Hoare Calculation and its Application

Robert Lang

TUM

March 2008 - Saint Petersburg - JASS 2008
A first example

```python
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end
end
```
A first example

function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

How can we prove, that for $x, y \in \mathbb{N}_0$:

$$result(x,y) = x + y$$
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on x:
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on x:
x = 0
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on x:
\[
x = 0 \Rightarrow \text{result}(0,y) = y, \text{ and } y = y + x
\]
\[
x == 0
\]
function result(x,y)  
    if x == 0  
        return (y);  
    else  
        result(x-1,y+1);  
    end

Proof of the assertion by induction on x:  
\[
x = 0 \Rightarrow result(0,y) = y, \text{ and } y = y + x \checkmark
\]
\[x == 0\]
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on $x$:

$x = 0 \Rightarrow result(0,y) = y$, and $y = y + x \checkmark$

$\quad x==0$

Let the assumption be proved for some $x \in \mathbb{N}_0$ and all $y \in \mathbb{N}_0$. 
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on x:
\[ x = 0 \Rightarrow result(0,y) = y, \text{ and } y = y + x \checkmark \]
\[ x==0 \]

Let the assumption be proved for some \( x \in \mathbb{N}_0 \) and all \( y \in \mathbb{N}_0 \).
\[ \Rightarrow result(x+1,y) = result(x,y+1) \]
\[ \text{else} \]
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on $x$:

$x = 0 \Rightarrow result(0,y) = y, \text{ and } y = y + x$ \[ x==0] 

Let the assumption be proved for some $x \in \mathbb{N}_0$ and all $y \in \mathbb{N}_0$.  
$\Rightarrow result(x+1,y) = result(x,y+1)$ 
$\Rightarrow x + (y + 1)$ 
induction hypothesis
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on x:
\[ x = 0 \Rightarrow \text{result}(0,y) = y, \text{ and } y = y + x \checkmark \]
\[ x=0 \]

Let the assumption be proved for some \( x \in \mathbb{N}_0 \) and all \( y \in \mathbb{N}_0 \).
\[ \Rightarrow \text{result}(x+1,y) = \begin{cases} \text{result}(x,y+1) & \text{else} \\ x + (y + 1) & \text{induction hypotheose} \end{cases} \]
\[ \Rightarrow \text{result}(x+1,y) = x + (y + 1) = (x + 1) + y \]
function result(x,y)
    if x == 0
        return (y);
    else
        result(x-1,y+1);
    end

Proof of the assertion by induction on $x$:

$x = 0 \Rightarrow result(0,y) = y$, and $y = y + x \checkmark$

$x == 0$

Let the assumption be proved for some $x \in \mathbb{N}_0$ and all $y \in \mathbb{N}_0$.

$\Rightarrow result(x+1,y) = result(x,y+1)$

$\begin{align*}
&\text{else} \\
&\text{induction hypothesis} \\
&\Rightarrow result(x+1,y) = x + (y + 1) = (x + 1) + y
\end{align*} \square$
A second example

function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;
A second example

function result_2(x, y)
    while x > 0
        x = x - 1;
        y = y + 1;
    end
    return y;

How can we prove, that for $x, y \in \mathbb{N}_0$:

$$\text{result}_2(x, y) = x + y$$
A second example

function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

How can we prove, that for $x, y \in \mathbb{N}_0$: 

$$\text{result}_2(x,y) = x + y$$

As easy as in the first example?
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
x = 0
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
\[ x = 0 \Rightarrow \text{result}_2(0,y) = y \text{ and } y = y + x \]
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
\[ x = 0 \Rightarrow \text{result}_2(0,y) = y \text{ and } y = y + x \]
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
\[ x = 0 \Rightarrow \text{result}_2(0,y) = y \text{ and } y = y + x \]

Let the assumption be proved for some \( x \in \mathbb{N}_0 \) and all \( y \in \mathbb{N}_0 \).
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on $x$:
$x = 0 \Rightarrow result_2(0,y) = y$ and $y = y + x \checkmark$

Let the assumption be proved for some $x \in \mathbb{N}_0$ and all $y \in \mathbb{N}_0$. 
$\text{result}_2(x+1,y)$
function result_2(x,y)
    while x > 0
        x = x-1;
        y = y+1;
    end
    return y;

Try to prove the assertion by induction on x:
\[ x = 0 \Rightarrow \text{result}_2(0,y) = y \text{ and } y = y + x \checkmark \]

Let the assumption be proved for some \( x \in \mathbb{N}_0 \) and all \( y \in \mathbb{N}_0 \).
result_2(x+1,y) = ... It doesn’t work!
What is the problem?

There's no recursive run of result_2. Number of while-loop iterations depends on x. Values of x, y are changing during running time. ⇒ Mathematical methods of proof won't last! ⇒ We need new tools!
What is the problem?

- There's no recursive run of result_2.
What is the problem?

- There’s no recursive run of `result_2`.
- Number of `while`-loop-iterations depends on $x$. 
What is the problem?

- There’s no recursive run of `result_2`.
- Number of `while`-loop-iterations depends on `x`.
- Values of `x, y` are changing during running time.
What is the problem?

- There’s no recursive run of `result_2`.
- Number of `while`-loop-iterations depends on `x`.
- Values of `x`, `y` are changing during running time.

⇒ Mathematical methods of proof won’t last!
What is the problem?

- There's no recursive run of result_2.
- Number of while-loop-iterations depends on $x$.
- Values of $x, y$ are changing during running time.

$\Rightarrow$ Mathematical methods of proof won't last!

$\Rightarrow$ We need new tools!
Challenges

Let $P$ be a given program. We want to prove, that
Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
2. $P$ works for a given domain in that way it is built for.
Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
2. $P$ works for a given domain in that way it is built for.

Both tasks are as hard as the Halting Problem.
Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
2. $P$ works for a given domain in that way it is built for.

- Both tasks are as hard as the Halting Problem.
- You must prove them for every single $P$. 
Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
2. $P$ works for a given domain in that way it is built for.

- Both tasks are as hard as the Halting Problem.
- You must prove them for every single $P$.

In the following algorithms the termination is assumed.
Challenges

Let $P$ be a given program. We want to prove, that

1. $P$ terminates for all valid inputs.
2. $P$ works for a given domain in that way it is built for.

- Both tasks are as hard as the Halting Problem.
- You must prove them for every single $P$.

In the following algorithms the termination is assumed.
⇒ We just meet challenge 2 using Hoare Calculation . . .
C.A.R. Hoare

I conclude that there are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.
C.A.R. Hoare

Sir Charles Antony Richard Hoare

Source: Wikipedia, the free encyclopedia
Sir Charles Antony Richard Hoare

Source: Wikipedia, the free encyclopedia

“I conclude that there are two ways of constructing a software design:
I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies.
C.A.R. Hoare

Sir Charles Antony Richard Hoare

Source: Wikipedia, the free encyclopedia

“I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.”
Hoare-Triple

\{P\} S \{Q\}
Hoare-Triple

\{P\} \; S \; \{Q\}

- P, Q predicates with values true or false
Hoare-Triple

\{P\} S \{Q\}

- P, Q \textit{predicates} with values true or false
- S \textit{statement}, a program with correct syntax
Hoare-Triples are binary expressions with values true or false.

\[
\{P\} \, S \, \{Q\}
\]

- \(P, \, Q\) predicates with values true or false
- \(S\) statement, a program with correct syntax
Hoare-Triple

\{P\} S \{Q\}

- \(P, Q\) predicates with values true or false
- \(S\) statement, a program with correct syntax

Hoare-Triples are binary expressions with values true or false.

\{P\} S \{Q\} = true : ⇔
Hoare-Triple

\{P\} S \{Q\}

- P, Q predicates with values true or false
- S statement, a program with correct syntax

Hoare-Triples are binary expressions with values true or false.

\{P\} S \{Q\} = true \iff
If the predicate \{P\} is true immediately before execution of S, then immediately S has terminated, the predicate \{Q\} is true.
**Hoare-Triple**

\[ \{ P \} \, S \, \{ Q \} \]

- P, Q **predicates** with values true or false
- S **statement**, a program with correct syntax

Hoare-Triples are binary expressions with values true or false.

\[ \{ P \} \, S \, \{ Q \} = \text{true} : \iff \]

If the predicate \{ P \} is true immediately before execution of S, then immediately S has terminated, the predicate \{ Q \} is true.

Notation: \[ \frac{X}{Y} \]
Hoare-Triple

\{P\} S \{Q\}

- P, Q predicates with values true or false
- S statement, a program with correct syntax

Hoare-Triples are binary expressions with values true or false.

\{P\} S \{Q\} = true \iff
If the predicate \{P\} is true immediately before execution of S, then immediately S has terminated, the predicate \{Q\} is true.

Notation: \(\frac{X}{Y} : \iff X \Rightarrow Y\)
Hoare Rule 1: Skip-Axiom

\[
\begin{align*}
\text{true} & \quad \{A\} \text{ skip } \{A\}
\end{align*}
\]
Hoare Rule 1: Skip-Axiom

\[ \text{true} \quad \{ A \} \text{skip} \quad \{ A \} \]

\text{skip} \text{ means the program with no commands.}
Hoare Rule 2: Axiom of Assignment

\[
\text{true} \quad \frac{\{A_{\beta/x}\} \ x:=\beta \ {A}}{}
\]
Hoare Rule 2: Axiom of Assignment

\[
{\text{true}} \\
{}_{A\beta/x} x:=\beta \{A\}
\]

$A_{\beta/x}$ is predicate $A$, but $x$ instead of $\beta$. 
Hoare Rule 3: Rule of Composition

\[ \{A\} \text{S1} \{B\} \land \{B\} \text{S2} \{C\} \]

\[ \{A\} \text{S1, S2} \{C\} \]
Hoare Rule 4: Rule of Conditional Branching

\[
\frac{\{A \land B\} \text{ } S1 \text{ } \{Q\} \land \{A \land \neg B\} \text{ } S2 \text{ } \{Q\}}{\{A\} \text{ if } B \text{ then } S1 \text{ else } S2 \text{ end if } \{Q\}}
\]
Hoare Rule 5: Rule of Iteration

\[
\begin{align*}
\{I \land B\} & S \{I\} \\
\{I\} & \text{while } B \text{ loop } S \text{ end loop } \{I \land \neg B\}
\end{align*}
\]
Hoare Rule 5: Rule of Iteration

\[
\begin{align*}
\{I \land B\} & \quad S \quad \{I\} \\
\{I\} & \quad \textbf{while } B \quad \textbf{loop } S \quad \textbf{end loop } \quad \{I \land \neg B\}
\end{align*}
\]

Such an I is called loop-invariant.
Hoare Rule 6: Rule of Consequence

\[ A \Rightarrow A' \land \{ A' \} S \{ B' \} \land B' \Rightarrow B \]

\( \{ A \} S \{ B \} \)
Proof of result_2(x,y) using Hoare

function result_2(x,y)  function result_2(x,y)
Proof of \textit{result}_2(x,y) using Hoare

\begin{verbatim}
function result_2(x,y)
    \{P: x \geq 0 \land y \geq 0, r := x + y\}
end
\end{verbatim}
Proof of $\text{result}_2(x,y)$ using Hoare

function $\text{result}_2(x,y)$

\begin{align*}
\{ & \mathbf{P}: x \geq 0 \land y \geq 0, \ r := x + y \} \\
\{ & \mathbf{I} \} \\
\{ & \mathbf{Q}: y = r \} \\
\end{align*}

while $x > 0$

while $x > 0$
Proof of \texttt{result\_2(x,y)} using Hoare

\begin{verbatim}
function result\_2(x,y) function result\_2(x,y)

\{ \texttt{P: } x \geq 0 \land y \geq 0, \, r := x + y \}
\{ \}
while x > 0 while x > 0
\{ \}
\{ I \land B \}
\{ I \}
\{ \}
\{ \}
\end{verbatim}

end
des

end
Proof of $\text{result}_2(x,y)$ using Hoare

function $\text{result}_2(x,y)$

while $x > 0$

$x = x - 1;$
$y = y + 1;$

end

{$P: x \geq 0 \land y \geq 0, \quad r := x + y}$
{$I}$

while $x > 0$
{$I \land B$}

$x = x - 1;$
$y = y + 1;$
{$I$}

end

{$I \land \neg B$} (Rule of Iteration)
Proof of \( \text{result}_2(x,y) \) using Hoare

function \( \text{result}_2(x,y) \)

\[
\text{while } x > 0 \\
\quad x = x - 1; \\
\quad y = y + 1; \\
\text{end}
\]

\[
\text{function } \text{result}_2(x,y) \\
\{ P: x \geq 0 \land y \geq 0, \ r := x + y \} \\
\{ I \} \\
\text{while } x > 0 \\
\quad \{ I \land B \} \\
\quad x = x - 1; \\
\quad y = y + 1; \\
\quad \{ I \} \\
\text{end} \\
\{ I \land \neg B \} \ (\text{Rule of Iteration}) \\
\{ Q: y = r \} \]
Proof of \texttt{result}_2(x,y) using Hoare

\begin{align*}
\text{function } \texttt{result}_2(x,y) & \quad \text{function } \texttt{result}_2(x,y) \\
\{ \mathbf{P} : x \geq 0 \land y \geq 0, \ r := x + y \} & \{ \mathbf{P} : x \geq 0 \land y \geq 0, \ r := x + y \} \\
\{ \mathbf{l} \} & \{ \mathbf{l} \} \\
\text{while } x > 0 & \quad \text{while } x > 0 \\
\{ \mathbf{l} \land \mathbf{B} \} & \{ \mathbf{l} \land \mathbf{B} \} \\
x = x - 1; & \quad x = x - 1; \\
y = y + 1; & \quad y = y + 1; \\
\{ \mathbf{l} \} & \{ \mathbf{l} \} \\
\text{end} & \quad \text{end} \\
\{ \mathbf{l} \land \neg \mathbf{B} \} & \{ \mathbf{l} \land \neg \mathbf{B} \} \quad (\text{Rule of Iteration}) \\
\{ \mathbf{Q} : y = r \} & \{ \mathbf{Q} : y = r \} \\
\text{return } y; & \quad \text{return } y;
\end{align*}
Proof of result\_2(x,y) using Hoare

\begin{align*}
\text{function result\_2(x,y)}
\quad & \text{function result\_2(x,y)} \\
\{P: x \geq 0 \land y \geq 0, r := x + y\} \\
\{I\} \\
\text{while } x > 0 \\
\quad x = x-1; \\
\quad y = y+1; \\
\text{end} \\
\{I \land B\} \\
\{I\} \\
\{I \land \neg B\} \quad (\text{Rule of Iteration}) \\
\{Q: y = r\} \\
\text{return } y;
\end{align*}

- \textbf{B: } x > 0 \text{ (condition in while-loop)}
Proof of \text{result}_2(x,y) \text{ using Hoare}

\begin{align*}
\text{function result}_2(x,y) & \quad \text{function result}_2(x,y) \\
\{ \text{P: } x \geq 0 \land y \geq 0, r := x + y \} & \quad \{ \text{P: } x \geq 0 \land y \geq 0, r := x + y \} \\
\{ \rule \} & \quad \{ \rule \} \\
\text{while } x > 0 & \quad \text{while } x > 0 \\
& \quad \{ \rule \land B \} \\
& \quad \{ \rule \land B \} \\
x = x - 1; & \quad x = x - 1; \\
y = y + 1; & \quad y = y + 1; \\
\text{end} & \quad \text{end} \\
\{ \rule \land \neg B \} \ (\text{Rule of Iteration}) & \quad \{ \rule \land \neg B \} \\
\{ \text{Q: } y = r \} & \quad \{ \text{Q: } y = r \} \\
\text{return } y; & \quad \text{return } y; \\
\bullet \text{ B: } x > 0 \text{ (condition in while-loop) } \Rightarrow \neg \text{ B: } \neg(x > 0) &
\end{align*}
Proof of result_2(x,y) using Hoare

function result_2(x,y):

\{P: x \geq 0 \land y \geq 0, \ r := x + y\}

while x > 0

\{I\}

x = x-1;
y = y+1;

\{I\}

end

\{I \land \neg B\} (Rule of Iteration)

\{Q: y = r\}

return y;

- **B**: \(x > 0\) (condition in while-loop) \(\Rightarrow \neg B: \neg(x > 0)\)

- loop-invariant:
Proof of \( \text{result}_2(x,y) \) using Hoare

function \( \text{result}_2(x,y) \)

\[
\{ \text{P: } x \geq 0 \land y \geq 0, \ r := x + y \} \\
\{ I \} \\
\text{while } x > 0 \\
\{ I \land \text{B} \} \\
x = x-1; \\
y = y+1; \\
\text{end} \\
\{ I \land \neg \text{B} \} \ (\text{Rule of Iteration}) \\
\{ \text{Q: } y = r \} \\
\text{return } y; \\
\]

- \( \text{B: } x > 0 \) (condition in while-loop) \( \Rightarrow \neg \text{B: } \neg(x > 0) \)
- loop-invariant: \( I: \ r = x + y \)
while $x > 0$
while \( x > 0 \)
\[
\{ \text{I: } r = x + y \land B: x > 0 \}
\]
while $x > 0$

\{ \text{Item 1} \}
\{ \text{Item 2} \}

$x = x - 1;$

$y = y + 1;$
while $x > 0$

{ Item 1 }  
\[ r = x + y \land B: x > 0 \]

{ Item 2 }  
\[ x = x - 1; \]

{ Item 2 }  
\[ y = y + 1; \]

{ Item 1 }  
\[ r = x + y, x \geq 0 \]
while $x > 0$

$\{|: r = x + y \land B: x > 0\}$

$\{\text{Item 1}\}$

$x = x-1;$

$\{\text{Item 2}\}$

$y = y+1;$

$\{|: r = x + y, x \geq 0\}$

- Rule of Assign.:
while $x > 0$

\{ \text{I}: r = x + y \land B: x > 0 \}\{ \text{Item 1} \}

$x = x-1;$

\{ \text{Item 2} \}

$y = y+1;$

\{ \text{I}: r = x + y, \ x \geq 0 \}\}

- Rule of Assign.: Item 2: $\{x + (y + 1), \ x \geq 0\}$
while $x > 0$

\{\text{Item 1}\} \quad x = x-1; \quad \{\text{Item 2}\} \quad y = y+1; \quad \{\text{Item 1}: r = x + y, \ x \geq 0\}

- Rule of Assign.: Item 2: \{x + (y + 1), \ x \geq 0\}
- Assign.: 
while $x > 0$
  
  \{ l: r = x + y \land B: x > 0 \} \\
  \{ l: \text{Item 1} \} \\
  x = x - 1; \\
  \{ l: \text{Item 2} \} \\
  y = y + 1; \\
  \{ l: r = x + y, x \geq 0 \} \\

- Rule of Assign.: \text{Item 2}: \{ x + (y + 1), x \geq 0 \} \\
- Assign.: \text{Item 1}: \{(x - 1) + (y + 1), x - 1 \geq 0\}
while $x > 0$

\[
\begin{align*}
\{ & \text{: } r = x + y \land B: x > 0 \\
\{ & \text{Item 1} \\
& x = x - 1; \\
\{ & \text{Item 2} \\
& y = y + 1; \\
\} & \text{: } r = x + y, x \geq 0
\end{align*}
\]

- Rule of Assign.: \text{Item 2: } \{(x - 1) + (y + 1), x - 1 \geq 0\}
- Assign.: \text{Item 1: } \{(x - 1) + (y + 1), x - 1 \geq 0\}
- In fact:
while $x > 0$
{
  $\{ \text{l: } r = x + y \land B: x > 0 \}$

  $\{ \text{Item 1} \}$
  $x = x - 1$;
  $\{ \text{Item 2} \}$
  $y = y + 1$;
  $\{ \text{l: } r = x + y, x \geq 0 \}$

  Rule of Assign.: $\text{Item 2}: \{(x - 1) + (y + 1), x - 1 \geq 0\}$
  Assign.: $\text{Item 1}: \{(x - 1) + (y + 1), x - 1 \geq 0\}$
  In fact: $\{(x - 1) + (y + 1), x - 1 \geq 0\}$
while $x > 0$

\{\text{Item 1}\: r = x + y \land B: x > 0\}

\{\text{Item 2}\}

$x = x - 1$;

\{\text{Item 2}\}

$y = y + 1$;

\{\text{Item 1}\: r = x + y, x \geq 0\}

- Rule of Assign.: \text{Item 2}: \{(x - 1) + (y + 1), x \geq 0\}
- Assign.: \text{Item 1}: \{(x - 1) + (y + 1), x - 1 \geq 0\}
- In fact: \{(x - 1) + (y + 1), x - 1 \geq 0\} \equiv \{\text{Item 1}\: r = x + y \land B: x > 0\}$
while \( x > 0 \)
\[ \{ I: \ r = x + y \land B: \ x > 0 \} \]
\[ \{ \text{Item 1} \} \]
\( x = x - 1; \)
\[ \{ \text{Item 2} \} \]
\( y = y + 1; \)
\[ \{ I: \ r = x + y, \ x \geq 0 \} \]

- Rule of Assign.: \{Item 2\}: \( \{ x + (y + 1), \ x \geq 0 \} \)
- Assign.: \{Item 1\}: \( \{ (x - 1) + (y + 1), \ x - 1 \geq 0 \} \)
- In fact: \( \{ (x - 1) + (y + 1), \ x - 1 \geq 0 \} \Rightarrow \{ I: \ x + y \land B: \ x > 0 \} \) because \( x - 1 \geq 0 \Leftrightarrow x > 0 \) for integer \( x \).
function result_2(x,y)
  \{P: x \geq 0 \land y \geq 0, \ r := x + y\}
  \{! : r = x + y\}
  \textbf{while} \ x > 0
  \{! : r = x + y \and \mathbf{B} : x > 0 \land y \geq 0\}
  \ x = x - 1;
  \ y = y + 1;
  \{! : r = x + y\}
end
function result_2(x,y)
    {\textbf{P:} \( x \geq 0 \land y \geq 0 \), \( r := x + y \)}
    {\textbf{I:} \( r = x + y \)}
    while \( x > 0 \)
        {\textbf{I:} \( r = x + y \) \land \textbf{B:} x > 0 \ y \geq 0 \}
        \( x = x-1; \)
        \( y = y+1; \)
        {\textbf{I:} \( r = x + y \)}
    end
    {\textbf{I:} \( r = x + y \) \land \neg \textbf{B:} \neg(x > 0) \}
function result_2(x,y)
  \{P: x \geq 0 \land y \geq 0, r := x + y\}
  \{I: r = x + y\}
  while x > 0
    \{I: r = x + y \land B: x > 0\}
    x = x-1;
    y = y+1;
    \{I: r = x + y\}
  end
  \{I: r = x + y \land \neg B: \neg(x > 0)\}
  \{Q: y = r\}
function result_2(x,y)
    \{P: x ≥ 0 ∧ y ≥ 0, r := x + y\}
    \{I: r = x + y\}
    while x > 0
        \{I: r = x + y ∧ B: x > 0 y ≥ 0\}
        x = x-1;
        y = y+1;
        \{I: r = x + y\}
    end
    \{I: r = x + y ∧ ¬B: ¬(x > 0)\}
    \{Q: y = r\}
    return y;
function result_2(x, y)
  \{P: x \geq 0 \land y \geq 0, \ r := x + y\}
  \{I: r = x + y\}
  while x > 0
    \{I: r = x + y \land B: x > 0 \ y \geq 0\}
    x = x-1;
    y = y+1;
    \{I: r = x + y\}
  end
  \{I: r = x + y \land \neg B: \neg(x > 0)\}
  \{Q: y = r\}
  return y;

\textbf{P: } x \geq 0 \land y \geq 0
function result_2(x,y)

\{\textbf{P} : x \geq 0 \land y \geq 0, \ r := x + y\}
\{\textbf{I} : r = x + y\}

\textbf{while} x > 0

\{\textbf{I} : r = x + y \land \textbf{B} : x > 0 \ y \geq 0\}
\ x = x - 1;
\ y = y + 1;
\ \{\textbf{I} : r = x + y\}
\textbf{end}

\{\textbf{I} : r = x + y \land \neg \textbf{B} : \neg(x > 0)\}
\{\textbf{Q} : y = r\}
return y;

\textbf{P} : x \geq 0 \land y \geq 0 \land \neg(x > 0)
function result_2(x,y)
    {P: \( x \geq 0 \land y \geq 0 \), \( r := x + y \)}
    {I: \( r = x + y \)}
    while \( x > 0 \)
        {I: \( r = x + y \) \( \land \) B: \( x > 0 \) \( y \geq 0 \)}
        x = x-1;
        y = y+1;
        {I: \( r = x + y \)}
    end
    {I: \( r = x + y \) \( \land \) \( \neg \) B: \( \neg (x > 0) \)}
    {Q: \( y = r \)}
    return y;

\( P: x \geq 0 \land y \geq 0 \) \( \land \) \( \neg (x > 0) \) \( \Rightarrow \) \( x = 0 \)
function result_2(x, y)
  {P: x ≥ 0 ∧ y ≥ 0, r := x + y}
  {I: r = x + y}
  while x > 0
    {I: r = x + y ∧ B: x > 0 y ≥ 0}
    x = x-1;
    y = y+1;
    {I: r = x + y}
  end
  {I: r = x + y ∧ ¬ B: ¬(x > 0)}
  {Q: y = r}
return y;

P: x ≥ 0 ∧ y ≥ 0 and ¬(x > 0) ⇒ x = 0
⇒ Q: y =
function result_2(x, y)
  {P: x ≥ 0 ∧ y ≥ 0, r := x + y}
  {I: r = x + y}
  while x > 0
    {I: r = x + y ∧ B: x > 0 y ≥ 0}
    x = x - 1;
    y = y + 1;
    {I: r = x + y}
  end
  {I: r = x + y ∧ ¬ B: ¬(x > 0)}
  {Q: y = r}
  return y;

P: x ≥ 0 ∧ y ≥ 0 and ¬(x > 0) ⇒ x = 0
⇒ Q: y = y + x
function result_2(x,y)

\[ \begin{align*}
\{P: x \geq 0 \land y \geq 0, \ r := x + y\} \\
\{I: r = x + y\} \\
\text{while } x > 0 \\
\quad \{I: r = x + y \land B: x > 0 \ y \geq 0\} \\
\quad x = x - 1; \\
\quad y = y + 1; \\
\quad \{I: r = x + y\} \\
\text{end} \\
\{I: r = x + y \land \neg B: \neg(x > 0)\} \\
\{Q: y = r\} \\
\text{return } y;
\end{align*} \]

\[\begin{align*}
P: x \geq 0 \land y \geq 0 \text{ and } \neg(x > 0) & \Rightarrow x = 0 \\
\Rightarrow Q: y = y + x = r \quad (I: r = x + y \text{ loop-invariant})
\end{align*}\]
function result_2(x, y)
    \{P: x \geq 0 \land y \geq 0, r := x + y\}
    \{I: r = x + y\}
    \text{while } x > 0
        \{I: r = x + y \land B: x > 0 \land y \geq 0\}
        x = x-1;
        y = y+1;
        \{I: r = x + y\}
    \text{end}
    \{I: r = x + y \land \neg B: \neg (x > 0)\}
    \{Q: y = r\}
    return y;

\begin{align*}
\text{P: } & x \geq 0 \land y \geq 0 \land \neg (x > 0) \implies x = 0 \\
\implies Q: & \text{ } y = y + x = r \text{ (I: } r = x + y \text{ loop-invariant)}
\end{align*}
Numerical Quadrature

Let $f : [a, b] \to \mathbb{R}$ be sufficiently smooth (e.g. $f \in C^2$).

The functional of the definite integral is given by

$$F(f, a, b) := \int_{a}^{b} f(x) \, dx$$

Numerical Quadrature means: Calculate an approximation for the numerical value of $F(f, a, b)$. 
Numerical Quadrature

Let \( f : [a, b] \to \mathbb{R} \) be sufficiently smooth (e.g. \( f \in C^2 \)).
Numerical Quadrature

Let $f : [a, b] \rightarrow \mathbb{R}$ be sufficiently smooth (e.g. $f \in C^2$).

The functional of the definite integral is given by
Numerical Quadrature

Let $f : [a, b] \to \mathbb{R}$ be sufficiently smooth (e.g. $f \in C^2$).

The functional of the definite integral is given by

$$F(f, a, b) := \int_a^b f(x) \, dx$$
Numerical Quadrature

Let $f : [a, b] \to \mathbb{R}$ be sufficiently smooth (e.g. $f \in C^2$).

The functional of the definite integral is given by

$$F(f, a, b) := \int_a^b f(x) \, dx$$

Numerical Quadrature means:
Let \( f : [a, b] \rightarrow \mathbb{R} \) be sufficiently smooth (e.g. \( f \in C^2 \)).

The functional of the definite integral is given by

\[
F(f, a, b) := \int_{a}^{b} f(x) \, dx
\]

Numerical Quadrature means:
Calculate an approximation for the numerical value of \( F(f, a, b) \).
The Trapezoidal Rule

Approximation with linear function:
The Trapezoidal-Rule

Approximation with linear function:

\[ F \approx T := (b - a) \cdot \frac{f(a) + f(b)}{2} \]
Dividing $[a, b]$ into smaller, equidistant intervals:

$$F \approx TS := (b - a) \frac{1}{n+1} \left[ f(a) + \sum_{k=1}^{n} f(x_k) + f(b) \right]$$

In the picture: $n = 4$

The errors

$$\Delta F = |F - T| \text{ or } \Delta F = |F - TS|$$

depend on the second derivative:

$$\Delta F \leq (b - a) \frac{1}{12} n^2 \cdot ||f''||_\infty$$
Dividing \([a, b]\) into smaller, equidistant intervals: \(\Rightarrow\) piecewise linear functions

\[
F \approx TS := \frac{b - a}{n + 1} \cdot \left[ \frac{f(a)}{2} + \sum_{k=1}^{n} f(x_k) + \frac{f(b)}{2} \right]
\]
Dividing \([a, b]\) into smaller, equidistant intervals: \(\Rightarrow\) piecewise linear functions

\[
F \approx TS := \frac{b - a}{n + 1} \cdot \left[ \frac{f(a)}{2} + \sum_{k=1}^{n} f(x_k) + \frac{f(b)}{2} \right]
\]

In the picture: \(n = 4\)
Dividing $[a, b]$ into smaller, equidistant intervals: $\Rightarrow$ piecewise linear functions

\[ F \approx TS := \frac{b - a}{n + 1} \cdot \left[ \frac{f(a)}{2} + \sum_{k=1}^{n} f(x_k) + \frac{f(b)}{2} \right] \]

In the picture: $n = 4$

The errors $\Delta F = |F - T|$ or $\Delta F = |F - TS|$ depend on the second derivative:
Dividing \([a, b]\) into smaller, equidistant intervals: \(\Rightarrow\) piecewise linear functions

\[
F \approx TS := \frac{b - a}{n + 1} \cdot \left[ \frac{f(a)}{2} + \sum_{k=1}^{n} f(x_k) + \frac{f(b)}{2} \right]
\]

In the picture: \(n = 4\)

The errors \(\Delta F = |F - T|\) or \(\Delta F = |F - TS|\) depend on the second derivative:

\[
\Delta F \leq \frac{(b - a)^3}{12 \cdot n^2} \cdot ||f''||_{\infty}
\]
Hierarchical Decomposition

To approximate $F(f, a, b)$ we start with the Trapezoidal-Rule:
Hierarchical Decomposition

To approximate $F(f, a, b)$ we start with the Trapezoidal-Rule:

$$F(f, a, b) \approx T(f, a, b) = (b - a) \cdot \frac{f(a) + f(b)}{2}$$
Hierachical Decomposition

To approximate $F(f, a, b)$ we start with the Trapezoidal-Rule:

$$F(f, a, b) \approx T(f, a, b) = (b - a) \cdot \frac{f(a) + f(b)}{2}$$

There is a residuum $S(f, a, b)$ with:
Hierarchical Decomposition

To approximate $F(f, a, b)$ we start with the Trapezoidal-Rule:

$$F(f, a, b) \approx T(f, a, b) = (b - a) \cdot \frac{f(a) + f(b)}{2}$$

There is a residuum $S(f, a, b)$ with:

$$F(f, a, b) = T(f, a, b) + S(f, a, b)$$
Now decompose $S(f, a, b)$ into a triangle $D$ with projected height
Now decompose $S(f, a, b)$ into a triangle $D$ with projected height

$$h = f \left( \frac{a + b}{2} \right) - \frac{f(a) + f(b)}{2}$$
Now decompose $S(f, a, b)$ into a triangle $D$ with projected height

$$h = f \left( \frac{a + b}{2} \right) - \frac{f(a) + f(b)}{2}$$

The area of $D$ is given by:
Now decompose $S(f, a, b)$ into a triangle $D$ with projected height

$$h = f\left(\frac{a + b}{2}\right) - \frac{f(a) + f(b)}{2}$$

The area of $D$ is given by:

$$D(f, a, b) = \frac{b - a}{2} \cdot h$$
The new residuum can be determined by using this idea recursively:

$$S(f, a, b) = D(f, a, b) + S(f, a, \frac{a+b}{2}) + S(f, \frac{a+b}{2}, b)$$
The new residuum can be determined by using this idea recursively:
The new residuum can be determined by using this idea recursively:

\[ S(f, a, b) = D(f, a, b) + S(f, a, \frac{a + b}{2}) + S(f, \frac{a + b}{2}, b) \]
Approximation via Basis Functions

If \( u : [a, b] \to \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \): 

\[
u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)
\]

Now we can write easily:

\[
F(f, a, b) \approx \int_{a}^{b} u(x) = \int_{a}^{b} \sum_{k=1}^{N} \alpha_k \Phi_k(x) = \sum_{k=1}^{N} \alpha_k \int_{a}^{b} \Phi_k(x)
\]
Approximation via Basis Functions

If $u : [a, b] \rightarrow \mathbb{R}$ is an approximation to $f$, then
Approximation via Basis Functions

If $u : [a, b] \to \mathbb{R}$ is an approximation to $f$, then

$$F(f, a, b) \approx F(u, a, b)$$
Approximation via Basis Functions

If \( u : [a, b] \rightarrow \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \):
Approximation via Basis Functions

If $u : [a, b] \rightarrow \mathbb{R}$ is an approximation to $f$, then

$$F(f, a, b) \approx F(u, a, b)$$

Let $u(x)$ be a linear combination of basis functions $\Phi_k(x)$:

$$u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)$$
Approximation via Basis Functions

If \( u : [a, b] \to \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \):

\[
u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)
\]

Now we can write easily:
Approximation via Basis Functions

If \( u : [a, b] \rightarrow \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \):

\[
u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)
\]

Now we can write easily:

\[
F(f, a, b) \approx \int_{a}^{b} u(x) =
\]
Approximation via Basis Functions

If \( u : [a, b] \to \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \):

\[
u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)
\]

Now we can write easily:

\[
F(f, a, b) \approx \int_a^b u(x) = \int_a^b \sum_{k=1}^{N} \alpha_k \Phi_k(x) =
\]
Approximation via Basis Functions

If \( u : [a, b] \rightarrow \mathbb{R} \) is an approximation to \( f \), then

\[
F(f, a, b) \approx F(u, a, b)
\]

Let \( u(x) \) be a linear combination of basis functions \( \Phi_k(x) \):

\[
u(x) = \sum_{k=1}^{N} \alpha_k \Phi_k(x)
\]

Now we can write easily:

\[
F(f, a, b) \approx \int_{a}^{b} u(x) = \int_{a}^{b} \sum_{k=1}^{N} \alpha_k \Phi_k(x) = \sum_{k=1}^{N} \alpha_k \int_{a}^{b} \Phi_k(x)
\]
Define „hat functions“ as basis functions via
Define „hat functions“ as basis functions via

$$\Phi_{n,i} = \Phi \left( \frac{x - x_{n,i}}{h_n} \right)$$
Define „hat functions“ as basis functions via

$$\Phi_{n,i} = \Phi \left( \frac{x - x_{n,i}}{h_n} \right)$$

$$\Phi(x) := \max\{1 - |x|, 0\}$$
Define „hat functions“ as basis functions via

\[ \Phi_{n,i} = \Phi \left( \frac{x - x_{n,i}}{h_n} \right) \]

- \( \Phi(x) := \max\{1 - |x|, 0\} \)
- mesh size \( h_n := 2^{-n} \)
Define „hat functions“ as basis functions via

\[ \Phi_{n,i} = \Phi \left( \frac{x - x_{n,i}}{h_n} \right) \]

- \( \Phi(x) := \max\{1 - |x|, 0\} \)
- **mesh size** \( h_n := 2^{-n} \)
- **grid points** \( x_{n,i} = i \cdot h_n \)
Define „hat functions“ as basis functions via

\[ \Phi_{n,i} = \Phi \left( \frac{x - x_{n,i}}{h_n} \right) \]

- \( \Phi(x) := \max\{1 - |x|, 0\} \)
- **mesh size** \( h_n := 2^{-n} \)
- **grid points** \( x_{n,i} = i \cdot h_n \)
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \rightarrow \mathbb{R}$ with $u(0) = u(1) = 0$. Then
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \to \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$
\Psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}
$$
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \to \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$\psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}$$

is a generator system for $V_N$ (but for $N > 1$ not a basis).
Let \( V_N \) be the space of the continuous, on grid \( h_n \) piecewise linear functions \( u : [0, 1] \to \mathbb{R} \) with \( u(0) = u(1) = 0 \). Then

\[
\psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}
\]

is a generator system for \( V_N \) (but for \( N > 1 \) not a basis).

We use the hierarchical basis:
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \rightarrow \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$
\Psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}
$$

is a generator system for $V_N$ (but for $N > 1$ not a basis). We use the hierarchical basis:

$$
\Psi^H_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n | i \text{ odd} \} 
$$
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \rightarrow \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$
\Psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}
$$

is a generator system for $V_N$ (but for $N > 1$ not a basis).

We use the hierarchical basis:

$$
\Psi^H_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \mid i \text{ odd} \}
$$

$$
W_n := \text{span}\{ \Phi_{n,i} \mid \alpha_{n,i} = 0 \text{ for all even } i \}
$$
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \rightarrow \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$\Psi_N := \bigcup_{n=1}^{N} \{\Phi_{n,i} : 1 \leq i < 2^n\}$$

is a generator system for $V_N$ (but for $N > 1$ not a basis).

We use the hierarchical basis:

$$\Psi_N^H := \bigcup_{n=1}^{N} \{\Phi_{n,i} : 1 \leq i < 2^n \mid i \text{ odd}\}$$

$$W_n := \text{span}\{\Phi_{n,i} \mid \alpha_{n,i} = 0 \text{ for all even } i\} \Rightarrow V_N = V_{N-1} \oplus W_N$$
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \to \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$
\Psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}
$$

is a generator system for $V_N$ (but for $N > 1$ not a basis).

We use the hierarchical basis:

$$
\Psi^H_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \mid i \text{ odd} \}
$$

$$
W_n := \text{span}\{ \Phi_{n,i} \mid \alpha_{n,i} = 0 \text{ for all even } i \} \Rightarrow V_N = V_{N-1} \oplus W_N
$$

$$
\Rightarrow V_N = \bigoplus_{n=1}^{N} W_n
$$
Let $V_N$ be the space of the continuous, on grid $h_n$ piecewise linear functions $u : [0, 1] \to \mathbb{R}$ with $u(0) = u(1) = 0$. Then

$$\Psi_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \}$$

is a generator system for $V_N$ (but for $N > 1$ not a basis).

We use the hierarchical basis:

$$\Psi^H_N := \bigcup_{n=1}^{N} \{ \Phi_{n,i} : 1 \leq i < 2^n \mid i \text{ odd} \}$$

$$W_n := \text{span}\{ \Phi_{n,i} \mid \alpha_{n,i} = 0 \text{ for all even } i \} \Rightarrow V_N = V_{N-1} \oplus W_N$$

$$\Rightarrow V_N = \bigoplus_{n=1}^{N} W_n \text{ (inductive argument with } V_1 = W_1)$$
The hierarchical basis for $W_1$, $W_2$ and $W_3$
Approximation
Let $v \in V_N$ be a vector:
Let $v \in V_N$ be a vector:

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n-1} \alpha_{n,i} \Phi_{n,i}(x)$$
Let $v \in V_N$ be a vector:

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n-1} \alpha_{n,i} \Phi_{n,i}(x)$$

The program \texttt{HierachicalBasis}(N) should convert $v(x)$ into the hierachical basis: ($N > 1$)
Let $v \in V_N$ be a vector:

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n-1} \alpha_{n,i} \Phi_{n,i}(x)$$

The program $\text{HierarchicalBasis}(N)$ should convert $v(x)$ into the hierarchical basis: ($N > 1$)

$$v(x) = \sum_{n=1}^{N} \sum_{i=1}^{2^n-1} \alpha'_{n,i} \Phi_{n,i}(x)$$

with $\alpha'_{n,i} = 0$ for all even $i$. 
Program HierarchicalBasis(N)

function HierarchicalBasis(N)
Program HierarchicalBasis(N)

function HierarchicalBasis(N)
    for n = N-1,...,1:
        \[
        a_{n+1,2i-1} = a_{n+1,2i}/2
        \]
        \[
        a_{n+1,2i} = 0
        \]
        \[
        a_{n,2i} = a_{n+1,2i}/2
        \]
function HierachicalBasis(N)
    for n = N-1,...,1:
        for i = 1,...,\(2^n - 1\):
            \(a_{n+1,2i-1} = a_{n+1,2i}/2\)
            \(a_{n,2i} = a_{n+1,2i} - a_{n+1,2i-1}/2\)
            \(a_{n,2i} = 0\)

To prove the correctness of \(\text{HierachicalBasis}(N)\), the program must be written in a form Hoare Calculation can handle with:
function HierachicalBasis(N)
    for n = N-1,...,1 :
        for i = 1,...,2^n − 1 :
            a_{n+1,2i−1} = a_{n+1,2i}/2
            a_{n+1,2i+1} = a_{n+1,2i}/2

To prove the correctness of HierachicalBasis(N), the program must be written in a form Hoare Calculation can handle with:
function HierachicalBasis(N) 
  for n = N-1,...,1:
    for i = 1,...,2^n − 1:
      \( a_{n+1,2i-1} = a_{n+1,2i}/2 \)
      \( a_{n+1,2i+1} = a_{n+1,2i}/2 \)
      \( a_{n,i} = a_{n+1,2i} \)
function HierarchicalBasis(N)
    for n = N−1,...,1:
        for i = 1,...,2^n − 1:
            a_{n+1,2i−1} = a_{n+1,2i}/2
            a_{n+1,2i+1} = a_{n+1,2i}/2
            a_{n,i} + = a_{n+1,2i}
            a_{n+1,2i} = 0

function HierachicalBasis(N)
    for n = N-1,...,1:
        for i = 1,...,2^n − 1:
            a_{n+1,2i−1} = a_{n+1,2i}/2
            a_{n+1,2i+1} = a_{n+1,2i}/2
            a_{n,i} = a_{n+1,2i}
            a_{n+1,2i} = 0

To prove the correctness of HierachicalBasis(N), the program must be written in a form Hoare Calculation can handle with:
function HierarchicalBasis_Hoare(N)
function HierarchicalBasis_Hoare(N)
    n = N-1
    while n \neq 0

function HierarchicalBasis_Hoare(N)
    n = N-1
    while n \neq 0
        i = 1
        while i \neq 2^n
            [Equations not shown]
            i = i+1
            n = n-1
function HierarchicalBasis_Hoare(N)
    n = N-1
    while n \neq 0
        i = 1
        while i \neq 2^n
            a_{n+1,2i-1} = a_{n+1,2i-1} - a_{n+1,2i}/2
            a_{n+1,2i+1} = a_{n+1,2i+1} - a_{n+1,2i}/2
            a_{n,i} = a_{n,i} + a_{n+1,2i}
            a_{n+1,2i} = 0
        i = i + 1
    n = n - 1
function HierarchicalBasis_Hoare(N)
    n = N-1
    while n ≠ 0
        i = 1
        while i ≠ 2^n
            a_{n+1,2i-1} = a_{n+1,2i-1} - a_{n+1,2i}/2
            a_{n+1,2i+1} = a_{n+1,2i+1} - a_{n+1,2i}/2
            a_{n,i} = a_{n,i} + a_{n+1,2i}
            a_{n+1,2i} = 0
        i = i+1
    n = n-1
Why to use Hierarchical Basis

Adaptive stop criterion (via projected high):
\[ \alpha_n, i < \epsilon \Rightarrow \text{STOP} \]

The global error can be estimated by
\[ \Delta F \leq \epsilon (b - a) \]

For second-degree polynomials:
\[ S = \frac{4}{3} D \text{(Simpson's Rule)} \]

For high-dimensional functions (dimension \( d \)) the memory requirements
\[ M \propto N^d \]

\( N = \dim V \)

⇒ exponential increasing with \( d \)

With hierarchical basis:
\[ M \propto N \cdot (\ln N)^{d - 1} \]

⇒ New problem:
Program code very complicated!

How to be sure, there are no deficiencies?

⇒ Hoare Calculation!
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): 

\[ \alpha_n, i < \epsilon \Rightarrow \text{STOP} \]

The global error can be estimated by 

\[ \Delta F \leq \epsilon (b - a) \]

For second-degree polynomials:

\[ S = \frac{4}{3} D \] (Simpson's Rule)

For high-dimensional functions (dimension \( d \)) the memory requirements \( M \propto N^d \) (\( N = \dim V_N \)) \( \Rightarrow \) exponential increasing with \( d \)

With hierarchical basis:

\[ M \propto N \cdot (\ln N)^{d-1} \Rightarrow \text{New problem: Program code very complicated!} \]

How to be sure, there are no deficiencies? \( \Rightarrow \) Hoare Calculation!
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon$
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials:
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow$ STOP
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpsons-Rule)
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3} D$ (Simpsons-Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d$ ($N = \dim V_N$)
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpsons-Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d$ ($N = \dim V_N$)
$\Rightarrow$ exponential increasing with $d$
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): \( \alpha_{n,i} < \epsilon \Rightarrow \text{STOP} \)
- The global error can be estimated by \( \Delta F \leq \epsilon (b - a) \)
- For second-degree polynomials: \( S = \frac{4}{3} D \) (Simpson's Rule)

For high-dimensional functions (dimension \( d \)) the memory requirements \( M \propto N^d \) \((N = \text{dim } V_N)\)
\( \Rightarrow \text{exponential} \) increasing with \( d \)

With hierarchical basis:
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): \( \alpha_{n,i} < \epsilon \Rightarrow \text{STOP} \)
- The global error can be estimated by \( \Delta F \leq \epsilon (b - a) \)
- For second-degree polynomials: \( S = \frac{4}{3}D \) (Simpsons-Rule)

For high-dimensional functions (dimension \( d \)) the memory requirements \( M \propto N^d \) \((N = \text{dim } V_N)\)
\[ \Rightarrow \text{exponential increasing with } d \]
With hierarchical basis: \( M \propto N \cdot (\ln N)^{d-1} \)
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow$ STOP
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpsons-Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d$ ($N = \dim V_N$)
$\Rightarrow$ exponential increasing with $d$
With hierarchical basis: $M \propto N \cdot (\ln N)^{d-1}$
$\Rightarrow$ New problem:
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpsons-Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d$ ($N = \text{dim } V_N$)

$\Rightarrow$ exponential increasing with $d$

With hierarchical basis: $M \propto N \cdot (\ln N)^{d-1}$

$\Rightarrow$ New problem: Program code very complicated!
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow \text{STOP}$
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpsons-Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d \ (N = \dim V_N)$
\Rightarrow \text{exponential} increasing with $d$
With hierarchical basis: $M \propto N \cdot (\ln N)^{d-1}$
\Rightarrow New problem: Program code very complicated! How to be sure, there are no deficiencies?
Why to use Hierarchical Basis

- Adaptive stop criterion (via projected high): $\alpha_{n,i} < \epsilon \Rightarrow$ STOP
- The global error can be estimated by $\Delta F \leq \epsilon (b - a)$
- For second-degree polynomials: $S = \frac{4}{3}D$ (Simpson's Rule)

For high-dimensional functions (dimension $d$) the memory requirements $M \propto N^d$ ($N = \text{dim } V_N$)
⇒ exponential increasing with $d$
With hierarchical basis: $M \propto N \cdot (\ln N)^{d-1}$
⇒ New problem: Program code very complicated! How to be sure, there are no deficiencies?
⇒ Hoare Calculation!
References

- Michael Gellner: Der Umgang mit dem Hoare-Kalkül zur Programmverifikation
- Volker Claus: Einführung in die Informatik 2005/06 - Kapitel 7: Semantik von Programmen
- Peter Heinig: Program Verification using Hoare Logic - An Introduction
- Michael Bader, Stefan Zimmer: Hierarchische Zerlegung (eindimensional)
Thank you for your attention!