = std::sin(input) * input;
    return result;
}
int main() {
    double x = 3.0;
    std::cout << function<double>(x) << std::endl;

    codi::RealForward y = x; // y = (3.0, 0.0)
    y.setGradient(1.0); // y = (3.0, 1.0)
    // Automatic diff, chain rule
    // df/dinput = dsin(input)/dinput * dinput/dinput * input
    //             + sin(input) * dinput/dinput
    // f = (sin(3.0)*3.0, cos(input)*1.0*3.0 + sin(3.0)*1.0) = (0.42336, -2.8289
    codi::RealForward f = function<codi::RealForward>(y);
    double gradient = f.getGradient();
    std::cout << f << std::endl; // Outputs 0.4234
    std::cout << gradient << std::endl; // Outputs -2.8289
}
```

Automatic differentiation

- In the forward-mode, imagine that the input variable of type `Sacado::Fad::DFad<double>` is represented by a set of two doubles. $x = (3.0, 1.0)$.
- When a function is called on it, for example $z = \sin(x)$, the AD libraries know that it should be storing $z = (\sin(3.0), 1.0 * \cos(3.0))$
- Once all operators, such as (`*`, `/`, `+`, `-`, `pow`, `sqrt`, etc.) have their differentiator counterpart defined by the AD library, it is easy to imagine how to chain those operations.

Automatic differentiation

- Nocedal 2006

$$f(x) = (x_1 x_2 \sin x_3 + e^{x_1 x_2})/x_3 \tag{7}$$

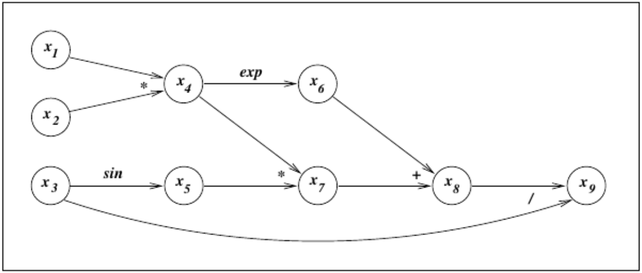


Figure 8.2 Computational graph for $f(x)$ defined in (8.26).

Forward-mode and Reverse-mode

- Forward-mode cost is proportional to the number of **inputs**
- Reverse-mode cost is proportional to the number of **outputs**
- Reverse-mode requires a "tape" that stores operations and runs it "backwards".
- Gradient of a single objective function is a great use of reverse-mode

Hands-on

- Easier to show
- Implementation will depend on AD library being used
- https://github.com/dougshidong/OptimizationTutorial