

Python Algorithmic Differentiation of Large Sparse Linear Inversions

An Introduction to AD and Reverse Mode Checkpointing

Bradley M. Bell

Applied Physics Laboratory
and Institute of Health Metrics
University of Washington

SIAM Conference on Computational Science and
Engineering



Outline

- 1 Introduction to AD
 - A Simple Function
 - Forward Mode
 - Reverse Mode
 - Checkpoint Motivation
- 2 Conjugate Gradient Algorithm
 - Mathematical
 - Python `cg_iterate`
 - Example
- 3 Algorithmic Differentiation
 - Python
 - Results
 - Discussion

Outline

- 1 Introduction to AD
 - A Simple Function
 - Forward Mode
 - Reverse Mode
 - Checkpoint Motivation
- 2 Conjugate Gradient Algorithm
 - Mathematical
 - Python `cg_iterate`
 - Example
- 3 Algorithmic Differentiation
 - Python
 - Results
 - Discussion

Outline

- 1 Introduction to AD
 - A Simple Function
 - Forward Mode
 - Reverse Mode
 - Checkpoint Motivation
- 2 Conjugate Gradient Algorithm
 - Mathematical
 - Python `cg_iterate`
 - Example
- 3 Algorithmic Differentiation
 - Python
 - Results
 - Discussion

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

A Simple Function

$$f(x) = \log[\exp(x_1) + \exp(x_2)]$$

- Input: scalar values x_1^0, x_2^0 .
- $s_1^0 = \exp(x_1^0)$.
- $s_2^0 = s_1^0 + \exp(x_2^0)$.
- $f^0 = \log(s_2^0)$
- Output: scalar value f^0

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Forward Mode

$$f^1 = (df/dx)(x^0)x^1$$

- Input: $x^0 \in \mathbf{R}^2$, $x^1 \in \mathbf{R}^2$, $s^0 \in \mathbf{R}^2$, $f^0 \in \mathbf{R}$.
- $s_1^1 = \exp(x_1^0)x_1^1$.
- $s_2^1 = s_1^1 + \exp(x_2^0)x_2^1 = \exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1$
- $f^1 = s_2^1/s_2^0 = [\exp(x_1^0)x_1^1 + \exp(x_2^0)x_2^1]/s_2^0$
- Output: scalar value f^1

First Order Reverse Mode

$$x^1 = f^1(df/dx)(x^0)$$

- Input: $x^0 \in \mathbf{R}^n$, $s^0 \in \mathbf{R}^n$, $f^0 \in \mathbf{R}$, $f^1 \in \mathbf{R}$.
- $s_2^1 = f^1/s_2^0$
- $x_2^1 = \exp(x_2^0)s_2^1. \quad = f^1 \exp(x_2^0)/s_2^0$
- $s_1^1 = s_2^1 \quad = f^1/s_2^0$
- $x_1^1 = \exp(x_1^0)s_1^1. \quad = f^1 \exp(x_1^0)/s_2^0$
- Output: vector value $x^1 \in \mathbf{R}^n$

First Order Reverse Mode

$$x^1 = f^1(df/dx)(x^0)$$

- Input: $x^0 \in \mathbf{R}^n$, $s^0 \in \mathbf{R}^n$, $f^0 \in \mathbf{R}$, $f^1 \in \mathbf{R}$.
- $s_2^1 = f^1/s_2^0$
- $x_2^1 = \exp(x_2^0)s_2^1. \quad = f^1 \exp(x_2^0)/s_2^0$
- $s_1^1 = s_2^1 \quad = f^1/s_2^0$
- $x_1^1 = \exp(x_1^0)s_1^1. \quad = f^1 \exp(x_1^0)/s_2^0$
- Output: vector value $x^1 \in \mathbf{R}^n$

First Order Reverse Mode

$$x^1 = f^1(df/dx)(x^0)$$

- Input: $x^0 \in \mathbf{R}^n$, $s^0 \in \mathbf{R}^n$, $f^0 \in \mathbf{R}$, $f^1 \in \mathbf{R}$.
- $s_2^1 = f^1/s_2^0$
- $x_2^1 = \exp(x_2^0)s_2^1. \quad = f^1 \exp(x_2^0)/s_2^0$
- $s_1^1 = s_2^1 \quad = f^1/s_2^0$
- $x_1^1 = \exp(x_1^0)s_1^1. \quad = f^1 \exp(x_1^0)/s_2^0$
- Output: vector value $x^1 \in \mathbf{R}^n$

First Order Reverse Mode

$$x^1 = f^1(df/dx)(x^0)$$

- Input: $x^0 \in \mathbf{R}^n$, $s^0 \in \mathbf{R}^n$, $f^0 \in \mathbf{R}$, $f^1 \in \mathbf{R}$.
- $s_2^1 = f^1/s_2^0$
- $x_2^1 = \exp(x_2^0)s_2^1$. $= f^1 \exp(x_2^0)/s_2^0$
- $s_1^1 = s_2^1$ $= f^1/s_2^0$
- $x_1^1 = \exp(x_1^0)s_1^1$. $= f^1 \exp(x_1^0)/s_2^0$
- Output: vector value $x^1 \in \mathbf{R}^n$

First Order Reverse Mode

$$x^1 = f^1(df/dx)(x^0)$$

- Input: $x^0 \in \mathbf{R}^n$, $s^0 \in \mathbf{R}^n$, $f^0 \in \mathbf{R}$, $f^1 \in \mathbf{R}$.
- $s_2^1 = f^1/s_2^0$
- $x_2^1 = \exp(x_2^0)s_2^1$. $= f^1 \exp(x_2^0)/s_2^0$
- $s_1^1 = s_2^1$ $= f^1/s_2^0$
- $x_1^1 = \exp(x_1^0)s_1^1$. $= f^1 \exp(x_1^0)/s_2^0$
- Output: vector value $x^1 \in \mathbf{R}^n$

Checkpoint Motivation

Suppose $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$

- Given $x^0 \in \mathbf{R}^n$, $x^1 \in \mathbf{R}^n$, forward mode computes

$$g^1 = (dg/dx)(x^0)x^1 \in \mathbf{R}^m$$

- Given $x^0 \in \mathbf{R}^n$, $g^1 \in \mathbf{R}^m$, reverse mode computes

$$x^1 = (g^1)^T (dg/dx)(x^0) \in \mathbf{R}^n$$

- Forward mode does not require intermediate values.
- Reverse mode requires intermediate values; e.g., s_n^0 .
- Checkpointing reduces memory during reverse mode at the expense of extra computations.

Checkpoint Motivation

Suppose $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$

- Given $x^0 \in \mathbf{R}^n$, $x^1 \in \mathbf{R}^n$, forward mode computes

$$g^1 = (dg/dx)(x^0)x^1 \in \mathbf{R}^m$$

- Given $x^0 \in \mathbf{R}^n$, $g^1 \in \mathbf{R}^m$, reverse mode computes

$$x^1 = (g^1)^T (dg/dx)(x^0) \in \mathbf{R}^n$$

- Forward mode does not require intermediate values.
- Reverse mode requires intermediate values; e.g., s_n^0 .
- Checkpointing reduces memory during reverse mode at the expense of extra computations.

Checkpoint Motivation

Suppose $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$

- Given $x^0 \in \mathbf{R}^n$, $x^1 \in \mathbf{R}^n$, forward mode computes

$$g^1 = (dg/dx)(x^0)x^1 \in \mathbf{R}^m$$

- Given $x^0 \in \mathbf{R}^n$, $g^1 \in \mathbf{R}^m$, reverse mode computes

$$x^1 = (g^1)^T (dg/dx)(x^0) \in \mathbf{R}^n$$

- Forward mode does not require intermediate values.
- Reverse mode requires intermediate values; e.g., s_n^0 .
- Checkpointing reduces memory during reverse mode at the expense of extra computations.

Checkpoint Motivation

Suppose $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$

- Given $x^0 \in \mathbf{R}^n$, $x^1 \in \mathbf{R}^n$, forward mode computes

$$g^1 = (dg/dx)(x^0)x^1 \in \mathbf{R}^m$$

- Given $x^0 \in \mathbf{R}^n$, $g^1 \in \mathbf{R}^m$, reverse mode computes

$$x^1 = (g^1)^T (dg/dx)(x^0) \in \mathbf{R}^n$$

- Forward mode does not require intermediate values.
- Reverse mode requires intermediate values; e.g., s_n^0 .
- Checkpointing reduces memory during reverse mode at the expense of extra computations.

Checkpoint Motivation

Suppose $g : \mathbf{R}^n \rightarrow \mathbf{R}^m$

- Given $x^0 \in \mathbf{R}^n$, $x^1 \in \mathbf{R}^n$, forward mode computes

$$g^1 = (dg/dx)(x^0)x^1 \in \mathbf{R}^m$$

- Given $x^0 \in \mathbf{R}^n$, $g^1 \in \mathbf{R}^m$, reverse mode computes

$$x^1 = (g^1)^T (dg/dx)(x^0) \in \mathbf{R}^n$$

- Forward mode does not require intermediate values.
- Reverse mode requires intermediate values; e.g., s_n^0 .
- Checkpointing reduces memory during reverse mode at the expense of extra computations.

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Definition

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $\mu_{k+1} = s_k / (d_k^T A d_k)$
 - $x_{k+1} = x_k + \mu_{k+1} d_k$
 - $g_{k+1} = g_k + \mu_{k+1} A d_k$
 - $s_{k+1} = g_{k+1}^T g_{k+1}$.
 - $d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k$
 - $k = k + 1$.
- Output: vector value x_k

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 - $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 - $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 - $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- Input: $A \in \mathbf{R}^{n \times n}$, $b \in \mathbf{R}^n$, $x_0 \in \mathbf{R}^n$, $\varepsilon \in \mathbf{R}_+$.
- $g_0 = Ax_0 - b$, $s_0 = g_0^T g_0$, $d_0 = -g_0$, $k = 0$.
- While $\sqrt{s_k} > \varepsilon$
 $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 $k = k + 1$.
- Output: vector value x_k
- Benefits
 - Checkpointing only stores $\{s_k\}$, $\{g_k\}$, $\{x_k\}$, and $\{d_k\}$, other intermediate values are recalculated.
 - Overloaded AD type only records $H(s, g, x, d)$, other calculations are in **float**.

CG Iteration Function

- `import numpy`
- `def cg_iterate (s, g, d, x, A) :`
- `Ad = A (d)` `# Adk`
- `dAd = numpy.inner(d, Ad)` `# dkTAdk`
- `mu = s / dAd` `# μk+1 = sk / (dkTAdk)`
- `xp = x + mu * d` `# xk+1 = xk + μk+1dk`
- `gp = g + mu * Ad` `# gk+1 = gk + μk+1Adk`
- `sp = numpy.inner(gp, gp)` `# sk+1 = gk+1Tgk+1`
- `dp = - gp + (sp / s) * d` `# dk+1 = -gk+1 + (sk+1/sk)dk`
- `return (sp, gp, dp, xp)`

CG Iteration Function

- `import` numpy
- `def` cg_iterate (s, g, d, x, A) :
- `Ad` = A (d) *#* Ad_k
- `dAd` = numpy.inner(d, Ad) *#* $d_k^T Ad_k$
- `mu` = s / dAd *#* $\mu_{k+1} = s_k / (d_k^T Ad_k)$
- `xp` = x + mu * d *#* $x_{k+1} = x_k + \mu_{k+1} d_k$
- `gp` = g + mu * Ad *#* $g_{k+1} = g_k + \mu_{k+1} Ad_k$
- `sp` = numpy.inner(gp, gp) *#* $s_{k+1} = g_{k+1}^T g_{k+1}$
- `dp` = - gp + (sp / s) * d *#* $d_{k+1} = -g_{k+1} + (s_{k+1} / s_k) d_k$
- `return` (sp, gp, dp, xp)

CG Iteration Function

- `import` numpy
- `def` cg_iterate (s, g, d, x, A) :
- `Ad = A (d)` `# Adk`
- `dAd = numpy.inner(d, Ad)` `# dkTAdk`
- `mu = s / dAd` `# μk+1 = sk / (dkTAdk)`
- `xp = x + mu * d` `# xk+1 = xk + μk+1dk`
- `gp = g + mu * Ad` `# gk+1 = gk + μk+1Adk`
- `sp = numpy.inner(gp, gp)` `# sk+1 = gk+1Tgk+1`
- `dp = - gp + (sp / s) * d` `# dk+1 = -gk+1 + (sk+1/sk)dk`
- `return` (sp, gp, dp, xp)

CG Iteration Function

- `import` numpy
- `def` cg_iterate (s, g, d, x, A) :
- `Ad = A (d)` `# Adk`
- `dAd = numpy.inner(d, Ad)` `# dkTAdk`
- `mu = s / dAd` `# μk+1 = sk / (dkTAdk)`
- `xp = x + mu * d` `# xk+1 = xk + μk+1dk`
- `gp = g + mu * Ad` `# gk+1 = gk + μk+1Adk`
- `sp = numpy.inner(gp, gp)` `# sk+1 = gk+1Tgk+1`
- `dp = - gp + (sp / s) * d` `# dk+1 = -gk+1 + (sk+1/sk)dk`
- `return` (sp, gp, dp, xp)

CG Iteration Function

- `import` numpy
- `def` cg_iterate (s, g, d, x, A) :
- `Ad = A (d)` `#` Ad_k
- `dAd = numpy.inner(d, Ad)` `#` $d_k^T Ad_k$
- `mu = s / dAd` `#` $\mu_{k+1} = s_k / (d_k^T Ad_k)$
- `xp = x + mu * d` `#` $x_{k+1} = x_k + \mu_{k+1} d_k$
- `gp = g + mu * Ad` `#` $g_{k+1} = g_k + \mu_{k+1} Ad_k$
- `sp = numpy.inner(gp, gp)` `#` $s_{k+1} = g_{k+1}^T g_{k+1}$
- `dp = - gp + (sp / s) * d` `#` $d_{k+1} = -g_{k+1} + (s_{k+1} / s_k) d_k$
- `return` (sp, gp, dp, xp)

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Problem

- Define $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Solve $Dx = c$ for $x \in \mathbf{R}^2$.
- The determinant of D is -2 .
 - D is invertible, hence equation above has a solution.
 - D is not positive definite, hence we cannot directly apply conjugate gradient.
- Solve $Ax = b$ where $A = D^T D$ and $b = D^T c$.
- A is positive definite.

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

Matrix Times a Vector

Compute matrix D or D^T times the vector v .

- `def D(v, transpose) :` $\# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
- `u = numpy.array([0. , 0.])` $\# u = 0$
- `for i in range(2) :` $\# i = 0 , 1$
 - `for j in range(2) :` $\# j = 0 , 1$
 - `D_ij = float(2 * i + j + 1)` $\# D_{ij} = 2i + j + 1$
 - `if not transpose :`
 - `u[i] = u[i] + D_ij * v[j]` $\# u_i = \sum_j D_{ij} v_j$
 - `else :`
 - `u[j] = u[j] + D_ij * v[i]` $\# u_j = \sum_i v_i D_{ij}$
- `return u`

cg_iterate Initialization

- `def A(v) :` $\# Av = D^T Dv$
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` $\# c = (1, 1)^T$
- `b = D(c, True)` $\# b = D^T c$
- `k = 0`
- `x = numpy.array([0. , 0.])` $\# x_0 = 0$
- `g = A(x) - b` $\# g_0 = Ax_0 - b$
- `s = numpy.inner(g, g)` $\# s_0 = g_0^T g_0$
- `d = - g` $\# d_0 = -g_0$

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = -g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = -g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = -g` `# $d_0 = -g_0$`

cg_iterate Initialization

- `def A(v) :` `# $Av = D^T Dv$`
- `return D(D(v, False), True)`
- `eps = 1e-7`
- `c = numpy.array([1., 1.])` `# $c = (1, 1)^T$`
- `b = D(c, True)` `# $b = D^T c$`
- `k = 0`
- `x = numpy.array([0. , 0.])` `# $x_0 = 0$`
- `g = A(x) - b` `# $g_0 = Ax_0 - b$`
- `s = numpy.inner(g, g)` `# $s_0 = g_0^T g_0$`
- `d = - g` `# $d_0 = -g_0$`

cg_iterate Validation

- **while** `math.sqrt(s) > eps` : **# while** $\sqrt{s_k} > \varepsilon$
- **#** $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
`s, g, d, x = cg_iterate(s, g, d, x, A)`
- `k = k + 1`
- **# known solution of $Dx = c$**
`v = numpy.array([-1. , 1.])`
- **# check that $\|x - v\|_\infty < 2\varepsilon$**
`assert all(abs(x - v) < 2 * eps)`
- **# check that at most two iterations were used**
`assert k <= 2`

cg_iterate Validation

- **while** `math.sqrt(s) > eps` : `# while $\sqrt{s_k} > \epsilon$`
- `# (sk+1, gk+1, xk+1, dk+1) = H(sk, gk, xk, dk)`
`s, g, d, x = cg_iterate(s, g, d, x, A)`
- `k = k + 1`
- `# known solution of $Dx = c$`
`v = numpy.array([-1. , 1.])`
- `# check that $\|x - v\|_\infty < 2\epsilon$`
`assert all(abs(x - v) < 2 * eps)`
- `# check that at most two iterations were used`
`assert k <= 2`

cg_iterate Validation

- **while** `math.sqrt(s) > eps` : `# while $\sqrt{s_k} > \epsilon$`
- `# (sk+1, gk+1, xk+1, dk+1) = H(sk, gk, xk, dk)`
`s, g, d, x = cg_iterate(s, g, d, x, A)`
- `k = k + 1`
- `# known solution of $Dx = c$`
`v = numpy.array([-1. , 1.])`
- `# check that $\|x - v\|_\infty < 2\epsilon$`
`assert all(abs(x - v) < 2 * eps)`
- `# check that at most two iterations were used`
`assert k <= 2`

cg_iterate Validation

- **while** `math.sqrt(s) > eps` : `# while $\sqrt{s_k} > \epsilon$`
- `# (sk+1, gk+1, xk+1, dk+1) = H(sk, gk, xk, dk)`
`s, g, d, x = cg_iterate(s, g, d, x, A)`
- `k = k + 1`
- `# known solution of Dx = c`
`v = numpy.array([-1. , 1.])`
- `# check that $\|x - v\|_\infty < 2\epsilon$`
`assert all(abs(x - v) < 2 * eps)`
- `# check that at most two iterations were used`
`assert k <= 2`

cg_iterate Validation

- **while** `math.sqrt(s) > eps` : `# while $\sqrt{s_k} > \epsilon$`
- `# (sk+1, gk+1, xk+1, dk+1) = H(sk, gk, xk, dk)`
`s, g, d, x = cg_iterate(s, g, d, x, A)`
- `k = k + 1`
- `# known solution of $Dx = c$`
`v = numpy.array([-1. , 1.])`
- `# check that $\|x - v\|_\infty < 2\epsilon$`
`assert all(abs(x - v) < 2 * eps)`
- `# check that at most two iterations were used`
`assert k <= 2`

cg_iterate Validation

- **while** math.sqrt(s) > eps : **# while** $\sqrt{s_k} > \varepsilon$
- **#** $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) = H(s_k, g_k, x_k, d_k)$
 s, g, d, x = cg_iterate(s, g, d, x, A)
- k = k + 1
- **# known solution of $Dx = c$**
 v = numpy.array([-1. , 1.])
- **# check that $\|x - v\|_\infty < 2\varepsilon$**
assert all(abs(x - v) < 2 * eps)
- **# check that at most two iterations were used**
assert k <= 2

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
H = pycppad.adfun(a_y, a_z)

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Include Dependence of A in Recording of $z = H(y)$

- Let u be vector of non-zeros in representation of D .
- $y = \text{pack}(s, g, d, x, u)$
- $a_y = \text{pycppad.independent}(y)$
- Extract a_u from a_y and use it to define $a_A(a_v)$.
- $a_s, a_g, a_d, a_x = \text{unpack_sgdx}(n, a_y)$
- $a_s, a_g, a_d, a_x = \text{cg_iterate}(a_s, a_g, a_d, a_x, a_A)$
- $a_z = \text{pack}(a_s, a_g, a_d, a_x)$
- # Define $H : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
 $H = \text{pycppad.adfun}(a_y, a_z)$

Iterate Forward Storing Checkpoints

- # Initial checkpoint is empty and z is (s_0, g_0, d_0, x_0, u) .
 checkpoint = []
 z = pack(s, g, d, x)
- while math.sqrt(z[0]) > eps : # $z[0]$ is s_k
 # Store forward input, except u which is constant.
 checkpoint.append(z)
- # Forward mode takes a single vector argument
 y = pack(z, u)
- # Use H to compute $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})$
 z = H.forward(0, y)

Iterate Forward Storing Checkpoints

- # Initial checkpoint is empty and z is (s_0, g_0, d_0, x_0, u) .
 checkpoint = []
 z = pack(s, g, d, x)
- while math.sqrt(z[0]) > eps : # z[0] is s_k
 # Store forward input, except u which is constant.
 checkpoint.append(z)
- # Forward mode takes a single vector argument
 y = pack(z, u)
- # Use H to compute $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})$
 z = H.forward(0, y)

Iterate Forward Storing Checkpoints

- **# Initial checkpoint is empty and z is (s_0, g_0, d_0, x_0, u) .**
 checkpoint = []
 z = pack(s, g, d, x)
- **while** math.sqrt(z[0]) > eps : **# z[0] is s_k**
Store forward input, except u which is constant.
 checkpoint.append(z)
- **# Forward mode takes a single vector argument**
 y = pack(z, u)
- **# Use H to compute $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})$**
 z = H.forward(0, y)

Iterate Forward Storing Checkpoints

- # Initial checkpoint is empty and z is (s_0, g_0, d_0, x_0, u) .
 checkpoint = []
 z = pack(s, g, d, x)
- while math.sqrt(z[0]) > eps : # z[0] is s_k
 # Store forward input, except u which is constant.
 checkpoint.append(z)
- # Forward mode takes a single vector argument
 y = pack(z, u)
- # Use H to compute $(s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})$
 z = H.forward(0, y)

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- `for z in reversed(checkpoint) :` $\# z_k = (s_k, g_k, d_k, x_k)$
- `y = pack(z, u)` $\# y_k = (s_k, g_k, d_k, x_k, u)$
- `z = H.forward(0, y)` $\# z_{k+1}$ (not used)
- `d_y = H.reverse(1, d_z)` $\# (z_N^1)^T \partial F / \partial y_k$
- `d_u = d_u + d_y[len(z) :]` $\#$ accumulate $(z_N^1)^T \partial F / \partial u$
- `d_z = d_y[: len(z)]` $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- `for z in reversed(checkpoint) :` $\# z_k = (s_k, g_k, d_k, x_k)$
- `y = pack(z, u)` $\# y_k = (s_k, g_k, d_k, x_k, u)$
- `z = H.forward(0, y)` $\# z_{k+1}$ (not used)
- `d_y = H.reverse(1, d_z)` $\# (z_N^1)^T \partial F / \partial y_k$
- `d_u = d_u + d_y[len(z) :]` $\# \text{accumulate } (z_N^1)^T \partial F / \partial u$
- `d_z = d_y[: len(z)]` $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- y = pack(z, u) $\# y_k = (s_k, g_k, d_k, x_k, u)$
- z = H.forward(0, y) $\# z_{k+1}$ (not used)
- d_y = H.reverse(1, d_z) $\# (z_N^1)^T \partial F / \partial y_k$
- d_u = $d_u + d_y[\text{len}(z) :]$ $\#$ accumulate $(z_N^1)^T \partial F / \partial u$
- d_z = $d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- y = pack(z, u) $\# y_k = (s_k, g_k, d_k, x_k, u)$
- z = H.forward(0, y) $\# z_{k+1}$ (not used)
- d_y = H.reverse(1, d_z) $\# (z_N^1)^T \partial F / \partial y_k$
- d_u = $d_u + d_y[\text{len}(z) :]$ $\#$ accumulate $(z_N^1)^T \partial F / \partial u$
- d_z = $d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- y = pack(z, u) $\# y_k = (s_k, g_k, d_k, x_k, u)$
- z = H.forward(0, y) $\# z_{k+1}$ (not used)
- d_y = H.reverse(1, d_z) $\# (z_N^1)^T \partial F / \partial y_k$
- d_u = $d_u + d_y[\text{len}(z) :]$ $\#$ accumulate $(z_N^1)^T \partial F / \partial u$
- d_z = $d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- y = pack(z, u) $\# y_k = (s_k, g_k, d_k, x_k, u)$
- z = H.forward(0, y) $\# z_{k+1}$ (not used)
- d_y = H.reverse(1, d_z) $\# (z_N^1)^T \partial F / \partial y_k$
- d_u = $d_u + d_y[\text{len}(z) :]$ $\#$ accumulate $(z_N^1)^T \partial F / \partial u$
- d_z = $d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- $y = \text{pack}(z, u)$ $\# y_k = (s_k, g_k, d_k, x_k, u)$
- $z = H.\text{forward}(0, y)$ $\# z_{k+1}$ (not used)
- $d_y = H.\text{reverse}(1, d_z)$ $\# (z_N^1)^T \partial F / \partial y_k$
- $d_u = d_u + d_y[\text{len}(z) :]$ $\# \text{accumulate } (z_N^1)^T \partial F / \partial u$
- $d_z = d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Reverse Mode with Checkpointing

Let N be last iteration, define $F(y)$ by

$$y_k = (s_k, g_k, d_k, x_k, u), \quad z_k = (s_k, g_k, d_k, x_k), \quad F(y_0) = z_N$$

Given z_N^1 compute $y_0^1 = (z_N^1)^T (\partial F / \partial u)(y_0)$

- Initialize $d_u = 0$ and $d_z = z_N^1$.
- **for** z **in reversed**(checkpoint) : $\# z_k = (s_k, g_k, d_k, x_k)$
- $y = \text{pack}(z, u)$ $\# y_k = (s_k, g_k, d_k, x_k, u)$
- $z = H.\text{forward}(0, y)$ $\# z_{k+1}$ (not used)
- $d_y = H.\text{reverse}(1, d_z)$ $\# (z_N^1)^T \partial F / \partial y_k$
- $d_u = d_u + d_y[\text{len}(z) :]$ $\# \text{accumulate } (z_N^1)^T \partial F / \partial u$
- $d_z = d_y[: \text{len}(z)]$ $\# (z_N^1)^T \partial F / \partial z_k$

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Results

- $D \in \mathbf{R}^{n \times n}$ is diagonally dominant with 2% non-zero.
- N is number of iterations for convergence.
- c is right hand side.
- r is the residual $Dx_N - c$.
- t is number of seconds to compute derivative w.r.t D .

• n	N	$ c $	$ r $	t
• 500	46	12.86	2.41e-08	0.58
• 1000	29	17.94	1.04e-08	2.11
• 1500	25	22.48	2.82e-09	4.70
• 2000	23	25.55	1.85e-09	8.61

Discussion

- The required memory for reverse mode is $N(1 + 3n)$.
- We could recalculate g_k and only require $N(1 + 2n)$.
- These ideas apply to many other iterative methods.
- pycppad obtains C++ Speed using boost-python and cppad.
- The entire calculation could be done in C++ using cppad.

Discussion

- The required memory for reverse mode is $N(1 + 3n)$.
- We could recalculate g_k and only require $N(1 + 2n)$.
- These ideas apply to many other iterative methods.
- pycppad obtains C++ Speed using boost-python and cppad.
- The entire calculation could be done in C++ using cppad.

Discussion

- The required memory for reverse mode is $N(1 + 3n)$.
- We could recalculate g_k and only require $N(1 + 2n)$.
- These ideas apply to many other iterative methods.
- pycppad obtains C++ Speed using boost-python and cppad.
- The entire calculation could be done in C++ using cppad.

Discussion

- The required memory for reverse mode is $N(1 + 3n)$.
- We could recalculate g_k and only require $N(1 + 2n)$.
- These ideas apply to many other iterative methods.
- pycppad obtains C++ Speed using boost-python and cppad.
- The entire calculation could be done in C++ using cppad.

Discussion

- The required memory for reverse mode is $N(1 + 3n)$.
- We could recalculate g_k and only require $N(1 + 2n)$.
- These ideas apply to many other iterative methods.
- pycppad obtains C++ Speed using boost-python and cppad.
- The entire calculation could be done in C++ using cppad.

References

autodiff: Bucker M, et. al.
A Community Portal for Automatic Differentiation

boost-python: Abrahams D., et. al.,
A Library for Interoperability Between C++ and Python

cppad: Bell B.M.
A Package for Differentiation of C++ Algorithms

pycppad: Walter S.F and Bell B.M.,
Python Algorithmic Differentiation Using CppAD

References

autodiff: Bucker M, et. al.
A Community Portal for Automatic Differentiation

boost-python: Abrahams D., et. al.,
A Library for Interoperability Between C++ and Python

cppad: Bell B.M.
A Package for Differentiation of C++ Algorithms

pycppad: Walter S.F and Bell B.M.,
Python Algorithmic Differentiation Using CppAD

References

autodiff: Bucker M, et. al.
A Community Portal for Automatic Differentiation

boost-python: Abrahams D., et. al.,
A Library for Interoperability Between C++ and Python

cppad: Bell B.M.
A Package for Differentiation of C++ Algorithms

pycppad: Walter S.F and Bell B.M.,
Python Algorithmic Differentiation Using CppAD

References

autodiff: Bucker M, et. al.
A Community Portal for Automatic Differentiation

boost-python: Abrahams D., et. al.,
A Library for Interoperability Between C++ and Python

cppad: Bell B.M.
A Package for Differentiation of C++ Algorithms

pycppad: Walter S.F and Bell B.M.,
Python Algorithmic Differentiation Using CppAD