Continuity and Robustness of Programs
Seminar: Robustness of Hardware and Software Systems
Prof. Dr.-Ing. Jan Reineke

Markus Schneider

Saarbrücken, December 12, 2013
For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.

- in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.

- in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
- in **mobile devices**: they use slightly stale satellite data
- in **randomized and approximate algorithms** for performance gains
- in **differential privacy** to guarantee privacy in statistical databases
Motivation

- For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.
  - in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
  - in **mobile devices**: they use slightly stale satellite data
  - in **randomized and approximate algorithms** for performance gains
  - in **differential privacy** to guarantee privacy in statistical databases
- This uncertainty can be **probabilistic or nondeterministic**.
Motivation

For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.

- in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
- in **mobile devices**: they use slightly stale satellite data
- in **randomized and approximate algorithms** for performance gains
- in **differential privacy** to guarantee privacy in statistical databases

This uncertainty can be **probabilistic or nondeterministic**.

→ We will introduce a concept of **continuity for programs**.
The Challenge: Handling the Control Flow

- Conditional branching.

```
1: if x > 2 then
2:   y := \frac{1}{2} \cdot x
3: else
4:   y := -5x + 11
5: end if
```
Conditional branching.
1: \( \text{if } x > 2 \text{ then} \)
2: \( y := \frac{1}{2} \cdot x \)
3: \( \text{else} \)
4: \( y := -5x + 11 \)
5: \( \text{end if} \)

Loops.
1: \( \text{while } W \neq \emptyset \text{ do} \)
2: choose edge \((v, w) \in G\) such that \(d[w]\) is minimal
3: remove \((v, w)\) from \(W\)
4: \( \text{if } d[w] + G[w, v] < d[v] \text{ then} \)
6: \( \text{end if} \)
7: \( \text{end while} \)
The Challenge: Handling the Control Flow

- **Conditional branching.**
  1: `if x > 2 then`
  2: \( y := \frac{1}{2} \cdot x \)
  3: `else`
  4: \( y := -5x + 11 \)
  5: `end if`

- **Loops.**
  1: `while W \neq \emptyset do`
  2: choose edge \((v, w)\) \(\in G\) such that \(d[w]\) is minimal
  3: remove \((v, w)\) from \(W\)
  4: `if \(d[w] + G[w, v] < d[v]\) then`
  6: `end if`
  7: `end while`

\(\rightarrow\) **Control flow** makes an automated continuity analysis difficult.
A Necessary Tool: Metrics

- We consider a program as the mathematical function that it implements.
- To be able to talk about continuity we have to define a metric for each datatype.
A Necessary Tool: Metrics

- We consider a **program** as the **mathematical function** that it implements.
- To be able to talk about continuity we have to **define a metric for each datatype**.
- Examples of metrics:
  - **integer** and **real**, associated with the **Euclidean metric**
    \[
    d(x, y) = |x - y|
    \]
A Necessary Tool: Metrics

- We consider a **program** as the mathematical function that it implements.
- To be able to talk about continuity we have to **define a metric for each datatype**.
- Examples of metrics:
  - *integer* and *real*, associated with the **Euclidean metric**
    \[
    d(x, y) = |x - y|
    \]
  - *integer arrays* and *real arrays*, associated with the **maximum norm**
    \[
    d(A_1, A_2) = L_\infty(A_1, A_2) = \max_i(|A_1[i] - A_2[i]|)
    \]
Closeness of Program States

Continuity analysis of programs requires a definition of a “distance” between two program states.
Closeness of Program States

Continuity analysis of programs requires a definition of a “distance” between two program states.

Given two states $\sigma$ and $\sigma' \in \Sigma(P)$ and any $\epsilon > 0$, we define:

- $\sigma$ and $\sigma'$ are $\epsilon$-close with respect to variable $x_i$ and write $\sigma \approx_{\epsilon,i} \sigma'$:

$$d(\sigma(i), \sigma'(i)) < \epsilon$$
Closeness of Program States

Continuity analysis of programs requires a definition of a “distance” between two program states.

Given two states $\sigma$ and $\sigma' \in \Sigma(P)$ and any $\epsilon > 0$, we define:

- $\sigma$ and $\sigma'$ are $\epsilon$-close with respect to variable $x_i$ and write
  $$\sigma \approx_{\epsilon,i} \sigma' \iff d(\sigma(i), \sigma'(i)) < \epsilon$$

- $\sigma'$ is an $\epsilon$-perturbation of $\sigma$ with respect to variable $x_i$ and write
  $$\sigma \equiv_{\epsilon,i} \sigma' \iff \sigma \approx_{\epsilon,i} \sigma' \land \forall j \neq i : \sigma(j) = \sigma'(j)$$
Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program
Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program
Continuity of a Program

**Well-known \( \epsilon-\delta \)-Definition of Continuous Functions:**
A function \( f : D \rightarrow \mathbb{R} \) is continuous at a point \( x \in D \), if

\[
\forall \epsilon > 0 \ \exists \delta > 0 \ \forall y \in D : \ |x - y| < \delta \Rightarrow |f(x) - f(y)| < \epsilon
\]
Well-known $\epsilon$-$\delta$-Definition of Continuous Functions:
A function $f : D \to \mathbb{R}$ is continuous at a point $x \in D$, if

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in D : |x - y| < \delta \Rightarrow |f(x) - f(y)| < \epsilon$$

Continuity of a Program:
A program $P$ is continuous at a state $\sigma$ with respect to an input variable $x_i$ and an output variable $x_j$, if

$$\forall \epsilon > 0 \exists \delta > 0 \forall \sigma' \in \Sigma(P) : \sigma \equiv_{\delta,i} \sigma' \Rightarrow \llbracket P \rrbracket(\sigma) \approx_{\epsilon,j} \llbracket P \rrbracket(\sigma')$$
Verifying Continuity (1)

- **Goal:** establish an automated framework for proving a program to be continuous
Goal: establish an automated framework for proving a program to be continuous

The analysis is

- **sound** (a program proven continuous is indeed continuous),
- but **incomplete** (a program may be continuous even if the analysis is not able to derive this).
Verifying Continuity (1)

- **Goal**: establish an automated framework for proving a program to be continuous

- The analysis is
  - **sound** (a program proven continuous is indeed continuous),
  - but **incomplete** (a program may be continuous even if the analysis is not able to derive this).

- Breaking down a program into its syntactic substructures we get a set of **inference rules** of the style

\[
P \text{ is SKIP or } x := e \quad \frac{}{b \vdash \text{Cont}(P, \text{In}, \text{Out})}
\]


to derive **continuity judgements**.
Disallowing divisions the critical statements are **conditional branches**.

- The branches have to be *output-equivalent* at the decision boundary of the branch.

```
1: if x > 2 then
2:   y := \frac{1}{2} \cdot x
3: else
4:   y := -5x + 11
5: end if
```
Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program
Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function \( f : D \to \mathbb{R} \) is Lipschitz continuous, if there is a constant \( K \) so that any \( \pm \epsilon \)-change to \( x \) can change \( f(x) \) at most by \( \pm K \cdot \epsilon \).
Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function $f : D \to \mathbb{R}$ is Lipschitz continuous, if there is a constant $K$ so that any $\pm \varepsilon$-change to $x$ can change $f(x)$ at most by $\pm K \cdot \varepsilon$.

Lipschitz Continuity of a Program:
Let $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function that takes the size of variable $x_i$ as its input. A program $P$ is $K$-Lipschitz with respect to an input variable $x_i$ and an output variable $x_j$, if $\forall \sigma, \sigma' \in \Sigma(P)$ and $\forall \varepsilon > 0$

$$\sigma \equiv_{\varepsilon,i} \sigma' \Rightarrow \llbracket P \rrbracket(\sigma) \approx_{K \cdot \varepsilon,j} \llbracket P \rrbracket(\sigma')$$
Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function \( f : D \to \mathbb{R} \) is Lipschitz continuous, if there is a constant \( K \) so that any \( \pm \epsilon \)-change to \( x \) can change \( f(x) \) at most by \( \pm K \cdot \epsilon \).

Lipschitz Continuity of a Program:
Let \( K : \mathbb{N} \to \mathbb{R}_{\geq 0} \) be a function that takes the size of variable \( x_i \) as its input. A program \( P \) is \( K \)-Lipschitz with respect to an input variable \( x_i \) and an output variable \( x_j \), if \( \forall \sigma, \sigma' \in \Sigma(P) \) and \( \forall \epsilon > 0 \)

\[
\sigma \equiv_{\epsilon,i} \sigma' \Rightarrow \llbracket P \rrbracket(\sigma) \approx_{K \cdot \epsilon,j} \llbracket P \rrbracket(\sigma')
\]

where \( K \) only depends on the size of \( \sigma(i) \). The size of a variable \( v \) is defined as

- \( \|v\| := 1 \), if \( v \) is an integer or a real,
- \( \|v\| := N \), if \( v \) is an array of size \( N \).
Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function \( f : D \to \mathbb{R} \) is Lipschitz continuous, if there is a constant \( K \) so that any \( \pm \epsilon \)-change to \( x \) can change \( f(x) \) at most by \( \pm K \cdot \epsilon \).

Lipschitz Continuity of a Program:
Let \( K : \mathbb{N} \to \mathbb{R}_{\geq 0} \) be a function that takes the size of variable \( x_i \) as its input. A program \( P \) is \( K \)-Lipschitz with respect to an input variable \( x_i \) and an output variable \( x_j \), if \( \forall \sigma, \sigma' \in \Sigma(P) \) and \( \forall \epsilon > 0 \)

\[
\sigma \equiv_{\epsilon, i} \sigma' \land (||\sigma(i)|| = ||\sigma'(i)||) \Rightarrow \| \langle P \rangle (\sigma) \| \approx_{K, \epsilon, j} \| \langle P \rangle (\sigma') \|
\]

where \( K \) only depends on the size of \( \sigma(i) \). The size of a variable \( v \) is defined as

- \( ||v|| := 1 \), if \( v \) is an integer or a real,
- \( ||v|| := N \), if \( v \) is an array of size \( N \).
Example (1): Sorting Algorithms

- \(\text{Sort}_1\) maps an array to its sorted permutation.

**Example:**

\[
\text{Sort}_1(6, 3, 3, 1) = (1, 3, 3, 6)
\]
\[
\text{Sort}_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)
\]

Perturbing each item of an array at most by \(\pm \epsilon\) changes each item of the output array at most by \(\pm \epsilon\).

- \(\text{Sort}_2\) maps an array to the list of indices giving the order.

**Example:**

\[
\text{Sort}_2(6, 3, 3, 1) = (4, 2, 3, 1)
\]
\[
\text{Sort}_2(6, 3 + \epsilon, 3, 1) = (4, 3, 2, 1)
\]

Perturbing one item by \(\pm \epsilon\) can already lead to unbounded changes in the corresponding outputs.

\(\text{Sort}_1\) is Lipschitz continuous, \(\text{Sort}_2\) is not even continuous.
Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.

Example:

$$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
$$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

Perturbing each item of an array at most by $\pm \epsilon$ changes each item of the output array at most by $\pm \epsilon$. 
Example (1): Sorting Algorithms

- $\text{Sort}_1$ maps an array to its sorted permutation.
  
  Example:
  
  \[
  \text{Sort}_1(6, 3, 3, 1) = (1, 3, 3, 6) \\
  \text{Sort}_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)
  \]

  Perturbing each item of an array at most by $\pm\epsilon$ changes each item of the output array at most by $\pm\epsilon$.

- $\text{Sort}_2$ maps an array to the list of indices giving the order.
  
  Example:

  \[
  \text{Sort}_2(6, 3, 3, 1) = (4, 2, 3, 1) \\
  \text{Sort}_2(6, 3 + \epsilon, 3, 1) = (4, 3, 2, 1)
  \]

$\text{Sort}_1$ is Lipschitz continuous, $\text{Sort}_2$ is not even continuous.
Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  Example:

$$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
$$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

Perturbing each item of an array at most by $\pm \epsilon$ changes each item of the output array at most by $\pm \epsilon$.

- $Sort_2$ maps an array to the list of indices giving the order.
  Example:

$$Sort_2(6, 3, 3, 1) = (4, 2, 3, 1)$$
$$Sort_2(6, 3 + \epsilon, 3, 1) = (4, 3, 2, 1)$$

Perturbing one item by $\pm \epsilon$ can already lead to unbounded changes in the corresponding outputs.
Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  
  Example:

  $$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
  $$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

  Perturbing each item of an array at most by $\pm \epsilon$ changes each item of the output array at most by $\pm \epsilon$.

- $Sort_2$ maps an array to the list of indices giving the order.
  
  Example:

  $$Sort_2(6, 3, 3, 1) = (4, 2, 3, 1)$$
  $$Sort_2(6, 3 + \epsilon, 3, 1) = (4, 3, 2, 1)$$

  Perturbing one item by $\pm \epsilon$ can already lead to unbounded changes in the corresponding outputs.

  → $Sort_1$ is Lipschitz continuous, $Sort_2$ is not even continuous.
Example (2): Shortest Path Algorithms

- $SP_1$ maps a graph to its minimal distance array $d$.
- $SP_2$ maps a graph to an array containing the shortest paths.

$\rightarrow SP_1$ is continuous, $SP_2$ is not.
Example (2): Shortest Path Algorithms

- $SP_1$ maps a graph to its minimal distance array $d$.
- $SP_2$ maps a graph to an array containing the shortest paths.

→ $SP_1$ is continuous, $SP_2$ is not.

We have to define the output of our program exactly!
Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes *proportionally* to any change on the inputs.

A program is called robust, if it is $K$-Lipschitz for some Lipschitz constant $K$. 
Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.
- The upper bound $K \cdot \epsilon$ on the output changes does not depend on the values of the input variables.
Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.
- The upper bound $K \cdot \varepsilon$ on the output changes does not depend on the values of the input variables.

$\rightarrow$ The program behaves predictably on uncertain inputs.
Robustness of Programs

For Lipschitz continuous programs we can state:

▶ The output changes proportionally to any change on the inputs.

▶ The upper bound $K \cdot \varepsilon$ on the output changes does not depend on the values of the input variables.

→ The program behaves predictably on uncertain inputs.

A program is called robust, if it is $K$-Lipschitz for some Lipschitz constant $K$. 
Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program
Our Two Step Procedure

The sequence of assignment or `skip`-statements that $P$ executes on some input is called a control flow path of $P$. 
Our Two Step Procedure

The sequence of assignment or \texttt{SKIP}-statements that $P$ executes on some input is called a \textbf{control flow path} of $P$.

Let $x_j$ be the input and $x_i$ be the output variable of our program.
Our Two Step Procedure

The sequence of assignment or *skip* -statements that $P$ executes on some input is called a **control flow path** of $P$.

Let $x_j$ be the input and $x_i$ be the output variable of our program.

**Lipschitz continuity** of a program is proven by establishing that

1. $P$ is **continuous** in all states w.r.t. input $x_j$ and output $x_i$.
2. Each **control flow path** of $P$ is $K$-Lipschitz w.r.t. input $x_j$ and output $x_i$. 
The remaining task is to find out the Lipschitz constants for each control flow path (if there exists one).
The Idea for Finding Lipschitz Constants

The remaining task is to find out the Lipschitz constants for each control flow path (if there exists one).

Our approach:

▶ Compute **Lipschitz matrices** containing upper bounds on the slope of any computation that can be carried out in a control flow path of $P$. 
Lipschitz Matrices

Let program $P$ have $n$ variables $x_1, \ldots, x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ as its matrix elements.
Lipschitz Matrices

Let program $P$ have $n$ variables $x_1, \ldots, x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ as its matrix elements.
- We will derive a set $\mathcal{J}$ of Lipschitz matrices.
- A judgement $P : \mathcal{J}$ means:
  For each control flow path $C$ in $P$ and each $x_i, x_j$ there is a $J \in \mathcal{J}$ such that $C$ is $J_{ij}$-Lipschitz in input $x_j$ and output $x_i$. 

Note the similarity to the Jacobian:
- If the program represents a differentiable function, $J_{ij}$ is an upper bound on $|\frac{\partial x_i}{\partial x_j}|$. 
Lipschitz Matrices

Let program $P$ have $n$ variables $x_1, \ldots, x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ as its matrix elements.
- We will derive a set $\mathcal{J}$ of Lipschitz matrices.
- A judgement $P : \mathcal{J}$ means:
  For each control flow path $C$ in $P$ and each $x_i, x_j$ there is a $J \in \mathcal{J}$ such that $C$ is $J_{ij}$-Lipschitz in input $x_j$ and output $x_i$.

Note the similarity to the **Jacobian**:

- If the program represents a differentiable function, $J_{ij}$ is an upper bound on $|\frac{\partial x_i}{\partial x_j}|$. 
Merging of Lipschitz Matrices

- Given any judgement $P : \mathcal{J}$, we can merge two arbitrary Lipschitz matrices $A$ and $B \in \mathcal{J}$. Formally, we can infer

$$P : (\mathcal{J} \setminus \{A, B\}) \cup \{A \sqcup B\}$$

where the **merge operation** $\sqcup$ is defined as

$$(A \sqcup B)_{ij} = \max(A_{ij}, B_{ij}) \quad \forall i, j \in \{1, \ldots, n\}$$
Rules for Deriving Lipschitz Matrices (1)

skip \[ \text{SKIP} : \{1\} \]
Rules for Deriving Lipschitz Matrices (1)

**skip**

\[ \text{skip} \quad \text{SKIP} : \{1\} \]

**weaken**

\[ P : \mathcal{J} \quad J_1, J_2 \in \mathcal{J} \]

\[ P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\} \]
Rules for Deriving Lipschitz Matrices (1)

- **skip**
  \[ \text{skip} \quad \text{SKIP : } \{1\} \]

- **weaken**
  \[ \frac{P : J \quad J_1, J_2 \in J}{P : (J \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}} \]

- **ITE**
  \[ \frac{P_1 : J_1 \quad P_2 : J_2}{(\text{IF } B \text{ THEN } P_1 \text{ ELSE } P_2) : J_1 \sqcup J_2} \]

\[ E : J \]

\[ \forall J \in J \forall i, j : J \quad J_{ij} \geq 1 \lor J_{ij} = 0 \]

\[ P : \{J_1 \cdot J_2 \cdot \ldots \cdot J_M | J_i \in J\} \]
Rules for Deriving Lipschitz Matrices (1)

skip $\text{skip}$

$\text{SKIP : } \{1\}$

weaken

$P : J, J_1, J_2 \in J$

$P : (J \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}$

ITE

$P_1 : J_1, P_2 : J_2$

$(\text{IF } B \text{ THEN } P_1 \text{ ELSE } P_2) : J_1 \cup J_2$

sequence

$P_1 : J_1, P_2 : J_2$

$(P_1; P_2) : \{J_2 \cdot J_1 \mid J_1 \in J_1, J_2 \in J_2\}$
Rules for Deriving Lipschitz Matrices (1)

\[
\text{skip} \quad \text{SKIP : } \{1\}
\]

weaken
\[
P : \mathcal{J} \quad J_1, J_2 \in \mathcal{J} \\
\quad \quad \quad P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}
\]

ITE
\[
P_1 : \mathcal{J}_1 \quad P_2 : \mathcal{J}_2 \\
\quad (\text{IF } B \text{ THEN } P_1 \text{ ELSE } P_2) : \mathcal{J}_1 \cup \mathcal{J}_2
\]

sequence
\[
P_1 : \mathcal{J}_1 \quad P_2 : \mathcal{J}_2 \\
\quad (P_1 ; P_2) : \{J_2 \cdot J_1 \mid J_1 \in \mathcal{J}_1, J_2 \in \mathcal{J}_2\}
\]

while
\[
P = \text{WHILE } b \text{ DO } R \quad R : \mathcal{J} \quad \text{Bound}^+(P, M) \\
\quad \forall J \in \mathcal{J} \forall i, j : J_{ij} \geq 1 \lor J_{ij} = 0 \\
\quad P : \{J_1 \cdot J_2 \cdot \ldots \cdot J_M \mid J_i \in \mathcal{J}\}
\]
Rules for Deriving Lipschitz Matrices (2)

For assignments we first define a vector $\nabla e$ whose $j$-th element is an upper bound on $|\frac{\partial [e]}{\partial x_j}|$:

$$\nabla e(j) = \begin{cases} 
0, & \text{if } e \text{ is a constant} \\
1, & \text{if } e \text{ is } x_j \text{ or } x_j[k] \text{ for some } k \\
0, & \text{if } e \text{ is } x_l \text{ or } x_l[k] \text{ for some } k \text{ and } l \neq j \\
\nabla a(j) + \nabla b(j), & \text{if } e \text{ is } (a + b) \\
\nabla a(j) |b| + \nabla b(j) |a|, & \text{if } e \text{ is } (a \cdot b) \text{ and } a \text{ or } b \text{ is a constant} \\
\infty, & \text{otherwise}
\end{cases}$$
Rules for Deriving Lipschitz Matrices (2)

For assignments we first define a vector $\nabla e$ whose $j$-th element is an upper bound on $|\frac{\partial [e]}{\partial x_j}|$:

$$\nabla e(j) = \begin{cases} 
0, & \text{if } e \text{ is a constant} \\
1, & \text{if } e \text{ is } x_j \text{ or } x_j[k] \text{ for some } k \\
0, & \text{if } e \text{ is } x_l \text{ or } x_l[k] \text{ for some } k \text{ and } l \neq j \\
\nabla a(j) + \nabla b(j), & \text{if } e \text{ is } (a + b) \\
\nabla a(j)|b| + \nabla b(j)|a|, & \text{if } e \text{ is } (a \cdot b) \text{ and } a \text{ or } b \text{ is a constant} \\
\infty, & \text{otherwise} 
\end{cases}$$

assign $(x_i := e) : \{J\}$ where $J_{kj} := \begin{cases} 
\nabla e(j), & \text{if } k = i \\
1, & \text{if } k = j \neq i \\
0, & \text{otherwise} 
\end{cases}$
array-assign \((x_i[m] := e) : \{J, I\}\)

with the same matrix \(J\): \(J_{kj} := \begin{cases} 
\nabla_e(j), & \text{if } k = i \\
1, & \text{if } k = j \neq i \\
0, & \text{otherwise}
\end{cases}\)
Example: Dijkstra’s-Algorithm

\textsc{Dijkstra}(G: \text{real array}, \text{src}: \text{int})

1: \quad ... \\
2: \quad \textbf{while } W \neq \emptyset \textbf{ do} \\
3: \quad \text{choose edge } (v, w) \in G \text{ such that } d[w] \text{ is minimal} \\
4: \quad \text{remove } (v, w) \text{ from } W \\
5: \quad \textbf{if } d[w] + G[w, v] \text{ < } d[v] \textbf{ then} \\
6: \quad \quad d[v] := d[w] + G[w, v] \\
7: \quad \textbf{end if} \\
8: \quad \textbf{end while}
Example: Dijkstra’s-Algorithm

\textbf{Dijkstra}(G: real array, \textit{src}: int)

1: ... 
2: \textbf{while} \ W \neq \emptyset \ \textbf{do}
3: \quad \text{choose edge} \ (v, w) \in G \ \text{such that} \ d[w] \ \text{is minimal}
4: \quad \text{remove} \ (v, w) \ \text{from} \ W
5: \quad \text{if} \ d[w] + G[w, v] < d[v] \ \text{then}
6: \quad \quad d[v] := d[w] + G[w, v]
7: \quad \text{end if}
8: \text{end while}

\textbf{Dijkstra} is continuous and we can infer the Lipschitz matrix

\[
\begin{pmatrix}
1 & 0 \\
N & 1
\end{pmatrix}
\]

so that \textbf{Dijkstra} is $N$-Lipschitz in input $G =: x_0$ and output $d =: x_1$, where $N$ denotes the number of edges in $G$. 
Conclusion

- We asked for a theory about **robustness** of programs to uncertainty.
- **Lipschitz continuity** is an adequate answer to this question. It is a strong property.
- Developing an **automated** continuity analysis is demanding.
- The analysis is proven to be **sound**, but **incomplete**.
Conclusion

- We asked for a theory about robustness of programs to uncertainty.
- Lipschitz continuity is an adequate answer to this question. It is a strong property.

- Developing an automated continuity analysis is demanding.
- The analysis is proven to be sound, but incomplete.

- Arising questions:
  - Is it satisfactory to live without divisions?
  - The degree of automation remains unclear.
