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

International Congress on Industrial & Applied Math, 2011
Outline

1. **Background**
   - Example of Forward and Reverse Mode
   - Checkpointing

2. **Checkpointing Conjugate Gradient Algorithm**
   - Conjugate Gradient Iteration
   - Conjugate Gradient AD
   - Results and Discussion
Outline

1 Background
   - Example of Forward and Reverse Mode
   - Checkpointing

2 Checkpointing Conjugate Gradient Algorithm
   - Conjugate Gradient Iteration
   - Conjugate Gradient AD
   - Results and Discussion
A Simple Defining

\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- Input: scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[ s_j^0 = s_{j-1}^0 + \exp(x_j^0). \]
- \( f^0 = \log(s_n^0) \)
- Output: scalar value \( f^0 \)
A Simple Defining

\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- Input: scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[ s_j^0 = s_{j-1}^0 + \exp(x_j^0) \).
- \( f^0 = \log(s_n^0) \)
- Output: scalar value \( f^0 \)
A Simple Defining

\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- Input: scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[ s_j^0 = s_{j-1}^0 + \exp(x_j^0) \]
- \( f^0 = \log(s_n^0) \)
- Output: scalar value \( f^0 \)
A Simple Defining

\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- Input: scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[ s_j^0 = s_{j-1}^0 + \exp(x_j^0) \].
- \( f^0 = \log(s_n^0) \)
- Output: scalar value \( f^0 \)
A Simple Defining

\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- Input: scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[ s_j^0 = s_{j-1}^0 + \exp(x_j^0) \]
- \( f^0 = \log(s_n^0) \)
- Output: scalar value \( f^0 \)
\[ f(x) = \log[\exp(x_1) + \cdots + \exp(x_n)] \]

- **Input:** scalar values \( x_1^0, \ldots, x_n^0 \).
- \( s_1^0 = \exp(x_1^0) \).
- For \( j = 2, \ldots, n \)
  \[
  s_j^0 = s_{j-1}^0 + \exp(x_j^0) .
  \]
- \( f^0 = \log(s_n^0) \)
- **Output:** scalar value \( f^0 \)
Forward Mode

\[ f^{(1)}(x^0) \cdot x^1 \]

- Input: \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s^1_i = \exp(x^0_i)x^1_i \).
- For \( j = 2, \ldots, n \)
  \[ s^1_j = s^1_{j-1} + \exp(x^0_j)x^1_j. \]
- \( f^1 = s^1_n/s^0_n \)
- Output: scalar value \( f^1 \)
Forward Mode

\[ f^{(1)}(x^0) \cdot x^1 \]

- **Input:** \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s^1_1 = \exp(x^0_1)x^1_1 \).
- For \( j = 2, \ldots, n \) \( s^1_j = s^1_{j-1} + \exp(x^0_j)x^1_j \).
- \( f^1 = s^1_n/s^0_n \)
- **Output:** scalar value \( f^1 \)
Forward Mode

\[ f^{(1)}(x^0) \cdot x^1 \]

- **Input:** \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s_1^1 = \exp(x_1^0)x_1^1 \).
- For \( j = 2, \ldots, n \)
  \[ s_j^1 = s_{j-1}^1 + \exp(x_j^0)x_j^1. \]
- \( f^1 = s_n^1/s_n^0 \)
- **Output:** scalar value \( f^1 \)
Forward Mode

\[
f^{(1)}(x^0) \cdot x^1
\]

- Input: \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s_1^1 = \exp(x_1^0)x_1^1 \).
- For \( j = 2, \ldots, n \)
  \[
s_j^1 = s_{j-1}^1 + \exp(x_j^0)x_j^1.
\]
- \( f^1 = s_n^1/s_n^0 \)
- Output: scalar value \( f^1 \)
Forward Mode

\[ f^{(1)}(x^0) \cdot x^1 \]

- Input: \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s^1_1 = \exp(x^0_1)x^1_1 \).
- For \( j = 2, \ldots, n \)
  \[ s^1_j = s^1_{j-1} + \exp(x^0_j)x^1_j. \]
- \( f^1 = s^1_n/s^0_n \)
- Output: scalar value \( f^1 \)
Forward Mode

\[ f^{(1)}(x^0) \cdot x^1 \]

- Input: \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R} \).
- \( s_1^1 = \exp(x^0_1) x^1_1 \).
- For \( j = 2, \ldots, n \)
  \[ s_j^1 = s_{j-1}^1 + \exp(x^0_j) x^1_j. \]
- \( f^1 = s_n^1 / s_n^0 \)
- Output: scalar value \( f^1 \)
Reverse Mode

\[ w^1 f^{(1)}(x^0) \]

- Input: \( x^0 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R}, w^1 \in \mathbb{R} \).
- \( s_n^1 = w^1 / s_n^0 \)
- For \( j = n, \ldots, 2 \)
  \[ s_{j-1}^1 = s_j^1 \]
  \[ x_j^1 = \exp(x_j^0) s_j^1 . \]
- \( x_1^1 = \exp(x_1^0) s_1^1 . \)
- Output: vector value \( x^1 \in \mathbb{R}^n \)
Reverse Mode

\[ w^1 f^{(1)}(x^0) \]

- **Input:** \( x^0 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R}, w^1 \in \mathbb{R} \).
- \( s^n_1 = w^1 / s^n_1 \)
- For \( j = n, \ldots, 2 \)
  \[ s_{j-1}^1 = s_j^1 \]
  \[ x_j^1 = \exp(x_j^0) s_j^1. \]
- \( x_1^1 = \exp(x_1^0) s_1^1. \)
- **Output:** vector value \( x^1 \in \mathbb{R}^n \)
Reverse Mode

\[ w^1 f^{(1)}(x^0) \]

Input: \( x^0 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R}, w^1 \in \mathbb{R}. \)

\( s_n^1 = w^1 / s_n^0 \)

For \( j = n, \ldots , 2 \)

\[ s_{j-1}^1 = s_j^1 \]

\[ x_j^1 = \exp(x_j^0) s_j^1. \]

\( x_1^1 = \exp(x_1^0) s_1^1. \)

Output: vector value \( x^1 \in \mathbb{R}^n \)
Reverse Mode

\[ w^1 f^{(1)}(x^0) \]

- **Input:** \( x^0 \in \mathbb{R}^n, \ s^0 \in \mathbb{R}^n, \ f^0 \in \mathbb{R}, \ w^1 \in \mathbb{R}. \)
- \( s_n^1 = w^1 / s_n^0 \)
- For \( j = n, \ldots, 2 \)
  \[ s_{j-1}^1 = s_j^1 \]
  \[ x_j^1 = \exp(x_0^j) s_j^1. \]
- \( x_1^1 = \exp(x_0^1) s_1^1. \)
- **Output:** vector value \( x^1 \in \mathbb{R}^n \)
\[ w^1 f^{(1)}(x^0) \]

- **Input:** \( x^0 \in \mathbb{R}^n, \ s^0 \in \mathbb{R}^n, \ f^0 \in \mathbb{R}, \ w^1 \in \mathbb{R}. \)
- \( s_n^1 = w^1 / s_n \)
- For \( j = n, \ldots, 2 \)
  \[ s_{j-1}^1 = s_j^1 \]
  \[ x_j^1 = \exp(x_j^0) s_j^1. \]
- \( x_1^1 = \exp(x_1^0) s_1^1. \)
- **Output:** vector value \( x^1 \in \mathbb{R}^n \)
Reverse Mode

\[ w^1 f^{(1)}(x^0) \]

- Input: \( x^0 \in \mathbb{R}^n, s^0 \in \mathbb{R}^n, f^0 \in \mathbb{R}, w^1 \in \mathbb{R} \).

- \( s_n^1 = w^1 / s_n^0 \)

- For \( j = n, \ldots, 2 \)
  \[
  s_{j-1}^1 = s_j^1 \\
  x_j^1 = \exp(x_j^0) s_j^1.
  \]

- \( x_1^1 = \exp(x_1^0) s_1^1 \).

- Output: vector value \( x^1 \in \mathbb{R}^n \)
Motivation: $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$

- Given $x^0 \in \mathbb{R}^n$, $x^1 \in \mathbb{R}^n$, forward mode computes $g^1 \in \mathbb{R}^m$ where $g^1 = g^{(1)}(x^0) \cdot x^1$
- Given $x^0 \in \mathbb{R}^n$, $w^1 \in \mathbb{R}^m$, reverse mode computes $x^1 \in \mathbb{R}^n$ where $x^1 = (w^1)^T g^{(1)}(x^0)$
- Forward mode can be implemented without storing intermediate values; e.g., $s_1, \ldots, s_n$ above.
- Reverse mode requires storing intermediate values.
- Checkpointing reduces memory during reverse mode at the expense of extra computations.
Motivation: $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$

- Given $x^0 \in \mathbb{R}^n$, $x^1 \in \mathbb{R}^n$, forward mode computes $g^1 \in \mathbb{R}^m$ where $g^1 = g^{(1)}(x^0) \cdot x^1$
- Given $x^0 \in \mathbb{R}^n$, $w^1 \in \mathbb{R}^m$, reverse mode computes $x^1 \in \mathbb{R}^n$ where $x^1 = (w^1)^T g^{(1)}(x^0)$

- Forward mode can be implemented without storing intermediate values; e.g., $s_1, \ldots, s_n$ above.
- Reverse mode requires storing intermediate values.
- Checkpointing reduces memory during reverse mode at the expense of extra computations.
Motivation: $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$

- Given $x^0 \in \mathbb{R}^n$, $x^1 \in \mathbb{R}^n$, forward mode computes $g^1 \in \mathbb{R}^m$ where $g^1 = g^{(1)}(x^0) \cdot x^1$
- Given $x^0 \in \mathbb{R}^n$, $w^1 \in \mathbb{R}^m$, reverse mode computes $x^1 \in \mathbb{R}^n$ where $x^1 = (w^1)^T g^{(1)}(x^0)$
- Forward mode can be implemented without storing intermediate values; e.g., $s_1, \ldots, s_n$ above.
- Reverse mode requires storing intermediate values.
- Checkpointing reduces memory during reverse mode at the expense of extra computations.
Motivation: \( g : \mathbb{R}^n \rightarrow \mathbb{R}^m \)

- Given \( x^0 \in \mathbb{R}^n, x^1 \in \mathbb{R}^n \), forward mode computes \( g^1 \in \mathbb{R}^m \)
  where \( g^1 = g^{(1)}(x^0) \cdot x^1 \)
- Given \( x^0 \in \mathbb{R}^n, w^1 \in \mathbb{R}^m \), reverse mode computes \( x^1 \in \mathbb{R}^n \)
  where \( x^1 = (w^1)^T g^{(1)}(x^0) \)
- Forward mode can be implemented without storing intermediate values; e.g., \( s_1, \ldots, s_n \) above.
- Reverse mode requires storing intermediate values.
- Checkpointing reduces memory during reverse mode at the expense of extra computations.
Motivation: $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$

- Given $x^0 \in \mathbb{R}^n$, $x^1 \in \mathbb{R}^n$, forward mode computes $g^1 \in \mathbb{R}^m$ where $g^1 = g^{(1)}(x^0) \cdot x^1$
- Given $x^0 \in \mathbb{R}^n$, $w^1 \in \mathbb{R}^m$, reverse mode computes $x^1 \in \mathbb{R}^n$ where $x^1 = (w^1)^T g^{(1)}(x^0)$
- Forward mode can be implemented without storing intermediate values; e.g., $s_1, \ldots, s_n$ above.
- Reverse mode requires storing intermediate values.
- Checkpointing reduces memory during reverse mode at the expense of extra computations.
Conjugate Gradient Algorithm: I

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

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

- **Output:** vector value \( x_k \)
Conjugate Gradient Algorithm: I

- **Input:** \( A \in \mathbb{R}^{n \times n}, \ b \in \mathbb{R}^n, \ x_0 \in \mathbb{R}^n, \ \varepsilon \in \mathbb{R}_+ \).
- \( g_0 = Ax_0 - b, \ s_0 = g_0^T d_0, \ d_0 = -g_0, \ k = 0. \)
- **While** \( \sqrt{s_k} > \varepsilon \)
  - \( \alpha_{k+1} = s_k / (d_k^T Ad_k) \)
  - \( x_{k+1} = x_k + \alpha_{k+1} d_k \)
  - \( g_{k+1} = g_k + \alpha_{k+1} Ad_k \)
  - \( s_{k+1} = g_{k+1}^T g_{k+1} \cdot d_{k+1} = -g_{k+1}^T + (s_{k+1}/s_k) d_k \)
  - \( k = k + 1. \)

- **Output:** vector value \( x_k \)
Conjugate Gradient Algorithm: I

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

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

- **Output:** vector value $x_k$.
Conjugate Gradient Algorithm: I

- **Input:** $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, $\varepsilon \in \mathbb{R}_+$.  

- $g_0 = Ax_0 - b$, $s_0 = g_0^T d_0$, $d_0 = -g_0$, $k = 0$.

- **While** $\sqrt{s_k} > \varepsilon$
  
  \[
  \alpha_{k+1} = s_k / (d_k^T A d_k) \\
  x_{k+1} = x_k + \alpha_{k+1} d_k \\
  g_{k+1} = g_k + \alpha_{k+1} A d_k \\
  s_{k+1} = g_{k+1}^T g_{k+1} \\
  d_{k+1} = -g_{k+1}^T + (s_{k+1} / s_k) d_k \\
  k = k + 1.
  \]

- **Output:** vector value $x_k$
Conjugate Gradient Algorithm: II

- **Input:** $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, $\varepsilon \in \mathbb{R}_+$.  
- $g_0 = Ax_0 - b$, $s_0 = g_0^T d_0$, $d_0 = -g_0$, $k = 0$.  
- 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 double.
Conjugate Gradient Algorithm: II

- **Input:** \( A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}_+ \).

- \( g_0 = Ax_0 - b, \ s_0 = g_0^T d_0, \ d_0 = -g_0, \ k = 0. \)

- 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 \( \{s_k\}, \{g_k\}, \{x_k\}, \text{and} \ \{d_k\}. \) Other intermediate values are recalculated.
  - Overloaded AD type only records \( H(s, g, x, d) \), other calculations are in double.
Conjugate Gradient Algorithm: II

- **Input:** \( A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}_+ \).

- \( g_0 = Ax_0 - b, \ s_0 = g_0^T d_0, \ d_0 = -g_0, \ k = 0. \)

- **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 \( \{s_k\}, \{g_k\}, \{x_k\}, \text{and} \ \{d_k\}. \)
  - Other intermediate values are recalculated.
  - Overloaded AD type only records \( H(s, g, x, d) \), other calculations are in double.
Conjugate Gradient Algorithm: II

- **Input:** \( A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}_+ \).
- \( g_0 = Ax_0 - b, \quad s_0 = g_0^T d_0, \quad d_0 = -g_0, \quad k = 0. \)
- 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 double.
Conjugate Gradient Algorithm: II

Input: \( A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}_+ \).

\( g_0 = Ax_0 - b, \quad s_0 = g_0^T d_0, \quad d_0 = -g_0, \quad k = 0. \)

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 double.
Algorithm

```python
import numpy

def cg_iterate( s, g, d, x, A ):
    A_d = A( d )  # Ad_k
    d_A_d = numpy.inner(d, A_d)  # d^T Ad_k
    alpha = s / d_A_d  # \alpha_{k+1} = s_k/(d_k^T Ad_k)
    xp = x + alpha * d  # x_{k+1} = x_k + \alpha_{k+1}d_k
    gp = g + alpha * A_d  # g_{k+1} = g_k + \alpha_{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)
```

Bradley M. Bell

AD of Sparse Inversions
Algorithm

```python
def cg_iterate(s, g, d, x, A):
    A_d = A(d)  # $A_d$
    d_A_d = numpy.inner(d, A_d)  # $d^T A_d$
    alpha = s / d_A_d  # $\alpha_{k+1} = s_k / (d_k^T A d_k)$
    xp = x + alpha * d  # $x_{k+1} = x_k + \alpha_{k+1} d_k$
    gp = g + alpha * A_d  # $g_{k+1} = g_k + \alpha_{k+1} A d_k$
    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)
```

Bradley M. Bell  
AD of Sparse Inversions
import numpy
def cg_iterate( s, g, d, x, A ):
    A_d = A(d)
    d_A_d = numpy.inner(d, A_d)
    alpha = s / d_A_d
    xp = x + alpha * d
    gp = g + alpha * A_d
    sp = numpy.inner(gp, gp)
    dp = - gp + (sp / s) * d
    return ( sp, gp, dp, xp )

# Ad_k
# d^T_A d_k
# α_{k+1} = s_k / (d^T_k A d_k)
# x_{k+1} = x_k + α_{k+1} d_k
# g_{k+1} = g_k + α_{k+1} A d_k
# s_{k+1} = g^T_{k+1} g_{k+1}
# d_{k+1} = -g_{k+1} + (s_{k+1} / s_k) d_k
Algorithm

```python
import numpy

def cg_iterate(s, g, d, x, A):
    A_d = A(d)
    d_A_d = numpy.inner(d, A_d)
    alpha = s / d_A_d

    xp = x + alpha * d
    gp = g + alpha * A_d
    sp = numpy.inner(gp, gp)
    dp = -gp + (sp / s) * d

    return (sp, gp, dp, xp)
```

# $A d_k$
# $d_k^T A d_k$
# $\alpha_{k+1} = s_k / (d_k^T A d_k)$
# $x_{k+1} = x_k + \alpha_{k+1} d_k$
# $g_{k+1} = g_k + \alpha_{k+1} A d_k$
# $s_{k+1} = g_{k+1}^T g_{k+1}$
# $d_{k+1} = -g_{k+1} + (s_{k+1} / s_k) d_k$
Algorithm

```python
import numpy
def cg_iterate(s, g, d, x, A):
    A_d = A(d)
    d_A_d = numpy.inner(d, A_d)
    alpha = s / d_A_d
    xp = x + alpha * d
    gp = g + alpha * A_d
    sp = numpy.inner(gp, gp)
    dp = -gp + (sp / s) * d
    return (sp, gp, dp, xp)
```

# $A_d^k$
# $d^T A d^k$
# $\alpha_{k+1} = s_k / (d^T A d^k)$
# $x_{k+1} = x_k + \alpha_{k+1} d_k$
# $g_{k+1} = g_k + \alpha_{k+1} A d_k$
# $s_{k+1} = g^{T}_{k+1} g_{k+1}$
# $d_{k+1} = -g_{k+1} + (s_{k+1} / s_k) d_k$
import numpy
def cg_iterate ( s, g, d, x, A ) :
    A_d = A ( d )
    d_A_d = numpy.inner(d, A_d)
    alpha = s / d_A_d
    xp = x + alpha * d
    gp = g + alpha * A_d
    sp = numpy.inner(gp, gp)
    dp = - gp + (sp / s) * d
    return ( sp, gp, dp, xp )

# Ad_k
# d_k^T Ad_k
# α_k+1 = s_k / (d_k^T Ad_k)
# x_k+1 = x_k + α_k+1 d_k
# g_k+1 = g_k + α_k+1 Ad_k
# s_k+1 = g_k^T g_k+1
# d_k+1 = -g_k+1 + (s_k+1 / s_k) d_k
```python
import numpy
def cg_iterate(s, g, d, x, A):
    A_d = A(d)  # Ad_k
    d_A_d = numpy.inner(d, A_d)  # d^T_k Ad_k
    alpha = s / d_A_d  # \alpha_{k+1} = s_k / (d^T_k Ad_k)
    xp = x + alpha * d  # x_{k+1} = x_k + \alpha_{k+1} d_k
    gp = g + alpha * A_d  # g_{k+1} = g_k + \alpha_{k+1} Ad_k
    sp = numpy.inner(gp, gp)  # s_{k+1} = g^T_{k+1} 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)
```

Bradley M. Bell  
AD of Sparse Inversions
import numpy

def cg_iterate(s, g, d, x, A):
    A_d = A(d)  # $A_d$
    d_A_d = numpy.inner(d, A_d)  # $d^T A_d$
    alpha = s / d_A_d  # $\alpha_{k+1} = s_k / (d_k^T A d_k)$
    xp = x + alpha * d  # $x_{k+1} = x_k + \alpha_{k+1} d_k$
    gp = g + alpha * A_d  # $g_{k+1} = g_k + \alpha_{k+1} A d_k$
    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)
import numpy

def cg_iterate(s, g, d, x, A):
    A_d = A(d)
    # \( A_d \)
    d_A_d = numpy.inner(d, A_d)
    # \(d_d \)
    alpha = s / d_A_d
    # \( \alpha \)
    xp = x + alpha * d
    # \( x_{k+1} \)
    gp = g + alpha * A_d
    # \( g_{k+1} \)
    sp = numpy.inner(gp, gp)
    # \( s_{k+1} \)
    dp = -gp + (sp / s) * d
    # \( d_{k+1} \)
    return (sp, gp, dp, xp)
Algorithm

```python
import numpy

def cg_iterate ( s, g, d, x, A ) :
    A_d = A ( d )
    d_A_d = numpy.inner(d, A_d)
    alpha = s / d_A_d
    xp = x + alpha * d
    gp = g + alpha * A_d
    sp = numpy.inner(gp, gp)
    dp = - gp + (sp / s) * d
    return ( sp, gp, dp, xp )
```
Example Problem

- Solve $\begin{pmatrix} \begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ for $x \in \mathbb{R}^2$.

- $D = \begin{pmatrix} \begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $Dx = c$

- 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.

- $A = D^T D$, $b = D^T c$, $Ax = b$

- $A$ is positive definite.
Example Problem

- Solve \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \) for \( x \in \mathbb{R}^2 \).

- \( D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \), \( c = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \), \( Dx = c \)

- 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.

- \( A = D^T D \), \( b = D^T c \), \( Ax = b \)

- \( A \) is positive definite.
Example Problem

- Solve \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \) for \( x \in \mathbb{R}^2 \).
- \( D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \ c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \ Dx = c \)

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.

- \( A = D^T D, \ b = D^T c, \ Ax = b \)
- \( A \) is positive definite.
Example Problem

- Solve \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \) for \( x \in \mathbb{R}^2 \).

- \( D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \), \( c = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \), \( Dx = c \)

- 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.

- \( A = D^T D \), \( b = D^T c \), \( Ax = b \)

- \( A \) is positive definite.
Example Problem

- Solve $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ for $x \in \mathbb{R}^2$.

- $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $c = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $Dx = c$

- 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.

- $A = D^T D$, $b = D^T c$, $Ax = b$

- $A$ is positive definite.
Example Problem

- Solve \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \) for \( x \in \mathbb{R}^2 \).

- \( D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \),  \( c = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \),  \( Dx = c \)

- 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.

- \( A = D^T D \),  \( b = D^T c \),  \( Ax = b \)

- \( A \) is positive definite.
Example Problem

- Solve \( \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \) for \( x \in \mathbb{R}^2 \).

- \( D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \), \( c = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \), \( Dx = c \)

- 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.

- \( A = D^T D \), \( b = D^T c \), \( Ax = b \)

- \( A \) is positive definite.
Compute matrix $D$ or $D^T$ times the vector $v$.

```python
def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u
```

# $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
# $u = 0$

# $D_{ij} = 2i + j + 1$

# $u_i = \sum_j D_{ij}v_j$

# $u_j = \sum_i v_iD_{ij}$
Example Matrix Times a Vector

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

def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u

# D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}
# u = 0
# D_{ij} = 2i + j + 1
# u_i = \sum_j D_{ij} v_j
# u_j = \sum_i v_i D_{ij}
Example Matrix Times a Vector

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

```python
def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u
```

# $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

# $u = 0$

# $D_{ij} = 2i + j + 1$

# $u_i = \sum_j D_{ij} v_j$

# $u_j = \sum_i v_i D_{ij}$
Compute matrix $D$ or $D^T$ times the vector $v$.

```python
def D(v, transpose) :
    u = numpy.array( [ 0. , 0. ] )
    for i in range(2) :
        for j in range(2) :
            D_ij = float(2 * i + j + 1)
            if not transpose :
                u[i] = u[i] + D_ij * v[j]
            else :
                u[j] = u[j] + D_ij * v[i]
    return u
```

# $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
# $u = 0$

# $D_{ij} = 2i + j + 1$

# $u_i = \sum_j D_{ij}v_j$
# $u_j = \sum_i v_iD_{ij}$
Example Matrix Times a Vector

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

```python
def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u
```

# $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
# $u = 0$
# $D_{ij} = 2i + j + 1$
# $u_i = \sum_j D_{ij}v_j$
# $u_j = \sum_i v_iD_{ij}$
Compute matrix $D$ or $D^T$ times the vector $v$.

```python
def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u
```

# $D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$
# $u = 0$
# $D_{ij} = 2i + j + 1$
# $u_i = \sum_j D_{ij}v_j$
# $u_j = \sum_i v_iD_{ij}$
Example Matrix Times a Vector

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

```python
def D(v, transpose):
    u = numpy.array([0., 0.])
    for i in range(2):
        for j in range(2):
            D_ij = float(2 * i + j + 1)
            if not transpose:
                u[i] = u[i] + D_ij * v[j]
            else:
                u[j] = u[j] + D_ij * v[i]
    return u
```

$D = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$  
$u = 0$

$D_{ij} = 2i + j + 1$

$u_i = \sum_j D_{ij}v_j$

$u_j = \sum_i v_iD_{ij}$
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)
c = numpy.array([1., 1.])
b = D(c, True)
k = 0
x = numpy.array([0., 0.])
g = A(x) - b
s = numpy.inner(g, g)
d = -g
```

# $A = D^T D$
# $c = (1, 1)^T$
# $b = D^T c$
# $x_0 = 0$
# $g_0 = Ax_0 - b$
# $s_0 = g_0^T g_0$
# $d_0 = -g_0$
Example Setup

Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)
c = numpy.array([1., 1.])
b = D(c, True)
k = 0
x = numpy.array([0., 0.])
g = A(x) - b
s = numpy.inner(g, g)
d = -g
```

$# A = D^T D$
$# c = (1, 1)^T$
$# b = D^T c$
$# x_0 = 0$
$# g_0 = Ax_0 - b$
$# s_0 = g_0^T g_0$
$# d_0 = -g_0$
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)

c = numpy.array([1., 1.])
# c = (1, 1)^T
b = D(c, True)
# b = D^Tc
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^Tg_0
d = -g
# d_0 = -g_0
```
Example Setup

Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)
c = numpy.array([1., 1.])  # $c = (1, 1)^T$
b = D(c, True)  # $b = D^Tc$
k = 0
x = numpy.array([0., 0.])  # $x_0 = 0$
g = A(x) - b  # $g_0 = Ax_0 - b$
s = numpy.inner(g, g)  # $s_0 = g_0^Tg_0$
d = -g  # $d_0 = -g_0$
```
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)  # $A = D^T D$

c = numpy.array([1., 1.])  # $c = (1, 1)^T$

b = D(c, True)  # $b = D^T c$

k = 0

x = numpy.array([0., 0.])  # $x_0 = 0$

f = A(x) - b  # $g_0 = Ax_0 - b$

s = numpy.inner(g, g)  # $s_0 = g_0^T g_0$

d = -g  # $d_0 = -g_0$
```
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)

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 = A(x_0) - b
s = numpy.inner(g, g)
# s_0 = g^T g_0
d = -g
# d_0 = -g_0
```

Bradley M. Bell
AD of Sparse Inversions
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)

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
```
Example Setup

Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)

k = 0
x = numpy.array([0., 0.])
g = A(x) - b
s = numpy.inner(g, g)
d = -g
```

# $A = D^T D$
# $c = (1, 1)^T$
# $b = D^T c$
# $x_0 = 0$
# $g_0 = Ax_0 - b$
# $s_0 = g_0^T g_0$
# $d_0 = -g_0$
Define $A(v)$ and set initial values input to cg_iterate.

```python
def A(v):
    return D(D(v, False), True)
c = numpy.array([1., 1.])
b = D(c, True)
k = 0
x = numpy.array([0., 0.])
g = A(x) - b
s = numpy.inner(g, g)
d = -g
```

# $A = D^T D$

# $c = (1, 1)^T$

# $b = D^T c$

# $x_0 = 0$

# $g_0 = Ax_0 - b$

# $s_0 = g_0^T g_0$

# $d_0 = -g_0$
Example Setup

Define $A(v)$ and set initial values input to `cg_iterate`.

```python
def A(v):
    return D(D(v, False), True)
c = numpy.array([1., 1.])  # $c = (1, 1)^T$
b = D(c, True)               # $b = D^Tc$
k = 0                        # $x_0 = 0$
x = numpy.array([0., 0.])    # $g_0 = Ax_0 - b$
g = A(x) - b                 # $s_0 = g_0^Tg_0$
s = numpy.inner(g, g)        # $d_0 = -g_0$
d = -g                        # $A = D^T D$
```

Bradley M. Bell  AD of Sparse Inversions
Example Iteration and Validation

```python
while math.sqrt(s) > 1e-7:  # 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 for $j = 0, 1$, $|x_j - v_j| < 2\varepsilon$
assert all(abs(x - v) < 2*eps)  # check that at most two iterations were used
assert k <= 2
```
**Example Iteration and Validation**

```python
while math.sqrt(s) > 1e-7:  # 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 for $j = 0, 1$, $|x_j - v_j| < 2\varepsilon$
assert all( abs( x - v ) < 2 * eps )

# check that at most two iterations were used
assert k <= 2
```
Example Iteration and Validation

```python
while math.sqrt(s) > 1e-7 :  # 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 for \( j = 0, 1 \), \(|x_j - v_j| < 2\varepsilon\)
assert all( abs( x - v ) < 2 * eps )

# check that at most two iterations were used
assert k <= 2
```

Bradley M. Bell  AD of Sparse Inversions
while math.sqrt(s) > 1e-7 :  # 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 for $j = 0, 1$, $|x_j - v_j| < 2\varepsilon$
assert all( abs( x - v ) < 2 * eps )

# check that at most two iterations were used
assert k <= 2
Example Iteration and Validation

```python
while math.sqrt(s) > 1e-7:  # 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([ -1.0, 1.0 ])  

    # check that for $j = 0, 1$, $|x_j - v_j| < 2\varepsilon$
    assert all(abs(x - v) < 2 * eps)

    # check that at most two iterations were used
    assert k <= 2
```

Bradley M. Bell  AD of Sparse Inversions
Example Iteration and Validation

```python
while math.sqrt(s) > 1e-7:  # 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 for \( j = 0, 1, |x_j - v_j| < 2\varepsilon \)
assert all( abs( x - v ) < 2 * eps )

# check that at most two iterations were used
assert k <= 2
```
Recording the cg_iterate Function

Let $u$ be vector of non-zeros in representation of $D$.

- # $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.
  
  $y = \text{pack}_\text{sgdxu}(s, g, d, x, u)$

- # Start recording with $ay$ as independent variable vector.
  
  $ay = \text{pycppad.independent}(y)$

- Extract $au$ from $ay$ and use it to define $aA(v)$.

- # Define $F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.

  $as, ag, ad, ax = \text{unpack}_\text{sgdx}(n, ay)$

  $as, ag, ad, ax = \text{cg_iterate}(as, ag, ad, ax, aA)$

  $az = \text{pack}_\text{sgdx}(n, as, ag, ad, ax)$

  $F = \text{pycppad.adfun}(ay, az)$
Recording the cg_iterate Function

- Let $u$ be vector of non-zeros in representation of $D$.
- # $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.
  
  $y = \text{pack}_\text{sgd}_x\text{u}(s, g, d, x, u)$
  
- # Start recording with $ay$ as independent variable vector.
  
  $ay = \text{pycppad.independent}(y)$
  
- Extract $au$ from $ay$ and use it to define $aA(\nu)$.
  
- # Define $F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
  
  $as, ag, ad, ax = \text{unpack}_\text{sgd}_x(n, ay)$

  $as, ag, ad, ax = \text{cg}_\text{iterate}(as, ag, ad, ax, aA)$

  $az = \text{pack}_\text{sgd}_x(n, as, ag, ad, ax)$

  $F = \text{pycppad.adfun}(ay, az)$
Recording the cg_iterate Function

- Let $u$ be vector of non-zeros in representation of $D$.
- $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.
  \[
  y = \text{pack}_\text{sgdxu}(s, g, d, x, u)
  \]
- Start recording with $ay$ as independent variable vector.
  \[
  ay = \text{pycppad.independent}(y)
  \]
- Extract $au$ from $ay$ and use it to define $aA(v)$.
- Define $F: (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
  \[
  \text{as}, \text{ag}, \text{ad}, \text{ax} = \text{unpack}_\text{sgdx}(n, ay)
  \]
  \[
  \text{as}, \text{ag}, \text{ad}, \text{ax} = \text{cg}_\text{iterate}(\text{as}, \text{ag}, \text{ad}, \text{ax}, aA)
  \]
  \[
  az = \text{pack}_\text{sgdx}(n, \text{as}, \text{ag}, \text{ad}, \text{ax})
  \]
  \[
  F = \text{pycppad.adfun}(ay, az)
  \]
Recording the cg_iterate Function

- Let \( u \) be vector of non-zeros in representation of \( D \).
- \# \( y \) is vector containing \((s_0, g_0, d_0, x_0, u)\).
  \[
y = \text{pack}_\text{sgd}_\text{xu}(s, g, d, x, u)
\]
- \# Start recording with \( ay \) as independent variable vector.
  \[
  ay = \text{pycppad}.\text{independent}(y)
  \]
- Extract \( au \) from \( ay \) and use it to define \( aA(\nu) \).
- \# Define \( F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1}) \).
  \[
  \text{as, ag, ad, ax} = \text{unpack}_\text{sgd}_\text{x}(n, ay)
  \]
  \[
  \text{as, ag, ad, ax} = \text{cg}_\text{iterate}(\text{as, ag, ad, ax, aA})
  \]
  \[
  az = \text{pack}_\text{sgd}_\text{x}(n, \text{as, ag, ad, ax})
  \]
  \[
  F = \text{pycppad}.\text{adfun}(ay, az)
  \]
Let $u$ be vector of non-zeros in representation of $D$.

1. # $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.
   
   $y = \text{pack}_\text{sgdxu}(s, g, d, x, u)$

2. # Start recording with $ay$ as independent variable vector.
   
   $ay = \text{pycppad.independent}(y)$

3. Extract $au$ from $ay$ and use it to define $aA(v)$.

4. # Define $F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
   
   $as, ag, ad, ax = \text{unpack}_\text{sgdx}(n, ay)$

   $as, ag, ad, ax = \text{cg_iterate}(as, ag, ad, ax, aA)$

   $az = \text{pack}_\text{sgdx}(n, as, ag, ad, ax)$

   $F = \text{pycppad.adfun}(ay, az)$
Recording the cg_iterate Function

- Let $u$ be vector of non-zeros in representation of $D$.
- $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.
  
  ```
y = pack_sgdxu(s, g, d, x, u)
  ```
- Start recording with $ay$ as independent variable vector.
  
  ```
  ay = pycppad.independent(y)
  ```
- Extract $au$ from $ay$ and use it to define $aA(v)$.
- Define $F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.
  
  ```
as, ag, ad, ax = unpack_sgdx(n, ay)
as, ag, ad, ax = cg_iterate(as , ag, ad , ax, aA)
az = pack_sgdx(n, as , ag, ad , ax )
F = pycppad.adfun(ay, az)
  ```
Let $u$ be vector of non-zeros in representation of $D$.

$y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.

```python
y = pack_sgdxu(s, g, d, x, u)
```

Start recording with $ay$ as independent variable vector.

```python
ay = pycppad.independent(y)
```

Extract $au$ from $ay$ and use it to define $aA(v)$.

```python
as, ag, ad, ax = unpack_sgdix(n, ay)
as, ag, ad, ax = cg_iterate(as, ag, ad, ax, aA)
az = pack_sgdix(n, as, ag, ad, ax)
F = pycppad.adfun(ay, az)
```
Recording the cg_iterate Function

- Let \( u \) be vector of non-zeros in representation of \( D \).
- \# \( y \) is vector containing \((s_0, g_0, d_0, x_0, u)\).
  \[
  y = \text{pack}_\text{sgdxu}(s, g, d, x, u)
  \]
- \# Start recording with \( ay \) as independent variable vector.
  \[
  ay = \text{pycppad.independent}(y)
  \]
- Extract \( au \) from \( ay \) and use it to define \( aA(\nu) \).
- \# Define \( F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1}) \).
  \[
  \begin{align*}
  &\text{as, ag, ad, ax } = \text{unpack}_\text{sgdx}(n, ay) \\
  &\text{as, ag, ad, ax } = \text{cg_iterate}(aA) \\
  &az = \text{pack}_\text{sgdx}(n, as, ag, ad, ax) \\
  &F = \text{pycppad.adfun}(ay, az)
  \end{align*}
  \]
Recording the cg_iterate Function

- Let $u$ be vector of non-zeros in representation of $D$.
- $y$ is vector containing $(s_0, g_0, d_0, x_0, u)$.

\[
y = \text{pack}_\text{sgdx}(s, g, d, x, u)
\]

- Start recording with $ay$ as independent variable vector.

\[
ay = \text{pycppad}\text{.independent}(y)
\]

- Extract $au$ from $ay$ and use it to define $aA(\nu)$.

- Define $F : (s_k, g_k, d_k, x_k, u) \rightarrow (s_{k+1}, g_{k+1}, d_{k+1}, x_{k+1})$.

\[
as, ag, ad, ax = \text{unpack}_\text{sgdx}(n, ay)
\]

\[
as, ag, ad, ax = \text{cg}_\text{iterate}(as, ag, ad, ax, aA)
\]

\[
az = \text{pack}_\text{sgdx}(n, as, ag, ad, ax)
\]

\[
F = \text{pycppad}\text{.adfun}(ay, az)
\]
Iterate Forward Storing Checkpoints

# Initialize:
checkpoint = []

\[
z = \text{pack\_sgdx}(s, g, d, x)
\]

while math.sqrt(z[0]) > epsilon:
    \# z[0] = s
    \# forward mode takes a single vector argument
    y = \text{pack\_sgdxu}(z, u)

    \# Store forward input, except u which is constant
    checkpoint.append(z)

    \# Use F to compute \((s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})\)
    z = F.forward(0, y)
Iterate Forward Storing Checkpoints

# Initialize:
checkpoints = []
z = pack_sgdx(s, g, d, x)

while math.sqrt(z[0]) > epsilon:
    # forward mode takes a single vector argument
    y = pack_sgdxu(z, u)
    # Store forward input, except u which is constant
    checkpoints.append(z)
    # Use F to compute (s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})
    z = F.forward(0, y)
Iterate Forward Storing Checkpoints

# Initialize:
checkpoint = [ ]
z = pack_sgd(s, g, d, x)

while math.sqrt( z[0] ) > epsilon : # z[0] = s
    # forward mode takes a single vector argument
    y = pack_sgdxu(z, u)

    # Store forward input, except u which is constant
    checkpoint.append( z )

    # Use F to compute (s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})
z = F.forward(0, y)
Iterate Forward Storing Checkpoints

# Initialize:
checkpoint = [ ]
z = pack_sgdx(s, g, d, x)

while math.sqrt(z[0]) > epsilon :
    # z[0] = s
    y = pack_sgdxu(z, u)
    # forward mode takes a single vector argument
    checkpoint.append(z)
    # Use F to compute \((s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})\)
    z = F.forward(0, y)

Bradley M. Bell
AD of Sparse Inversions
Iterate Forward Storing Checkpoints

# Initialize:
checkpoint = [ ]
z = pack_sgdx(s, g, d, x)

while math.sqrt( z[0] ) > epsilon : # z[0] = s
    # forward mode takes a single vector argument
    y = pack_sgdxu(z, u)
    # Store forward input, except u which is constant
    checkpoint.append( z )

    # Use F to compute \( (s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1}) \)
z = F.forward(0, y)
# Initialize:
checkpoint = []
z = pack_sgdx(s, g, d, x)

while math.sqrt(z[0]) > epsilon:
    z[0] = s
    # forward mode takes a single vector argument
    y = pack_sgdxu(z, u)
    # Store forward input, except u which is constant
    checkpoint.append(z)

    # Use F to compute \((s_{k+1}, g_{k+1}, x_{k+1}, d_{k+1})\)
    z = F.forward(0, y)
Reverse Mode with Checkpointing

Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  
  $y = \text{pack}_{zu}(z, u)$
  
  $z = F.\text{forward}(0, y)$  # $z_{k+1}$ (not used)
  
  $dy = F.\text{reverse}(1, w)$
  
  $du = du + dy[\ len(z) : ]$  # $w \times \partial z_{k+1} \text{ w.r.t } u$
  
  $w = dy[ : \ len(z) ]$  # $w \times \partial z_{k+1} \text{ w.r.t } z_k$
Reverse Mode with Checkpointing

Compute $du$, the derivative of $w^Tz$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  y = pack_zu(z, u)
  z = F.forward(0, y)       # $z_{k+1}$ (not used)
  dy = F.reverse(1, w)
  du = du + dy[ len(z) : ] # $w \times$ partial $z_{k+1}$ w.r.t $u$
  w = dy[ : len(z) ]       # $w \times$ partial $z_{k+1}$ w.r.t $z_k$
Reverse Mode with Checkpointing

Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.

```python
for z in reversed(checkpoint):
    y = pack_zu(z, u)
    z = F.forward(0, y)  # z_{k+1} (not used)
    dy = F.reverse(1, w)
    du = du + dy[len(z) : ]  # w * partial z_{k+1} w.r.t u
    w = dy[:len(z)]  # w * partial z_{k+1} w.r.t z_k
```

Bradley M. Bell
AD of Sparse Inversions
Reverse Mode with Checkpointing

Compute $du$, the derivative of $w^Tz$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.

```python
for z in reversed(checkpoint):
    y = pack_zu(z, u)
    z = F.forward(0, y)  # z_{k+1} (not used)
    dy = F.reverse(1, w)
    du = du + dy[len(z):]  # w * partial z_{k+1} w.r.t u
    w = dy[:len(z)]       # w * partial z_{k+1} w.r.t z_k
```

Bradley M. Bell  
AD of Sparse Inversions
Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  
  ```python
  y = pack_zu(z, u)
  z = F.forward(0, y)  # z_k+1 (not used)
  dy = F.reverse(1, w)
  du = du + dy[len(z):]  # w * partial z_k+1 w.r.t u
  w = dy[:len(z)]  # w * partial z_k+1 w.r.t z_k
  ```
Compute $du$, the derivative of $w^Tz$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  
  $y = \text{pack}_zu(z, u)$
  $z = F.\text{forward}(0, y)$  # $z_{k+1}$ (not used)
  $dy = F.\text{reverse}(1, w)$
  $du = du + dy[len(z):]$  # $w$ * partial $z_{k+1}$ w.r.t $u$
  $w = dy[:len(z)]$  # $w$ * partial $z_{k+1}$ w.r.t $z_k$
Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  y = pack_zu(z, u)
  z = F.forward(0, y) # z_{k+1} (not used)
  dy = F.reverse(1, w)
  du = du + dy[ len(z) : ] # w * partial z_{k+1} w.r.t u
  w = dy[ : len(z) ] # w * partial z_{k+1} w.r.t z_k
Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- for $z$ in reversed(checkpoint) :
  
  $y = \text{pack}_zu(z, u)$
  $z = F.\text{forward}(0, y)$  # $z_{k+1}$ (not used)
  $dy = F.\text{reverse}(1, w)$
  $du = du + dy[ \text{len}(z) : ]$  # $w * \text{partial } z_{k+1}$ w.r.t $u$
  $w = dy[ : \text{len}(z) ]$  # $w * \text{partial } z_{k+1}$ w.r.t $z_k$
Compute $du$, the derivative of $w^T z$ w.r.t $u$ where $z \in \mathbb{R}^{1+3n}$ is the final value of $(s, g, d, x)$.

- If $w_i = 1$ and $w_j = 0$ for $j \neq i$, $du$ is derivative of $z_i$ w.r.t non-zeros in $D$.
- Initialize all components of the summation $du$ as zero.
- For $z$ in reversed(checkpoint):
  
  ```python
  y = pack_zu(z, u)
  z = F.forward(0, y)  # z_k+1 (not used)
  dy = F.reverse(1, w)
  du = du + dy[len(z):]  # w * partial z_k+1 w.r.t u
  w = dy[:len(z)]  # w * partial z_k+1 w.r.t z_k
  ```
Results

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

| $n$  | $k$  | $|c|$  | $|r|$         | $t$  |
|------|------|--------|--------------|------|
| 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

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

| $n$  | $k$ | $|c|$ | $|r|$    | $t$   |
|------|-----|------|---------|-------|
| 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

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

| $n$  | $k$  | $|c|$  | $|r|$       | $t$  |
|------|------|--------|-------------|------|
| 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

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

| $n$  | $k$  | $|c|$  | $|r|$ | $t$  |
|------|------|--------|-------|------|
| 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

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

| $n$  | $k$  | $|c|$  | $|r|$            | $t$         |
|------|------|--------|------------------|-------------|
| 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

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

| $n$ | $k$ | $|c|$ | $|r|$ | $t$ |
|-----|-----|------|------|-----|
| 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

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

| $n$  | $k$  | $|c|$    | $|r|$         | $t$  |
|------|------|----------|--------------|------|
| 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 |
$D \in \mathbb{R}^{n \times n}$ is diagonally dominant with 2% non-zero.

- $k$ is number of iterations for convergence.
- $c$ is right hand side.
- $r$ is the residual $Dx - c$.
- $t$ is number of seconds to compute derivative w.r.t $D$.

| $n$  | $k$ | $|c|$   | $|r|$      | $t$ |
|------|-----|--------|-----------|-----|
| 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

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

| $n$ | $k$ | $|c|$ | $|r|$ | $t$ |
|-----|-----|------|------|-----|
| 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 |
The required memory for reverse mode is $k(1 + 3n)$.

- We could recalculate $g_k$ and only require $k(1 + 2n)$.
- 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.
The required memory for reverse mode is $k(1 + 3n)$.

We could recalculate $g_k$ and only require $k(1 + 2n)$.

These ideas apply to many other iterative methods.

pyccppad obtains C++ Speed using boost-python and cppad.

The entire calculation could be done in C++ using cppad.
The required memory for reverse mode is $k(1 + 3n)$.

We could recalculate $g_k$ and only require $k(1 + 2n)$.

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 $k(1 + 3n)$.
- We could recalculate $g_k$ and only require $k(1 + 2n)$.
- 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.
The required memory for reverse mode is $k(1 + 3n)$.
We could recalculate $g_k$ and only require $k(1 + 2n)$.
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.
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
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_
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