Methods for large, unconstrained problems

- The Newton and Quasi-Newton methods require relative large amounts of memory \((n^2/2)\) element and floating point operations (flops) per iteration \(O(n^3)\) and \(O(n^2)\), respectively.

- On a 2.5GHz PC the numbers may look like:

<table>
<thead>
<tr>
<th>(n)</th>
<th>Memory ((n^2/2))</th>
<th>Time (O(n^3))</th>
<th>Time (O(n^2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>39 kB</td>
<td>3.2ms</td>
<td>60(\mu)s</td>
</tr>
<tr>
<td>1000</td>
<td>3.8 MB</td>
<td>1.76 s</td>
<td>7.6ms</td>
</tr>
<tr>
<td>10000</td>
<td>381 MB</td>
<td>29 min</td>
<td>0.76s</td>
</tr>
<tr>
<td>100000</td>
<td>37 GB</td>
<td>20 days</td>
<td>76 s</td>
</tr>
</tbody>
</table>

- For large \(n\), the computing and/or memory requirements are huge.

Conjugate vectors

- Assume a set of conjugate vectors \(\{p_i\}\) is known.
- Consider the vector \(y\) which is a sum of \(m+1\) such vectors, i.e.

\[
y = \sum_{i=0}^{m} \alpha_i p_i
\]

and calculate \(f(y) = \frac{1}{2} y^T A y - b^T y\):

\[
f(y) = \frac{1}{2} \left( \sum_{i=0}^{m} \alpha_i p_i \right)^T A \left( \sum_{j=0}^{m} \alpha_j p_j \right) - b^T \left( \sum_{i=0}^{m} \alpha_i p_i \right)
\]

\[
= \frac{1}{2} \sum_{i=0}^{m} \sum_{j=0}^{m} \alpha_i \alpha_j p_i^T A p_j - \sum_{i=0}^{m} \alpha_i b^T p_i
\]

\[
= \frac{1}{2} \sum_{i=0}^{m} \alpha_i^2 p_i^T A p_i - \sum_{i=0}^{m} \alpha_i b^T p_i
\]

\[
= \sum_{i=0}^{m} \left( \frac{1}{2} \alpha_i^2 p_i^T A p_i - \alpha_i b^T p_i \right).
\]
Minimizing over conjugate vectors

- It is then easy to minimize $f(x)$ over all such vectors $y$:

$$\min_y f(y) = \min_{\alpha_i} \left( \sum_{i=0}^{m} \alpha_i p_i \right) = \min_{\alpha_i} \sum_{i=0}^{m} \left( \frac{1}{2} \alpha_i^2 p_i^T A p_i - \alpha_i b^T p_i \right)$$

$$= \sum_{i=0}^{m} \min_{\alpha_i} \left( \frac{1}{2} \alpha_i^2 p_i^T A p_i - \alpha_i b^T p_i \right),$$

i.e. we have reduced the problem to a number of one-dimensional minimization problems.

- Each of these problems may be solved by setting the derivative with respect to $\alpha_i$ equal to zero:

$$\alpha (p_i^T A p_i) - b^T p_i = 0 \Rightarrow \alpha_i = \frac{b^T p_i}{p_i^T A p_i}.$$

- The conjugate-gradient method iteratively determines the set of conjugate vectors $\{p_i\}$ and their coefficients $\{\alpha_i\}$.

---

### Example

Let $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$, $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $r_0 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$, $\beta_0 = 0$.

- $p_0 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$, $\alpha_0 = 0.38$.

- $x_1 = \begin{bmatrix} 0.385 \\ 1.15 \end{bmatrix}$, $r_1 = \begin{bmatrix} -0.92 \\ 0.31 \end{bmatrix}$, $\beta_1 = 0.095$.

- $p_1 = \begin{bmatrix} -0.83 \\ 0.59 \end{bmatrix}$, $\alpha_1 = 0.87$.

- $x_2 = \begin{bmatrix} -0.33 \\ 1.67 \end{bmatrix}$, $r_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

---

### The Conjugate-Gradient Method

- Set $x_0 = 0$, $r_0 = b - Ax_0$, $p_{-1} = 0$, $\beta_0 = 0$ and specify the convergence tolerance $\epsilon$.

- Repeat for $i = 0, 1, \ldots$:
  - If $\|r_i\| < \epsilon$, terminate.
  - If $i > 0$ calculate $\beta_i = r_i^T r_{i-1} r_{i-1}^-1$.
  - Calculate the next conjugate vector $p_i = r_i + \beta_i p_{i-1}$.
  - Calculate the corresponding step length $\alpha_i = r_i^T r_i/p_i^T A p_i$.
  - Update the point $x_{i+1} = x_i + \alpha_i p_i$.
  - Update the residual $r_{i+1} = r_i - \alpha_i A p_i$.

- Except for the product $A p_i$, only vector operations (time complexity $O(n)$) are required and only vectors need to be stored.

---

### Convergence and pre-conditioning

- In exact arithmetic, the conjugate-gradient methods converges in $k$ iterations, where $k$ is the number of distinct eigenvalues of $A$ ($k \leq n$).

- In finite arithmetic, accumulated round-off errors will cause deviation from the orthogonality/conjugate properties, and will require more iterations.

- The conjugate-gradient method converges linearly with convergence constant

$$C = \frac{\|x_{i+1} - x_i\|_A}{\|x_i - x_{i-1}\|_A} \leq \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1},$$

where the error norm is defined by $\|y\|_A^2 = \frac{1}{2} y^T A y$.

- This may cause slow convergence if $\kappa(A)$ is large, which is not unusual.
Pre-conditioning

- In practice, the method is often augmented with a pre-conditioning step, which means that the equation system
  \[ Ax = b \]
  is replaced by the equivalent equation system
  \[ M^{-1}Ax = M^{-1}b, \]
  where the matrix \( M \) is called a pre-conditioner.

Algorithm for the Pre-Conditioned Conjugate-Gradient method

- Set \( x_0 = 0, r_0 = b - Ax_0, p_0 = 0, \beta_0 = 0 \) and specify the convergence tolerance \( \epsilon \).
- Repeat for \( i = 0, 1, \ldots \)
  - If \( \|r_i\| < \epsilon \), terminate.
  - Solve the pre-conditioner equation \( z_i = M^{-1}r_i \).
  - If \( i > 0 \) calculate \( \beta_i = r^T_i z_i / r^T_{i-1} z_{i-1} \).
  - Calculate the next conjugate vector \( p_i = z_i + \beta_i p_{i-1} \).
  - Calculate the step length \( \alpha_i = r^T_i z_i / p^T_i Ap_i \).
  - Update the point \( x_{i+1} = x_i + \alpha_i p_i \).
  - Update the residual \( r_{i+1} = r_i - \alpha_i Ap_i \).

Pre-conditioning

- The matrix \( M \) is introduced to speed up the convergence, which may be accomplished by reducing \( \kappa(M^{-1}A) \) or reducing the number of distinct eigenvalues of \( M^{-1}A \).
- The optimal solution is \( M = A \Rightarrow M^{-1}A = I \), but that would mean we had to solve the original equation \( Ax = b \) first!
- Thus, the pre-conditioner \( M \) should be similar to \( A \) but be easier to invert.
- Common pre-conditioners are diagonal matrices or (if \( A \) is sparse) a factorization where the fill-in is ignored (incomplete Cholesky), i.e. if
  \[ LL^T = A, \]
  then
  \[ L_0 L_0^T = A, \]
  where \( L_0 \) has the same non-zero pattern as \( A \).

Truncated-Newton methods

- Truncated-Newton methods work on the problem
  \[ \min f(x). \]
- They compute a search direction by finding an approximative solution of the Newton equation
  \[ \nabla^2 f(x_k) p \approx -\nabla f(x_k) \]
  using an iterative method, usually the conjugated-gradient method.
- The iterative method is halted (truncated) before the exact solution is found, hence the name.
- Given the approximate search direction \( p_k \), a line search is performed in the usual way.
Advantages of the Truncated-Newton method

- The Conjugate-Gradient algorithm only requires vector operations except for the product $Av = \nabla^2 f(x_k)v$.
- If the product $Av$ can be calculated cheaply with respect to time and storage for all $v$, this “minimal” change of the Newton method makes it suitable for large problems.
- This is satisfied if e.g. $A$ is sparse (large problems are often sparse), and/or $A$ has a structure such that the elements $a_{ij}$ can be calculated as a function of $i$ and $j$.
- Otherwise, the product $\nabla^2 f(x_k)v$ may be approximated by
  $\nabla^2 f(x_k)v \approx \frac{\nabla f(x_k + hv) - \nabla f(x_k)}{h}$,
for some small $h$, i.e. a difference approximation only requiring vector operations.

Trade-offs

- If we spend a high number of “inner” iterations to approximately solve the Newton equation, the search direction should be “better” and we should need fewer “outer” iterations to find the solution.
- To make the relation explicit, terminate the inner iteration when
  $\|\nabla^2 f(x_k)v + \nabla f(x_k)\| \leq \phi_k \|\nabla f(x_k)\|$, for some tolerance $\phi_k \geq 0$.
- If $\phi_k = c < 1$, the Truncated-Newton algorithm will converge linearly with convergence constant $c$.
- If $\phi_k \to 0$, it will converge super-linearly, and $\phi_k \leq \|\nabla f(x_k)\|$ will result in quadratic convergence.
- The special case $\phi_k = 0$ produces the Newton method proper.

Trade-offs

- Faster “outer” converge will require more “inner” iterations, which means that there is a need to strike a balance between the two to obtains as fast convergence as possible.
- Furthermore, a more costly pre-conditioner, e.g. an incomplete Cholesky instead of a diagonal pre-conditioner, may lead to faster total convergence.
- For large problems it often pays off to spend time to adjust the algorithms since the computation time may otherwise be very large.