Lecture Topic: A Revision
The Basics

- Problem and Instance
- Turing Machine
- Integers in a Turing Machine
- Floating-Point Numbers
- BSS Machine
- Computability and Complexity
The Tools

- The spectrum
- Matrix norms
- Solvers for linear systems
- Solvers for least squares
- Solvers for eigenvalue problems
- Necessary optimality conditions
- Solvers for sub-differentiable opt.
- Solvers for differentiable opt.
The Concepts

- The Condition
- The Witness
- The Convergence
- The Rate of Convergence
- The Stability
Problem and Instance

Traditional computer science begins with a finite alphabet.

By stringing elements of the alphabet one after another, one obtains strings.

A set of strings is called a language.

A decision problem is defined by a fixed set $S$, which is a subset of the language $U$ of all possible strings over the alphabet $A$.

A particular instance of the decision problem is to decide, given an element $u \in U$, whether $u$ is included in $S$. 
Problem and Instance

*Traditional computer science* begins with a finite alphabet.

By stringing elements of the alphabet one after another, one obtains strings.

A set of strings is called a language.

A decision problem is defined by a fixed set $S$, which is a subset of the language $U$ of all possible strings over the alphabet $A$.

A particular instance of the decision problem is to decide, given an element $u \in U$, whether $u$ is included in $S$. 
Traditional computer science begins with a finite alphabet. By stringing elements of the alphabet one after another, one obtains strings. A set of strings is called a language.

A decision problem is defined by a fixed set $S$, which is a subset of the language $U$ of all possible strings over the alphabet $A$.

A particular instance of the decision problem is to decide, given an element $u \in U$, whether $u$ is included in $S$. 
Problem and Instance

*Traditional computer science* begins with a finite alphabet.

By stringing elements of the alphabet one after another, one obtains strings.

A set of strings is called a language.

A decision problem is defined by a fixed set $S$, which is a subset of the language $U$ of all possible strings over the alphabet $A$.

A particular instance of the decision problem is to decide, given an element $u \in U$, whether $u$ is included in $S$. 
Problem and Instance

*Traditional computer science* begins with a finite alphabet. By stringing elements of the alphabet one after another, one obtains strings. A set of strings is called a language.

A decision problem is defined by a fixed set $S$, which is a subset of the language $U$ of all possible strings over the alphabet $A$.

A particular instance of the decision problem is to decide, given an element $u \in U$, whether $u$ is included in $S$. 
Turing Machine

A Turing machine is defined by:

- a finite, non-empty set $Q$ of objects, representing states
- a subset $F$ of $Q$, corresponding to “accepting” states, where computation halts
- $q_0 \in Q$, the initial state
- a finite, non-empty set $\Gamma$ of objects, representing the symbols to be used on a tape
- a partial function $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, 1\}$ where for a combination of a state and symbol read from the tape, we get the next state, the symbol to write onto the tape, and an instruction to shift the tape left (-1), right (+1), or keep in its position (0).
Integers

*Numbers on a Turing machine:* Modern computers can be seen as Turing machines over the alphabet of \( \{0, 1\} \). Computer memory can be seen as a tape partitioned into *words* of \( w \) bits each.

The \( w \) bits represent \( 2^w \) distinct integers, using some mapping from the \( w \)-tuple to the integers, such as

*two’s complement*, where an integer \(-2^{w-1} \leq x \leq 2^{w-1} - 1\) is encoded as \( x \) if \( x \geq 0 \) and \( x + 2^w \) if \( x < 0 \).
Integers

*Numbers on a Turing machine:* Modern computers can be seen as Turing machines over the alphabet of \( \{0, 1\} \). Computer memory can be seen as as a tape partitioned into *words* of \( w \) bits each.

The \( w \) bits represent \( 2^w \) distinct integers, using some mapping from the \( w \)-tuple to the integers, such as

*two’s complement*, where an integer \(-2^{w-1} \leq x \leq 2^{w-1} - 1\) is encoded as \( x \) if \( x \geq 0 \) and \( x + 2^w \) if \( x < 0 \).
Integers

*Numbers on a Turing machine:* Modern computers can be seen as Turing machines over the alphabet of $\{0, 1\}$. Computer memory can be seen as as a tape partitioned into *words* of $w$ bits each.

The $w$ bits represent $2^w$ distinct integers, using some mapping from the $w$-tuple to the integers, such as

*two’s complement*, where an integer $-2^{w-1} \leq x \leq 2^{w-1} - 1$ is encoded as $x$ if $x \geq 0$ and $x + 2^w$ if $x < 0$. 
Floating-Point Numbers

A floating point number system represents some the real numbers \( x \in \mathbb{R} \) as

\[
x = \pm.m \times b^e = \pm \left( \frac{m_1}{b^1} + \frac{m_2}{b^2} + \cdots + \frac{m_t}{b^t} \right) \times b^e
\]

\[
= \pm.(m_1 m_2 \ldots m_t)_b \times b^e,
\]

where \( m \) is called the mantissa, \( e \) is called the exponent, \( t \) is the precision or number of digits in the mantissa, and \( b \) is the base.

Bias is added to the exponent, to make it non-negative.

When the actual exponent is \( e_{\min} \leq e \leq e_{\max} \) and the mantissa is \( m_{\min} \leq m \leq m_{\max} \), the representable numbers \( x \) are:

\[
m_{\min} \times b^{e_{\min}} < |x| < m_{\max} \times b^{e_{\max}}, \quad \text{(2.1)}
\]

\[
b^{e_{\min}-1} \leq |x| \leq (1 - b^{-t})b^{e_{\max}} \approx b^{e_{\max}}. \quad \text{(2.2)}
\]

\[
\varepsilon_m = (00 \ldots 1)_b \times b^1 = b^{-t} \times b^1 = b^{1-t}.
\]
Floating-Point Numbers

A floating point number system represents some of the real numbers $x \in \mathbb{R}$ as

$$x = \pm . m \times b^e = \pm \left( \frac{m_1}{b^1} + \frac{m_2}{b^2} + \cdots + \frac{m_t}{b^t} \right) \times b^e$$

$$= \pm . (m_1 m_2 \ldots m_t)_b \times b^e,$$

where $m$ is called the mantissa, $e$ is called the exponent, $t$ is the precision or number of digits in the mantissa, and $b$ is the base.

Bias is added to the exponent, to make it non-negative.

When the actual exponent is $e_{\text{min}} \leq e \leq e_{\text{max}}$ and the mantissa is $m_{\text{min}} \leq m \leq m_{\text{max}}$, the representable numbers $x$ are:

$$m_{\text{min}} \times b^{e_{\text{min}}} < |x| < m_{\text{max}} \times b^{e_{\text{max}}}, \quad (2.1)$$

$$b^{e_{\text{min}}-1} \leq |x| \leq (1 - b^{-t}) b^{e_{\text{max}}} \approx b^{e_{\text{max}}}. \quad (2.2)$$

$$\varepsilon_m = .(00\ldots1)_b \times b^1 = b^{-t} \times b^1 = b^{1-t}.$$
Floating-Point Numbers

A floating point number system represents some the real numbers $x \in \mathbb{R}$ as

$$x = \pm.m \times b^e = \pm \left( \frac{m_1}{b^1} + \frac{m_2}{b^2} + \cdots + \frac{m_t}{b^t} \right) \times b^e$$

$$= \pm.(m_1m_2\ldots m_t)_b \times b^e,$$

where $m$ is called the mantissa, $e$ is called the exponent, $t$ is the precision or number of digits in the mantissa, and $b$ is the base.

Bias is added to the exponent, to make it non-negative.

When the actual exponent is $e_{\text{min}} \leq e \leq e_{\text{max}}$ and the mantissa is $m_{\text{min}} \leq m \leq m_{\text{max}}$, the representable numbers $x$ are:

$$m_{\text{min}} \times b^{e_{\text{min}}} < |x| < m_{\text{max}} \times b^{e_{\text{max}}}, \quad (2.1)$$

$$b^{e_{\text{min}}-1} \leq |x| \leq (1 - b^{-t})b^{e_{\text{max}}} \approx b^{e_{\text{max}}} . \quad (2.2)$$

$$\varepsilon_m = .(00\ldots1)_b \times b^1 = b^{-t} \times b^1 = b^{1-t}.$$
Floating-Point Numbers

A floating point number system represents some the real numbers $x \in \mathbb{R}$ as

$$x = \pm m \times b^e = \pm \left( \frac{m_1}{b^1} + \frac{m_2}{b^2} + \cdots + \frac{m_t}{b^t} \right) \times b^e$$

$$= \pm (m_1 m_2 \ldots m_t)_b \times b^e,$$

where $m$ is called the mantissa, $e$ is called the exponent, $t$ is the precision or number of digits in the mantissa, and $b$ is the base.

Bias is added to the exponent, to make it non-negative.

When the actual exponent is $e_{\text{min}} \leq e \leq e_{\text{max}}$ and the mantissa is $m_{\text{min}} \leq m \leq m_{\text{max}}$, the representable numbers $x$ are:

$$m_{\text{min}} \times b^{e_{\text{min}}} < |x| < m_{\text{max}} \times b^{e_{\text{max}}},$$

$$b^{e_{\text{min}}-1} \leq |x| \leq (1 - b^{-t})b^{e_{\text{max}}} \approx b^{e_{\text{max}}}. \quad (2.1)$$

$$\varepsilon_m = .(00 \ldots 1)_b \times b^1 = b^{-t} \times b^1 = b^{1-t}. \quad (2.2)$$
The three values \((\pm, m, e)\) are stored in a \(w\)-bit word.

IEEE standard 745 defines 8-byte *Double* with \(53 = 52 + 1\) significant bits mantissa and 11-bit exponent.

Double has the unit round-off of \(u_r = 2^{-53} \approx 1.11 \times 10^{-16}\), the smallest representable number \(2^{-1022} \approx 2.82 \times 10^{-308}\), and the largest representable number \((2 - 2^{-52}) \times 2^{1023} \approx 1.41 \times 10^{308}\).
The three values \((\pm, m, e)\) are stored in a \(w\)-bit word.

IEEE standard 745 defines 8-byte \textit{Double} with \(53 = 52 + 1\) significant bits mantissa and 11-bit exponent.

Double has the unit round-off of \(u_r = 2^{-53} \approx 1.11 \times 10^{-16}\),
the smallest representable number \(2^{-1022} \approx 2.82 \times 10^{-308}\), and
the largest representable number \((2 - 2^{-52}) \times 2^{1023} \approx 1.41 \times 10^{308}\).
IEEE 745

The three values $(\pm, m, e)$ are stored in a $w$-bit word.

IEEE standard 745 defines 8-byte Double with $53 = 52 + 1$ significant bits mantissa and 11-bit exponent.

Double has the unit round-off of $u_r = 2^{-53} \approx 1.11 \times 10^{-16}$,

the smallest representable number $2^{-1022} \approx 2.82 \times 10^{-308}$, and

the largest representable number $(2 - 2^{-52}) \times 2^{1023} \approx 1.41 \times 10^{308}$. 
The three values ($\pm$, $m$, $e$) are stored in a $w$-bit word.

IEEE standard 745 defines 8-byte *Double* with $53 = 52 + 1$ significant bits mantissa and 11-bit exponent.

*Double* has the unit round-off of $u_r = 2^{-53} \approx 1.11 \times 10^{-16}$,

the smallest representable number $2^{-1022} \approx 2.82 \times 10^{-308}$, and

the largest representable number $(2 - 2^{-52}) \times 2^{1023} \approx 1.41 \times 10^{308}$.
The three values ($\pm$, $m$, $e$) are stored in a $w$-bit word.

IEEE standard 745 defines 8-byte *Double* with $53 = 52 + 1$ significant bits mantissa and 11-bit exponent.

Double has the unit round-off of $u_r = 2^{-53} \approx 1.11 \times 10^{-16}$,

the smallest representable number $2^{-1022} \approx 2.82 \times 10^{-308}$, and

the largest representable number $(2 - 2^{-52}) \times 2^{1023} \approx 1.41 \times 10^{308}$. 

Big O Notation

To describe the behaviour of an algorithm on instance of size $n$, which requires a finite number $T(n)$ operations of the Turing machine in the worst possible case, we say $T(n)$ is $O(g(n))$ if $|T(n)/g(n)|$ is bounded from above as $n \to \infty$.

On the other hand, $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

Further, $f(n) = \Theta(g(n))$ whenever $f(n) = O(g(n))$ and $g(n) = O(f(n))$. 
Big O Notation

To describe the behaviour of an algorithm on instance of size $n$, which requires a finite number $T(n)$ operations of the Turing machine in the worst possible case, we say $T(n)$ is $O(g(n))$ if $|T(n)/g(n)|$ is bounded from above as $n \to \infty$.

On the other hand, $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

Further, $f(n) = \Theta(g(n))$ whenever $f(n) = O(g(n))$ and $g(n) = O(f(n))$. 
Big O Notation

To describe the behaviour of an algorithm on instance of size \( n \), which requires a finite number \( T(n) \) operations of the Turing machine in the worst possible case, we say \( T(n) \) is \( O(g(n)) \) if \( |T(n)/g(n)| \) is bounded from above as \( n \to \infty \).

On the other hand, \( f(n) = \Omega(g(n)) \) if \( g(n) = O(f(n)) \).

Further, \( f(n) = \Theta(g(n)) \) whenever \( f(n) = O(g(n)) \) and \( g(n) = O(f(n)) \).
Big O Notation

To describe the behaviour of an algorithm on instance of size $n$, which requires a finite number $T(n)$ operations of the Turing machine in the worst possible case, we say $T(n)$ is $O(g(n))$ if $|T(n)/g(n)|$ is bounded from above as $n \to \infty$.

On the other hand, $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

Further, $f(n) = \Theta(g(n))$ whenever $f(n) = O(g(n))$ and $g(n) = O(f(n))$. 
Real Computation

In contrast, *real computation* defines a decision problem using a fixed set $S \subseteq R^\infty$, where

$R^\infty$ is a direct sum of a possibly infinite number of sets $R$, e.g. real numbers.

A particular instance of the decision problem is given by vector $v \in R^\infty$ and asks whether $v$ is included in $S$. 
Real Computation

In contrast, real computation defines a decision problem using a fixed set $S \subseteq \mathbb{R}^\infty$, where

$\mathbb{R}^\infty$ is a direct sum of a possibly infinite number of sets $\mathbb{R}$, e.g. real numbers.

A particular instance of the decision problem is given by vector $v \in \mathbb{R}^\infty$ and asks whether $v$ is included in $S$. 
In contrast, \textit{real computation} defines a decision problem using a fixed set $S \subseteq \mathbb{R}^{\infty}$, where $\mathbb{R}^{\infty}$ is a direct sum of a possibly infinite number of sets $R$, e.g. real numbers.

A particular instance of the decision problem is given by vector $v \in \mathbb{R}^{\infty}$ and asks whether $v$ is included in $S$. 
BSS Machine

Subsequently, a BSS machine defined by:

- a set of \( r = i + o \) registers, where the vector in \( R^r \) is partitioned into input and output registers
- an input space \( X \subseteq R^i \), with an associated mapping from \( X \) to input registers
- an output space \( Y \subseteq R^o \), with an associated mapping from \( Y \) to output registers
- a “flow chart” \( F = (Q, E) \), where \( Q \) is a set of nodes, which includes initial node “1”, and \( E \) is a set of oriented edges. Each node \( q \in Q \) can be of the following types: assignment of a constant to a register, copying of a register to another register, addition, subtraction, multiplication, division (and, in some definitions, taking \( i \)th square roots) between values of two registers, with the output directed to a fixed register (0), and branching on the comparison of two registers.
BSS vs. Turing

A BSS machine makes it possible to compute (P over reals) or verify (NP over reals) results, which are impossible to compute (P in TM) or verify (NP in TM) using the Turing machine (TM).

On the other hand, Abel’s theorem suggests that roots of polynomials and eigenvalues remain impossible to compute exactly even on the BSS machine. There, one can analyse the rate of convergence on the BSS machine, before worrying about stability.
BSS vs. Turing

A BSS machine makes it possible to compute (P over reals) or verify (NP over reals) results, which are impossible to compute (P in TM) or verify (NP in TM) using the Turing machine (TM).

On the other hand, Abel’s theorem suggests that roots of polynomials and eigenvalues remain impossible to compute exactly even on the BSS machine.

There, one can analyse the rate of convergence on the BSS machine, before worrying about stability.
A BSS machine makes it possible to compute (P over reals) or verify (NP over reals) results, which are impossible to compute (P in TM) or verify (NP in TM) using the Turing machine (TM).

On the other hand, Abel’s theorem suggests that roots of polynomials and eigenvalues remain impossible to compute exactly even on the BSS machine. There, one can analyse the rate of convergence on the BSS machine, before worrying about stability.
Example Problems

Linear least squares \( \arg\min_{x \in \mathbb{R}^n} \|Ax - b\|_2, \)

\( \ell_1 \)-regularised least squares \( \arg\min_{x \in \mathbb{R}^n} \|Ax - b\|_2 + \lambda \|x\|_1, \)

\( \ell_0 \)-regularised least squares \( \arg\min_{x \in \mathbb{R}^n} \|Ax - b\|_2 + \lambda \|x\|_0, \)

Principal component analysis \( \arg\max_{x \in \mathbb{R}^n} \|Ax\|_2 \quad \text{s.t.} \quad \|x\|_2 \leq 1, \)

Low-rank approximation \( \arg\min_{B \in \mathbb{R}^{m \times n}} \|A - B\|_F \quad \text{s.t.} \quad \text{rank}(B) \leq k \)

Nuclear norm minimisation \( \arg\min_{M \in \mathbb{R}^{m \times n}} \|M\|_* \quad \text{s.t.} \quad M_{i,j} = A_{i,j} \quad \forall \ (i,j) \in E, \)

Rank minimisation \( \arg\min_{X \in \mathbb{R}^{m \times r}, Y \in \mathbb{R}^{n \times r}} \sum_{(i,j) \in E} \left((XY^T)_{i,j} - A_{i,j}\right)^2, \)
Questions to Ask

There are three important questions to ask:

- Is the (Lagrangian) function (sub-)differentiable?
- Is the (Lagrangian) function twice differentiable?
- Is the (Lagrangian) function convex?

We have seen examples of a variety of combinations of those properties.
### The Example Answers

<table>
<thead>
<tr>
<th>Problem</th>
<th>Sub-derivative</th>
<th>$1^{st}$ P.D.</th>
<th>$2^{nd}$ P.D.</th>
<th>Convex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear least squares</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\ell_1$-regularised least squares</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>$\ell_0$-regularised least squares</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Principal component analysis</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Low-rank approximation</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nuclear norm minimisation</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Rank minimisation</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Training SVMs</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
The Tools

- The spectrum
- Matrix norms
- Solvers for linear systems
- Solvers for least squares
- Solvers for eigenvalue problems
- Necessary optimality conditions
- Solvers for sub-differentiable opt.
- Solvers for differentiable opt.
The Spectrum

The spectrum: for a given $A \in \mathbb{R}^{n \times n}$, eigenvector $x \in \mathbb{R}^n$ and eigenvalue $\lambda \in \mathbb{R}$ are unknowns in the system of equations $Ax = \lambda x$. For $A \in \mathbb{R}^{m \times n}$, a singular value $\sigma \in \mathbb{R}, \sigma \geq 0$ and two non-zero singular vectors $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$ are the unknowns in $Av = \sigma u$. For, $0 < r = \text{rank}(A)$, we denote the non-zero singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$, or collectively, $\Sigma$. 
Matrix Norms

Matrix norms include:

Nuclear norm $\|A\|_* := \text{trace} \left( \sqrt{A^T A} \right) = \sum_{i=1}^{\min\{m, n\}} \sigma_i$.  

(4.1)

Frobenius norm $\|A\|_F := \sqrt{\text{trace}(A^T A)} = \left( \sum_{i=1}^{k} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2} = \sqrt{\min\{m, n\} \sum_{i=1}^{\min\{m, n\}} \sigma_i^2}$.  

(4.2)

Spectral norm $\|A\|_2 := \sqrt{\lambda_{\max}(A^* A)} = \sigma_{\max}(A)$  

(4.3)

where $\sqrt{A^* A}$ denotes a positive semidefinite $B$ such that $B = A^T A$. Generally, the use of an $\ell_0$-norm or rank as a regularising term in the objective is very desirable, from the statistical point of view, with $\ell_1$ and nuclear norm being very good approximations. $\ell_0$-norm and rank make problems non-convex, while $\ell_1$ and nuclear norm make problems non-smooth.
Solvers for Linear Systems

Gauss-Jordan can be thought of in terms of the decomposition $A = LDU^T$ where $L$ and $U^T$ are unit lower and unit upper triangular and $D = \text{diag}(u_{ii})$ is a diagonal matrix with $u_{11}, \ldots, u_{nn}$ on the diagonal.

**QRD:** takes $m \times n$ matrix $A$, $m \geq n$ and produces an $m \times m$ orthogonal matrix, i.e., $Q^TQ = QQ^T = I$, and an $m \times n$ upper-triangular matrix $R$.

**SVD:** takes an $m \times n$ matrix $A$ and produces an $m \times m$ orthogonal matrix $U$, an $m \times n$ matrix $\Sigma$ with $\Sigma_{i,i} \geq 0$ being the “singular values” of $A$, and an $n \times n$ orthogonal matrix $V^T$. 
Solvers for Linear Systems

Gauss-Jordan can be thought of in terms of the decomposition $A = LDU^T$ where $L$ and $U^T$ are unit lower and unit upper triangular and $D = \text{diag}(u_{ii})$ is a diagonal matrix with $u_{11}, \ldots, u_{nn}$ on the diagonal.

**QRD:** takes $m \times n$ matrix $A$, $m \geq n$ and produces an $m \times m$ orthogonal matrix, i.e., $Q^T Q = QQ^T = I$, and an $m \times n$ upper-triangular matrix $R$

**SVD:** takes an $m \times n$ matrix $A$ and produces an $m \times m$ orthogonal matrix $U$, an $m \times n$ matrix $\Sigma$ with $\Sigma_{i,i} \geq 0$ being the “singular values” of $A$, and an $n \times n$ orthogonal matrix $V^T$. 
Solvers for Linear Systems

Gauss-Jordan can be thought of in terms of the decomposition $A = LDU^T$ where $L$ and $U^T$ are unit lower and unit upper triangular and $D = \text{diag}(u_{ii})$ is a diagonal matrix with $u_{11}, \ldots, u_{nn}$ on the diagonal.

**QRD**: takes $m \times n$ matrix $A$, $m \geq n$ and produces an $m \times m$ orthogonal matrix, i.e., $Q^T Q = QQ^T = I$, and an $m \times n$ upper-triangular matrix $R$.

**SVD**: takes an $m \times n$ matrix $A$ and produces an $m \times m$ orthogonal matrix $U$, an $m \times n$ matrix $\Sigma$ with $\Sigma_{i,i} \geq 0$ being the “singular values” of $A$, and an $n \times n$ orthogonal matrix $V^T$. 
Solvers for Least Squares

The Moore–Penrose pseudo-inverse $A^\dagger$ is a generalisation of the inverse $A^{-1}A = I$ that satisfies $AA^\dagger A = A$, $A^\dagger AA^\dagger = A^\dagger$, $(AA^\dagger)^T = AA^\dagger$, and $(A^\dagger A)^T = A^\dagger A$. 
Power method for computing the greatest eigenvalue by absolute value, is based on
\[ x^{k+1} = \frac{Ax^k}{\|Ax^k\|}. \]
Alternatively, one can use QR or SVD decompositions.
Solvers for Eigenvalue Problems

Power method for computing the greatest eigenvalue by absolute value, is based on

\[ x^{k+1} = \frac{Ax^k}{\|Ax^k\|}. \]

Alternatively, one can use QR or SVD decompositions.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function \( f(x) \) one has \( \nabla f(x) = 0 \) ("first-order conditions").

For a twice-differentiable 1D \( f \), all local extrema satisfy \( \nabla^2 f(x) \neq 0 \) ("second-order conditions").

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function $f(x)$ one has $\nabla f(x) = 0$ ("first-order conditions").

For a twice-differentiable 1D $f$, all local extrema satisfy $\nabla^2 f(x) \neq 0$ ("second-order conditions").

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function $f(x)$ one has $\nabla f(x) = 0$ (“first-order conditions”).

For a twice-differentiable 1D $f$, all local extrema satisfy $\nabla^2 f(x) \neq 0$ (“second-order conditions”).

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function $f(x)$ one has $\nabla f(x) = 0$ (“first-order conditions”).

For a twice-differentiable 1D $f$, all local extrema satisfy $\nabla^2 f(x) \neq 0$ (“second-order conditions”).

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function \( f(x) \) one has \( \nabla f(x) = 0 \) ("first-order conditions").

For a twice-differentiable 1D \( f \), all local extrema satisfy \( \nabla^2 f(x) \neq 0 \) ("second-order conditions").

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Optimisation

The study of optimisation problems began as the study of necessary conditions for optimality.

Fermat’s theorem suggests that for all local extrema of differentiable 1D function \( f(x) \) one has \( \nabla f(x) = 0 \) ("first-order conditions").

For a twice-differentiable 1D \( f \), all local extrema satisfy \( \nabla^2 f(x) \neq 0 \) ("second-order conditions").

For a smooth function, one can derive higher-order necessary conditions from the Taylor expansion.

Notice, however, that a critical point need not be a local optimum, the less the global optimum.

We have also seen optimality conditions for subdifferentiable functions and constrained optimisation.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a subderivative of a function $f : \mathbb{R} \to \mathbb{R}$ at a point $y \in \mathbb{R}$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The subdifferential of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a subderivative of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point $y \in \mathbb{R}$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The subdifferential of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a subderivative of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point $y \in R$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The subdifferential of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a subderivative of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point $y \in \mathbb{R}$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The subdifferential of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a *subderivative* of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point $y \in \mathbb{R}$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The *subdifferential* of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Non-differentiable Optimisation

Without derivatives, one may either resort to derivative-free optimisation methods, or can consider subgradient methods.

Derivative-free optimisation discretises the set of possible inputs and check the outputs at the chosen points.

Subgradients methods are based on the notion of a subderivative of a function $f : \mathbb{R} \to \mathbb{R}$ at a point $y \in R$ is a real number $c$ such that $f(x) - f(y) \geq c(x - y)$ for all $x \in \mathbb{R}$.

The subdifferential of the function $f$ at $y$ is a set of all subderivatives of $f$ at $y$.

For convex subdifferentiable problems, one often uses coordinate descent techniques, where each iteration $k$ is associated with coordinate $1 \leq i \leq n$ and one updates only the $i$th coordinate.

For example, both sparse least squares and training of sparse SVMs are non-differentiable.
Differentiable Optimisation

Whenever the problem is (once) differentiable, one can apply gradient methods.

Whenever the problem is twice-differentiable, one can apply methods, which extend the *Newton* method for 1D optimisation, i.e., \( x^{k+1} = x^k - \frac{f'(x^k)}{f''(x^k)} \).

Such methods apply methods for 1D root finding \( x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} \) applied to an \( f(x) \) obtained from first-order optimality conditions, and are hence called second-order.
Differentiable Optimisation

Whenever the problem is (once) differentiable, one can apply gradient methods. Whenever the problem is twice-differentiable, one can apply methods, which extend the Newton method for 1D optimisation, i.e., $x^{k+1} = x^k - \frac{f'(x^k)}{f''(x^k)}$.

Such methods apply methods for 1D root finding $x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$ applied to an $f(x)$ obtained from first-order optimality conditions, and are hence called second-order.
Differentiable Optimisation

Whenever the problem is (once) differentiable, one can apply gradient methods.
Whenever the problem is twice-differentiable, one can apply methods, which extend the Newton method for 1D optimisation, i.e., $x^{k+1} = x^k - \frac{f'(x^k)}{f''(x^k)}$.

Such methods apply methods for 1D root finding $x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$ applied to an $f(x)$ obtained from first-order optimality conditions, and are hence called second-order.
Differentiable Optimisation

Whenever the problem is (once) differentiable, one can apply gradient methods. Whenever the problem is twice-differentiable, one can apply methods, which extend the Newton method for 1D optimisation, i.e., \( x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \).

Such methods apply methods for 1D root finding \( x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \) applied to an \( f(x) \) obtained from first-order optimality conditions, and are hence called second-order.
The Concepts

- The Condition
- The Witness
- The Convergence
- The Rate of Convergence
- The Stability
The Condition

The *condition* of an instance $p$ is a measure of the change in the solution $s$ in response to a change in $p$.

One performs perturbation analysis to derive a *relative condition* number:

$$\text{cond}_{\text{rel}}(p) = \max_{\Delta p \neq 0} \frac{\| \Delta s \|}{\| \Delta p \|} \frac{\| s \|}{\| p \|}.$$ 

For solving linear systems with a matrix $A$, the relative condition number and spectral condition number are:

$$1 \leq \frac{\max_{\lambda \in \Sigma(A)} |\lambda|}{\min_{\lambda \in \Sigma(A)} |\lambda|} \leq \| A \| \| A^{-1} \|$$

for any matrix norm $\| \|$.
The Condition

The *condition* of an instance $p$ is a measure of the change in the solution $s$ in response to a change in $p$.

One performs perturbation analysis to derive a *relative condition* number:

$$\text{cond}_{\text{rel}}(p) = \max_{\Delta p \neq 0} \frac{\|\Delta s\|/\|s\|}{\|\Delta p\|/\|p\|}.$$  

For solving linear systems with a matrix $A$, the relative condition number and spectral condition number are:

$$1 \leq \frac{\max_{\lambda \in \Sigma(A)} |\lambda|}{\min_{\lambda \in \Sigma(A)} |\lambda|} \leq \|A\|\|A^{-1}\|$$

for any matrix norm $\|\|$.
The Condition

The *condition* of an instance $p$ is a measure of the change in the solution $s$ in response to a change in $p$.

One performs perturbation analysis to derive a *relative condition* number:

$$\text{cond}_{\text{rel}}(p) = \max_{\Delta p \neq 0} \frac{\|\Delta s\|/\|s\|}{\|\Delta p\|/\|p\|}.$$

For solving linear systems with a matrix $A$, the relative condition number and spectral condition number are:

$$1 \leq \frac{\max_{\lambda \in \Sigma(A)} |\lambda|}{\min_{\lambda \in \Sigma(A)} |\lambda|} \leq \|A\|\|A^{-1}\|$$

for any matrix norm $\|\|$.
The Condition

For eigenvalue problems, the condition numbers are the same as for linear systems.

For least squares, the condition number is closely related:

\[ \text{cond}(A) = \|A\|_2 \|A^\dagger\|_2 = \sigma_1 \sigma_r. \]

For generic optimisation problems, condition numbers also exist, but are more involved.

As a rough guide, base-10 logarithm of the condition number above tells you the number of digits of accuracy you loose.

Often, the condition number of a well-posed problem (e.g., non-singular A) is inversely related to the distance to the nearest ill-posed problem (e.g., singular A).
The Condition

For eigenvalue problems, the condition numbers are the same as for linear systems.

For least squares, the condition number is closely related:

$$\text{cond}(A) = \|A\|_2 \|A^\dagger\|_2 = \sigma_1 \sigma_r.$$ 

For generic optimisation problems, condition numbers also exist, but are more involved.

As a rough guide, base-10 logarithm of the condition number above tells you the number of digits of accuracy you lose.

Often, the condition number of a well-posed problem (e.g., non-singular $A$) is inversely related to the distance to the nearest ill-posed problem (e.g., singular $A$).
The Condition

For eigenvalue problems, the condition numbers are the same as for linear systems.

For least squares, the condition number is closely related:

\[ \text{cond}(A) = \|A\|_2 \|A^\dagger\|_2 = \sigma_1 \sigma_r. \]

For generic optimisation problems, condition numbers also exist, but are more involved.

As a rough guide, base-10 logarithm of the condition number above tells you the number of digits of accuracy you lose.

Often, the condition number of a well-posed problem (e.g., non-singular \( A \)) is inversely related to the distance to the nearest ill-posed problem (e.g., singular \( A \)).
The Condition

For eigenvalue problems, the condition numbers are the same as for linear systems.

For least squares, the condition number is closely related:

\[
\text{cond}(A) = \|A\|_2 \|A^\dagger\|_2 = \sigma_1 \sigma_r.
\]

For generic optimisation problems, condition numbers also exist, but are more involved.

As a rough guide, base-10 logarithm of the condition number above tells you the number of digits of accuracy you lose.

Often, the condition number of a well-posed problem (e.g., non-singular \(A\)) is inversely related to the distance to the nearest ill-posed problem (e.g., singular \(A\)).
The Condition

For eigenvalue problems, the condition numbers are the same as for linear systems.

For least squares, the condition number is closely related:

\[
\text{cond}(A) = \|A\|_2 \|A^\dagger\|_2 = \sigma_1 \sigma_r.
\]

For generic optimisation problems, condition numbers also exist, but are more involved.

As a rough guide, base-10 logarithm of the condition number above tells you the number of digits of accuracy you loose.

Often, the condition number of a well-posed problem (e.g., non-singular \( A \)) is inversely related to the distance to the nearest ill-posed problem (e.g., singular \( A \)).
The Witness

Recall that \textit{local minimum} extends the notion in 1D optimisation, e.g., \(x^* \in \mathbb{R}\) such that \(f(x^*) \leq f(x)\) for all \(x \in [x^* - \epsilon, x^* + \epsilon]\).

\textit{Global minimum} extends the notion in 1D optimisation, i.e., \(x^* \in \mathbb{R}\) such that \(f(x^*) \leq f(x)\) for all \(x \in \mathbb{R}\).

In optimisation, checking whether a point \(x^* \in \mathbb{R}\) satisfies first-order optimality conditions is often much easier than checking whether \(x^*\) is a global minimum. A witness for global optimality can be obtained from Lagrangian duality,

\[
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \quad \text{("weak duality"), or via problem-specific means.}
\]

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that *local minimum* extends the notion in 1D optimisation, e.g., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in [x^* - \epsilon, x^* + \epsilon] \).

*Global minimum* extends the notion in 1D optimisation, i.e., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in \mathbb{R} \).

In optimisation, checking whether a point \( x^* \in \mathbb{R} \) satisfies first-order optimality conditions is often much easier than checking whether \( x^* \) is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

\[
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \quad \text{("weak duality"), or via problem-specific means.}
\]

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that local minimum extends the notion in 1D optimisation, e.g., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in [x^* - \epsilon, x^* + \epsilon] \).

Global minimum extends the notion in 1D optimisation, i.e., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in \mathbb{R} \).

In optimisation, checking whether a point \( x^* \in \mathbb{R} \) satisfies first-order optimality conditions is often much easier than checking whether \( x^* \) is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

\[
\min_{x \in \mathcal{X}} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in \mathcal{X}} L(x, y) \quad \text{("weak duality"), or via problem-specific means.}
\]

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that local minimum extends the notion in 1D optimisation, e.g., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in [x^* - \epsilon, x^* + \epsilon] \).

Global minimum extends the notion in 1D optimisation, i.e., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in \mathbb{R} \).

In optimisation, checking whether a point \( x^* \in \mathbb{R} \) satisfies first-order optimality conditions is often much easier than checking whether \( x^* \) is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

\[
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \quad \text{("weak duality")}, \text{ or via problem-specific means.}
\]

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that \textit{local minimum} extends the notion in 1D optimisation, e.g., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in [x^* - \epsilon, x^* + \epsilon] \).

\textit{Global minimum} extends the notion in 1D optimisation, i.e., \( x^* \in \mathbb{R} \) such that \( f(x^*) \leq f(x) \) for all \( x \in \mathbb{R} \).

In optimisation, checking whether a point \( x^* \in \mathbb{R} \) satisfies first-order optimality conditions is often much easier than checking whether \( x^* \) is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

\[
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \quad \text{("weak duality")},
\]

or via problem-specific means.

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that *local minimum* extends the notion in 1D optimisation, e.g., $x^* \in \mathbb{R}$ such that $f(x^*) \leq f(x)$ for all $x \in [x^* - \epsilon, x^* + \epsilon]$.

*Global minimum* extends the notion in 1D optimisation, i.e., $x^* \in \mathbb{R}$ such that $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}$.

In optimisation, checking whether a point $x^* \in \mathbb{R}$ satisfies first-order optimality conditions is often much easier than checking whether $x^*$ is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

$$
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \quad \text{("weak duality")}, \text{ or via problem-specific means.}
$$

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Witness

Recall that *local minimum* extends the notion in 1D optimisation, e.g., $x^* \in \mathbb{R}$ such that $f(x^*) \leq f(x)$ for all $x \in [x^* - \epsilon, x^* + \epsilon]$.

*Global minimum* extends the notion in 1D optimisation, i.e., $x^* \in \mathbb{R}$ such that $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}$.

In optimisation, checking whether a point $x^* \in \mathbb{R}$ satisfies first-order optimality conditions is often much easier than checking whether $x^*$ is a global minimum.

A witness for global optimality can be obtained from Lagrangian duality,

$$
\min_{x \in X} \max_{y \geq 0} L(x, y) \geq \max_{y \geq 0} \min_{x \in X} L(x, y) \text{ (“weak duality”), or via problem-specific means.}
$$

We have seen the use of the gap to measure the distance to optimality in training SVMs.

For general non-differentiable and non-convex functions, even a witness of local optimality may be hard to come up with, though.
The Convergence

*Global convergence* of an iterative method suggests that from any initial point, the method can be used to approximate the exact solution with an arbitrary precision in the BSS model.

For solving linear systems, we have seen a number of iterative methods with global convergence.

For convex and smooth optimisation problems, both sensible first- and second-order methods exhibit global convergence.

For non-convex optimisation problems and eigenvalue problems, none of the iterative methods we have seen is globally convergent. (There exists so called *homotopy* methods for optimisation of systems of polynomial equalities, which apply also to the eigenvalues and eigenvectors. They come close to being globally convergent, but are impractical.)

Most analyses are also restricted to the BSS machine.
The Convergence

Global convergence of an iterative method suggests that from any initial point, the method can be used to approximate the exact solution with an arbitrary precision in the BSS model.

For solving linear systems, we have seen a number of iterative methods with global convergence.

For convex and smooth optimisation problems, both sensible first- and second-order methods exhibit global convergence.

For non-convex optimisation problems and eigenvalue problems, none of the iterative methods we have seen is globally convergent. (There exists so called homotopy methods for optimisation of systems of polynomial equalities, which apply also to the eigenvalues and eigenvectors. They come close to being globally convergent, but are impractical.)

Most analyses are also restricted to the BSS machine.
The Convergence

Global convergence of an iterative method suggests that from any initial point, the method can be used to approximate the exact solution with an arbitrary precision in the BSS model.

For solving linear systems, we have seen a number of iterative methods with global convergence.

For convex and smooth optimisation problems, both sensible first- and second-order methods exhibit global convergence.

For non-convex optimisation problems and eigenvalue problems, none of the iterative methods we have seen is globally convergent. (There exists so called homotopy methods for optimisation of systems of polynomial equalities, which apply also to the eigenvalues and eigenvectors. They come close to being globally convergent, but are impractical.)

Most analyses are also restricted to the BSS machine.
The Convergence

*Global convergence* of an iterative method suggests that from any initial point, the method can be used to approximate the exact solution with an arbitrary precision in the BSS model.

For solving linear systems, we have seen a number of iterative methods with global convergence.

For convex and smooth optimisation problems, both sensible first- and second-order methods exhibit global convergence.

For non-convex optimisation problems and eigenvalue problems, none of the iterative methods we have seen is globally convergent. (There exists so called *homotopy* methods for optimisation of systems of polynomial equalities, which apply also to the eigenvalues and eigenvectors. They come close to being globally convergent, but are impractical.)

Most analyses are also restricted to the BSS machine.
The Convergence

*Global convergence* of an iterative method suggests that from any initial point, the method can be used to approximate the exact solution with an arbitrary precision in the BSS model.

For solving linear systems, we have seen a number of iterative methods with global convergence.

For convex and smooth optimisation problems, both sensible first- and second-order methods exhibit global convergence.

For non-convex optimisation problems and eigenvalue problems, none of the iterative methods we have seen is globally convergent. (There exists so called *homotopy* methods for optimisation of systems of polynomial equalities, which apply also to the eigenvalues and eigenvectors. They come close to being globally convergent, but are impractical.)

Most analyses are also restricted to the BSS machine.
The rate of convergence is an inverse function of an upper bound on the number of iterations required to reach a given precision.

A sequence of iterates \((x^k)\) has order of convergence \(p\) if the sequence of errors

\[
\|x^{k+1} - x^*\| \leq C\|x^k - x^*\|^p,
\]

where \(C\) possibly depends on \(k\), but can be bounded above by a constant.
The Rate of Convergence

The rate of convergence is an inverse function of an upper bound on the number of iterations required to reach a given precision.

A sequence of iterates \((x^k)\) has order of convergence \(p\) if the sequence of errors

\[ \|x^{k+1} - x^*\| \leq C\|x^k - x^*\|^p, \]

where \(C\) possibly depends on \(k\), but can be bounded above by a constant.
The Rate of Convergence

The rate of convergence is an inverse function of an upper bound on the number of iterations required to reach a given precision.

A sequence of iterates \((x^k)\) has order of convergence \(p\) if the sequence of errors

\[
\|x^{k+1} - x^*\| \leq C \|x^k - x^*\|^p,
\]

where \(C\) possibly depends on \(k\), but can be bounded above by a constant.
Sublinear Convergence

Sequences with $p < 1$ are said to have sublinear convergence.

For example for $p = 0.5$, the complexity estimate depends of the square of the desired accuracy,

i.e., $(\frac{C}{\epsilon})^2$ iterations are required to find $\epsilon$-accurate solution and each digit of accuracy requires the same amount of computation, as all the previous digits combined.

Generally, gradient and subgradient methods are sublinearly convergent, although on for smooth functions, $O(\sqrt{\frac{1}{\epsilon}})$ is both possible and optimal.
Sublinear Convergence

Sequences with $p < 1$ are said to have *sublinear convergence*. For example for $p = 0.5$, the complexity estimate depends on the square of the desired accuracy, i.e., $(\frac{C}{\epsilon})^2$ iterations are required to find $\epsilon$-accurate solution and each digit of accuracy requires the same amount of computation, as all the previous digits combined.

Generally, gradient and subgradient methods are sublinearly convergent, although on for smooth functions, $O(\sqrt{\frac{1}{\epsilon}})$ is both possible and optimal.
Sublinear Convergence

Sequences with $p < 1$ are said to have *sublinear convergence*. For example for $p = 0.5$, the complexity estimate depends of the square of the desired accuracy, i.e., $(\frac{C}{\epsilon})^2$ iterations are required to find $\epsilon$-accurate solution and each digit of accuracy requires the same amount of computation, as all the previous digits combined.

Generally, gradient and subgradient methods are sublinearly convergent, although on for smooth functions, $O(\sqrt{\frac{1}{\epsilon}})$ is both possible and optimal.
Sublinear Convergence

Sequences with $p < 1$ are said to have *sublinear convergence*. For example for $p = 0.5$, the complexity estimate depends of the square of the desired accuracy, i.e., $(\frac{C}{\epsilon})^2$ iterations are required to find $\epsilon$-accurate solution and each digit of accuracy requires the same amount of computation, as all the previous digits combined.

Generally, gradient and subgradient methods are sublinearly convergent, although on for smooth functions, $O(\sqrt{\frac{1}{\epsilon}})$ is both possible and optimal.
Sublinear Convergence

Sequences with $p < 1$ are said to have sublinear convergence.

For example for $p = 0.5$, the complexity estimate depends of the square of the desired accuracy,
i.e., $(\frac{C}{\epsilon})^2$ iterations are required to find $\epsilon$-accurate solution and
each digit of accuracy requires the same amount of computation, as all the
previous digits combined.

Generally, gradient and subgradient methods are sublinearly convergent, although
on for smooth functions, $O(\sqrt{\frac{1}{\epsilon}})$ is both possible and optimal.
Linear Convergence

Sequences with \( p = 1 \) are said to have linear convergence, which means that and the complexity estimate depends of the logarithm of the desired accuracy.

For example for \( \|x^{k+1} - x^*\| \leq C(1 - D)^k \), one needs \( \frac{1}{D} \left( \log C + \log \left( \frac{1}{\epsilon} \right) \right) \) iterations to find \( \epsilon \)-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with \( |\lambda_2/\lambda_1| \) instead of \( 1 - D \).

Gradient methods are linearly convergent for smooth and strongly convex functions.
Linear Convergence

Sequences with $p = 1$ are said to have linear convergence, which means that and the complexity estimate depends on the logarithm of the desired accuracy.

For example for $\|x^{k+1} - x^*\| \leq C(1 - D)^k$, one needs $\frac{1}{D}(\log C + \log(\frac{1}{\epsilon}))$ iterations to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with $|\lambda_2/\lambda_1|$ instead of $1 - D$.

Gradient methods are linearly convergent for smooth and strongly convex functions.
Linear Convergence

Sequences with $p = 1$ are said to have \textit{linear convergence}, which means that and the complexity estimate depends of the logarithm of the desired accuracy.

For example for $\| x^{k+1} - x^* \| \leq C(1 - D)^k$, one needs $\frac{1}{D}(\log C + \log(\frac{1}{\epsilon}))$ iterations to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with $|\lambda_2/\lambda_1|$ instead of $1 - D$.

Gradient methods are linearly convergent for smooth and strongly convex functions.
Linear Convergence

Sequences with $p = 1$ are said to have *linear convergence*, which means that and the complexity estimate depends on the logarithm of the desired accuracy.

For example for $\|x^{k+1} - x^*\| \leq C (1 - D)^k$, one needs $\frac{1}{D} (\log C + \log(\frac{1}{\epsilon}))$ iterations to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with $|\lambda_2/\lambda_1|$ instead of $1 - D$.

Gradient methods are linearly convergent for smooth and strongly convex functions.
Linear Convergence

Sequences with $p = 1$ are said to have linear convergence, which means that and the complexity estimate depends of the logarithm of the desired accuracy.

For example for $\|x^{k+1} - x^*\| \leq C(1 - D)^k$, one needs $\frac{1}{D}(\log C + \log(\frac{1}{\epsilon}))$ iterations to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with $|\lambda_2/\lambda_1|$ instead of $1 - D$.

Gradient methods are linearly convergent for smooth and strongly convex functions.
Linear Convergence

Sequences with $p = 1$ are said to have linear convergence, which means that and the complexity estimate depends of the logarithm of the desired accuracy.

For example for $\|x^{k+1} - x^*\| \leq C(1 - D)^k$, one needs $\frac{1}{D}(\log C + \log(\frac{1}{\epsilon}))$ iterations to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires the same amount of computation.

For solving linear systems, Gauss-Seidel and Jacobi methods had linear rate of convergence.

Power method is locally linearly convergent, with $|\lambda_2/\lambda_1|$ instead of $1 - D$.

Gradient methods are linearly convergent for smooth and strongly convex functions.
Quadratic Convergence

Sequences with $p = 2$ are said to have *quadratic convergence*,
which means that the rate is a double exponential function of the iteration counter
and the complexity estimate depends of the double logarithm of the desired accuracy,
i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.
Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method,
which is locally quadratically convergent under a variety of assumptions.
For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have \emph{quadratic convergence}, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends of the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions.

For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have *quadratic convergence*, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends of the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions.

For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have quadratic convergence, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends of the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions. For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have quadratic convergence, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends on the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions.

For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have *quadratic convergence*, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends of the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions.

For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
Quadratic Convergence

Sequences with $p = 2$ are said to have quadratic convergence, which means that the rate is a double exponential function of the iteration counter and the complexity estimate depends of the double logarithm of the desired accuracy, i.e., $\log \log \frac{1}{\epsilon}$ iterations are required to find $\epsilon$-accurate solution.

Each digit of accuracy hence requires less computation than the previous one. A prototypical example includes the Newton method, which is locally quadratically convergent under a variety of assumptions.

For alternating minimisation in matrix completion, we have also seen the super-linear convergence analysis of Keshavan et al.
The Stability

The stability of an algorithm suggests how many digits of accuracy you lose, when you perform the computation using floating-point numbers on a Turing machine, above and beyond what should be expected from the condition number.

This is often related to the magnitude of some intermediate results considered during the run of the algorithm.

In linear systems, we have seen that the Gaussian elimination on an $n \times n$ zero-one matrix can produce $2^{n-1}$ in its run, resulting in the loss of $n$ bits of precision, in the worst-case.

In contrast the numbers are $O(n^{2/3})$ for partial pivoting and $O(n^{1/2})$ for complete pivoting and $O(n^{1/2})$ for QR factorization in a random model, indicating little or no loss of precision.

For optimisation, a well-implemented second-order method should be more stable than a so called “accelerated” first-order method.
The Stability

The stability of an algorithm suggests how many digits of accuracy you lose, when you perform the computation using floating-point numbers on a Turing machine, above and beyond what should be expected from the condition number.

This is often related to the magnitude of some intermediate results considered during the run of the algorithm.

In linear systems, we have seen that the Gaussian elimination on an $n \times n$ zero-one matrix can produce $2^{n-1}$ in its run, resulting in the loss of $n$ bits of precision, in the worst-case.

In contrast the numbers are $O(n^{2/3})$ for partial pivoting and $O(n^{1/2})$ for complete pivoting and $O(n^{1/2})$ for QR factorization in a random model, indicating little or no loss of precision.

For optimisation, a well-implemented second-order method should be more stable than a so called “accelerated” first-order method.
The Stability

The stability of an algorithm suggests how many digits of accuracy you lose, when you perform the computation using floating-point numbers on a Turing machine, above and beyond what should be expected from the condition number.

This is often related to the magnitude of some intermediate results considered during the run of the algorithm.

In linear systems, we have seen that the Gaussian elimination on an $n \times n$ zero-one matrix can produce $2^{n-1}$ in its run, resulting in the loss of $n$ bits of precision, in the worst-case.

In contrast the numbers are $O(n^{2/3})$ for partial pivoting and $O(n^{1/2})$ for complete pivoting and $O(n^{1/2})$ for QR factorization in a random model, indicating little or no loss of precision.

For optimisation, a well-implemented second-order method should be more stable than a so called “accelerated” first-order method.
The Stability

The stability of an algorithm suggests how many digits of accuracy you lose, when you perform the computation using floating-point numbers on a Turing machine, above and beyond what should be expected from the condition number.

This is often related to the magnitude of some intermediate results considered during the run of the algorithm.

In linear systems, we have seen that the Gaussian elimination on an $n \times n$ zero-one matrix can produce $2^{n-1}$ in its run, resulting in the loss of $n$ bits of precision, in the worst-case.

In contrast the numbers are $O(n^{2/3})$ for partial pivoting and $O(n^{1/2})$ for complete pivoting and $O(n^{1/2})$ for QR factorization in a random model, indicating little or no loss of precision.

For optimisation, a well-implemented second-order method should be more stable than a so called “accelerated” first-order method.
The Stability

The stability of an algorithm suggests how many digits of accuracy you lose, when you perform the computation using floating-point numbers on a Turing machine, above and beyond what should be expected from the condition number.

This is often related to the magnitude of some intermediate results considered during the run of the algorithm.

In linear systems, we have seen that the Gaussian elimination on an $n \times n$ zero-one matrix can produce $2^{n-1}$ in its run, resulting in the loss of $n$ bits of precision, in the worst-case.

In contrast the numbers are $O(n^{2/3})$ for partial pivoting and $O(n^{1/2})$ for complete pivoting and $O(n^{1/2})$ for QR factorization in a random model, indicating little or no loss of precision.

For optimisation, a well-implemented second-order method should be more stable than a so called “accelerated” first-order method.
The Basics

- Problem and Instance
- Turing Machine
- Integers in a Turing Machine
- Floating-Point Numbers
- BSS Machine
- Computability and Complexity
The Tools

- The spectrum
- Matrix norms
- Solvers for linear systems
- Solvers for least squares
- Solvers for eigenvalue problems
- Necessary optimality conditions
- Solvers for sub-differentiable opt.
- Solvers for differentiable opt.
The Concepts

- The Condition
- The Witness
- The Convergence
- The Rate of Convergence
- The Stability