https://fpanalysistools.org/ISC26/

FPChecker

Tutorial

Compiler-Assisted Floating-Point Error Analysis and Profiling with FPChecker

Ignacio Laguna

ISC High Performance

LLNL-PRES-2012776

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Previous tutorials

https://fpanalysistools.org

- SC'25
- SC'24
- SC'19
- PEARC'19

Have any of you attended an earlier tutorial?

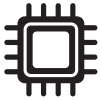
Learning Objectives

Identify numerical instabilities

— Isolate floating-point exceptions (**NaN** and **Infinity**) and **cancellation**

Characterize dynamic range and rounding error

— Generate ***per-line rounding error reports*** and ***dynamic range profiles***



Automate analysis via compiler instrumentation and FPChecker

— Lear to compile and run code with the tool

Implement data-driven mixed-precision strategies

— Use numerical error reports to assign lower or higher precision: **FP32** or **FP64**

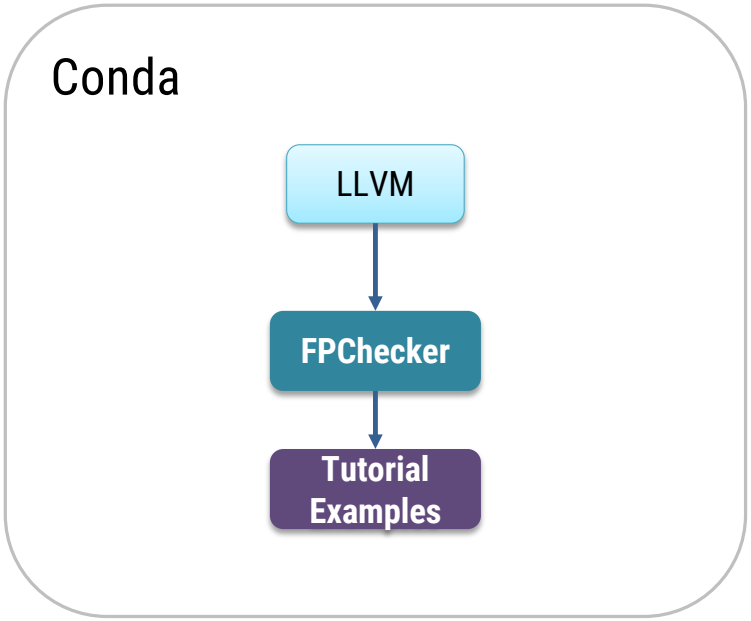
Gain experience profiling real-world HPC libraries

Accessing Slides

https://fpanalysistools.org/ISC26/

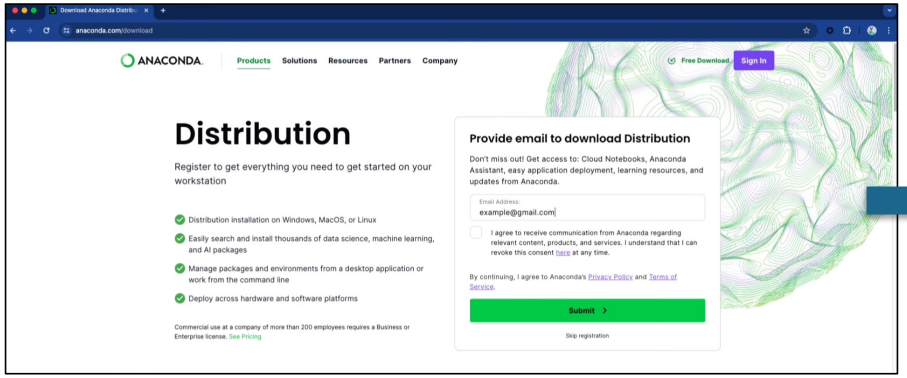
Install Conda
Only if you wan to run the examples in your laptop

- Why Conda?
  - — **Easy to deploy clang/LLVM** for the tutorial (a couple of minutes)
- Alternative method: **build clang/LLVM from source**
  - — Not recommended for the tutorial
  - — Best approach for production installation

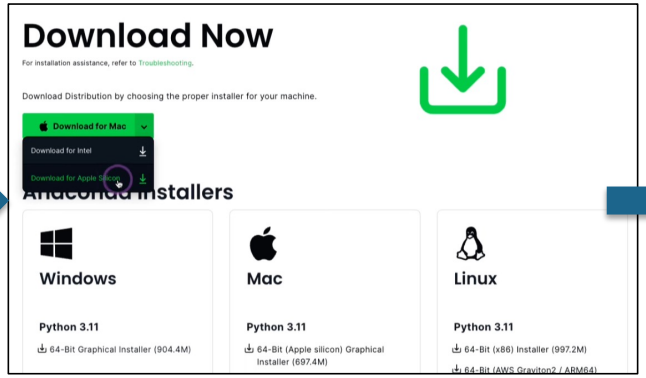


Steps to Install Conda in Mac

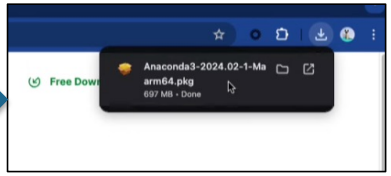
Got https://www.anaconda.com/ -> Download



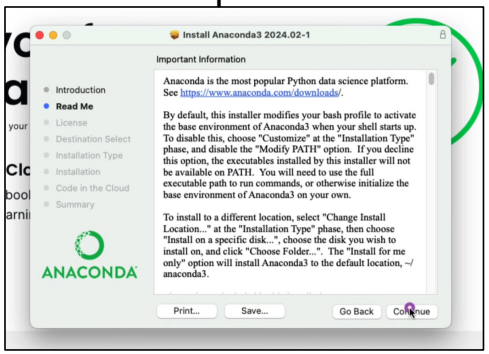
Download



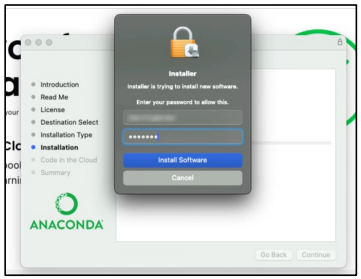
Open Installer



Follow the steps



Provide password if asked



In the terminal, you should see the word **base**:

```
(base) [user@system] $
```

Access to AWS instances

1. Add your name to a username here: https://fpchecker.org/usernames

2.
   - Connect to the AWS instance via ssh
   - Ask instructor for the password

```
$ ssh user[N]@IP_ADDRESS
```

IP Addresses:

```
35.157.176.43
51.102.86.169
52.58.211.80
```

Agenda

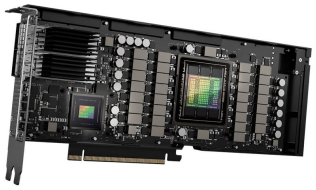
Module	Duration	Time (AM)
☑ Welcome	10	9:00 – 9:10
Motivation & Overview	5	9:10 – 9:15
Floating-Point Basics	20	9:15 – 9:35
FPChecker's Workflow	15	9:35 – 9:50
Installation, Access to Examples	15	9:50 – 10:05
Guided Practice 1: *Exceptions (linear solver)*	15	10:05 – 10:20
Guided Practice 2: *Dynamic Range Analysis*	15	10:20 – 10:35
Guided Practice 3: *Simple Mixed-Precision Example*	25	10:35 – 11:00
Break	30	11:00 – 11:30
Guided Practice 4: *Mixed-Precision in CG*	20	11:30 – 11:50
Demo: *Single-Line Errors in CG (Cancellation)*	10	11:50 – 12:00
Independent Practice: *Hypre linear solver package*	30	12:00 – 12:30
Recap & Next Steps	30	12:30 – 1pm

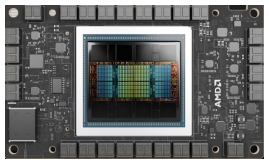
Module 1

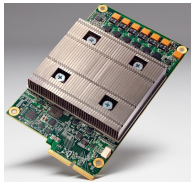
Motivation & Overview

- Problems the tool solves
- Need for lower & mixed precision
- FPChcker's overview

The Need for “Low-Precision” in HPC



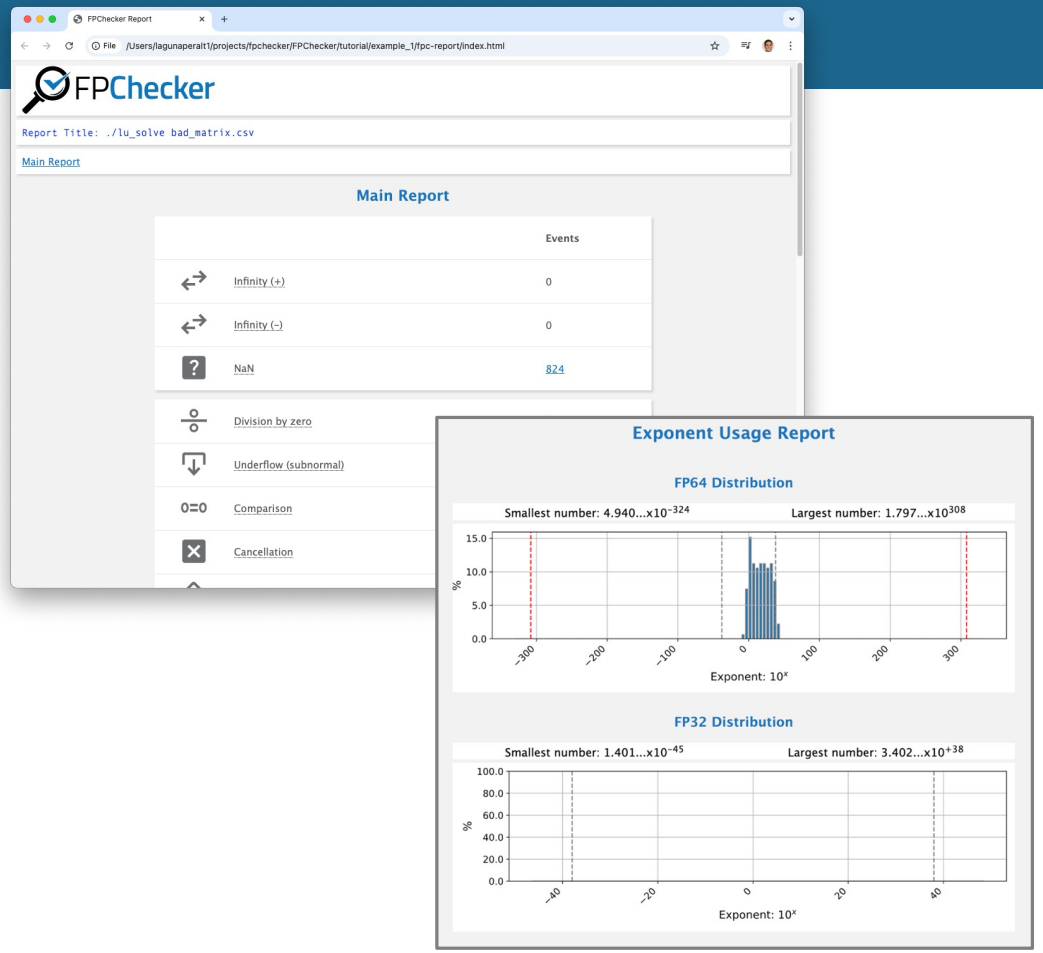




- Architectures promote lower precision formats:
  - FP16, FP8
  - Bfloat16
  - TensorFlowt-32
- We need tools to understand **mixed-precision** codes
  - Most codes use FP64
- Important metrics to understand:
  - Dynamic range of FP32, FP16
  - Rounding error accumulation

FPChecker Overview

- Detects floating-point exceptions
  - NaN, Infinity
- Shows impacted *lines of code*
- Shows other "code smells"
  - Cancellations, underflows
- Analyzes dynamic range
  - Is FP32 or FP64 enough?
- Computes accumulated rounding error
  - Helpful for *mixed-precision*
- Works with compiler instrumentation
  - Relies on Clang/LLVM
- Documentation: https://fpchecker.org/



Module 2

Floating-Point Basics

- Floating point representations
- Rounding error & relative rounding error
- How is relative error bounded?
- Cancellation
- Machine epsilon
- Exceptions

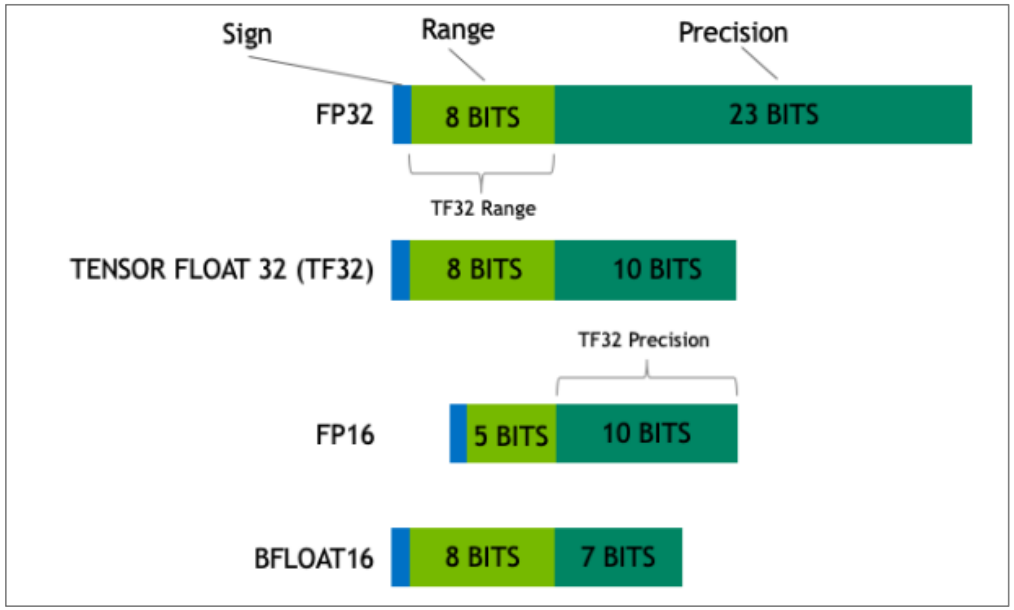
Floating-Point Representations

Standard formula for floating-point representations

$$\text{Value} = (-1)^{\text{sign}} \times \text{Mantissa} \times 2^{\text{Exponent}}$$

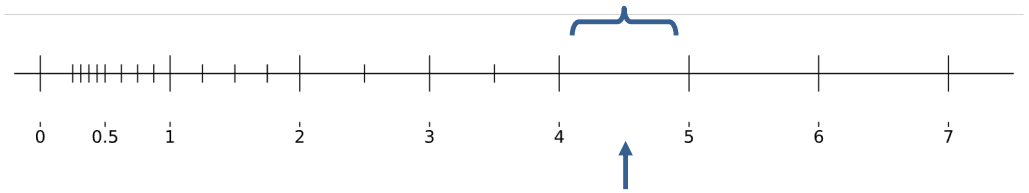
Exponent: determines the ***range***
Mantissa: determines the ***precision***

Bits for different formats



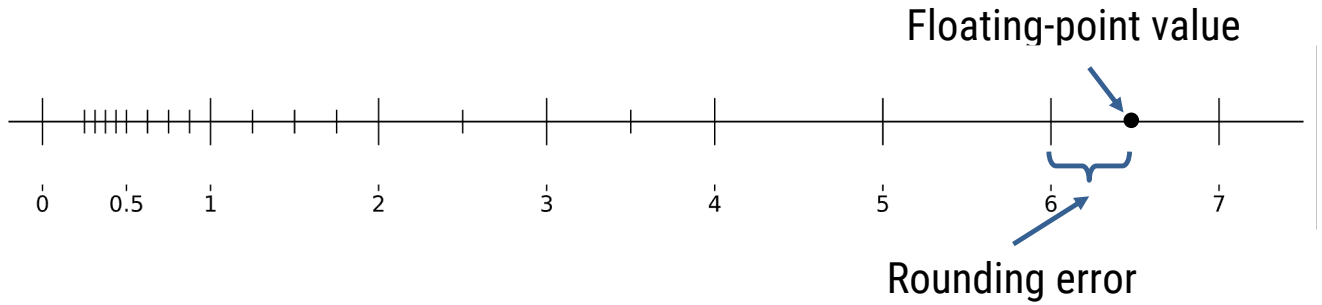
From https://developer.nvidia.com/

Significant Figures



- FP32 can roughly track **7 decimal digits** of precision.
- For a number 1.234567
  - The "**steps**" between it and the next representable number are tiny (0.000001)
- For a large number like 1,234,567,000
  - The next number it can represent might be 1,234,568,000

Rounding Error



Rounding error: *The distance between where you **wanted** to be and where you **landed***

Example: $\pi = 3.14159265358979323846...$

FP32	About **7 dig**its of precision	$\pi = 3.1415927$
FP64	About **15-17 digits** of precision	$\pi = 3.1415926535897931$

Rounding error can accumulate!

Representation Error

- Representation error:

  Mathematically: `0.1 + 0.2 = 0.3`

  In floating-point: `0.30000000000000004`

Python

```
>>>
>>> a = 0.1
>>> b = 0.2
>>> a+b
0.30000000000000004
>>>
```

Relative Rounding Error

$$\text{Relative Error} = \frac{|\text{Experimental} - \text{Theoretical}|}{|\text{Theoretical}|}$$

Absolute error is huge, but **relative error** is the same

Real Math Value	Computer Representation (7-digit limit)	Rounding Error	Relative Rounding Error
1.23456**89**	1.234568	+0.00000011	8.91×10^{-8}
123456**89**.0	123456800.0	+11.0	8.91×10^{-8}

- IEEE 754: relative error is bounded by a constant, **Machine Epsilon** (ϵ)
- Regardless of how big the number is
- For FP32: $\epsilon \approx 10^{-7}$
- For FP64: $\epsilon \approx 10^{-16}$

Precision Loss

Python

```
>>>
>>> (10e8 + 1e-8) - 10e8
0.0
>>>
```

- Mathematically the answer is 10^{-8}

- The computer must add:

- 10^{8} (a massive number)
- 10^{-8} (a tiny fraction)

- In FP32, we only have about **7 decimal digits** of precision to work with.
- When you try to add these two, you are essentially asking the computer to store:

100,000,000.00000001

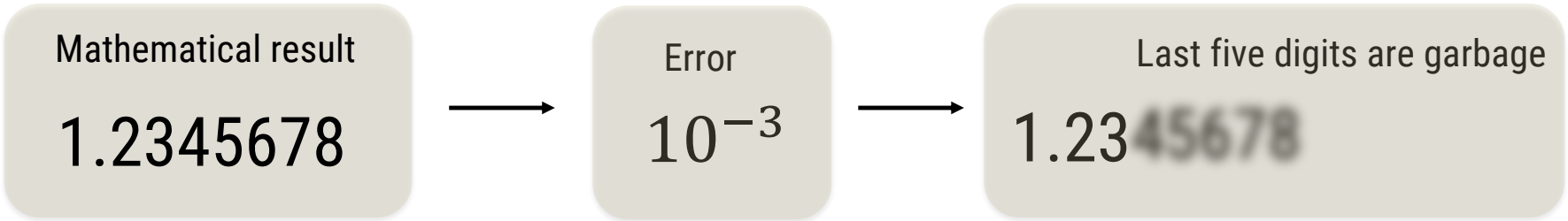
- **There are not enough bits to store it**, so it must be rounded to 10^{8}

- The subtraction becomes:

$$(10^{8}) - 10^{8} = 0$$

Significant Figures:
How many digits can we actually trust?

- Relative rounding error directly measures **how many digits** we can trust
- If relative error is 10^{-3}
  - You have roughly **3 decimal digits** you can trust



For Mixed Precision

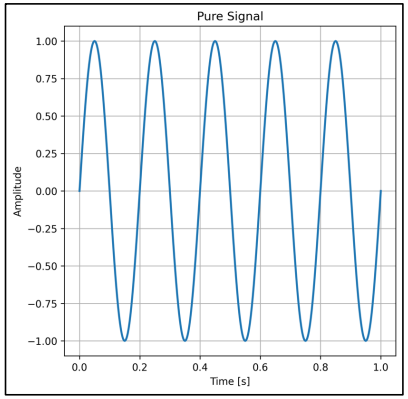
- *Increase precision* if relative error is **large**
- *Keep/decrease precision* is relative error is **small**

Signal-to-Noise Ratio Analogy:
When to increase precision?

Think of your calculation as a ***radio broadcast***

- **The Signal:** The actual value you are trying to compute
- **The Noise:** The rounding error introduced

FP32 precision

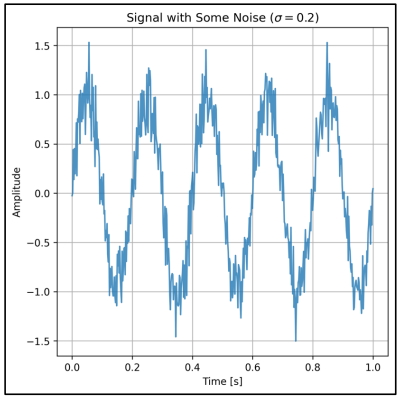


- Original signal
- Intended math

Case 1

Relative error is small:

- Signal is **much louder** than the noise.
- Your result is *"mostly correct"*

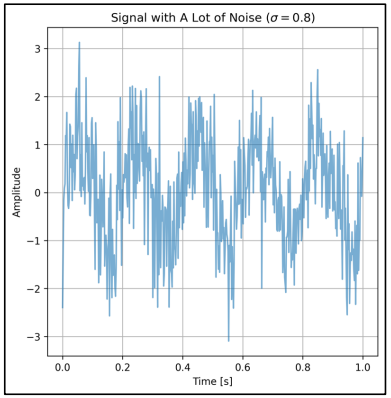


Keep in FP32

Case 2

Relative error is large:

- The "*noise*" has completely **drowned out** "*signal*"
- Digits you are seeing are essentially random
- Values no longer represent the math you intended



Increase to FP64

Cancellation Explained

- Occurs when ***subtracting two nearly identical numbers***
- It's the most aggressive way to lose precision
- Can delete entire result in a single subtraction
- Primary reason scientific simulations *"blow up"*

Suppose:

```
x = 1.0000000000000001
y = 1.0000000000000002
```

Mathematically: `y-x = 0.0000000000000001`

In floating-point: `y-x = 2.2204460492503130e-16`

Large relative error!

Because of Cancellation Relative Error can be 100% or More

$$Relative\ error = \frac{|Approximation\ - True\ Value|}{|True\ Value|}$$

- Subtracting two nearly equal numbers (*cancellation*)

a_{true} = 1234567.891
b_{true} = 1234567.890

$c_{true} = a_{true} - b_{true}$
c_{true} = 0.001

- **Single precision** had about 7 decimal digits of precision
- Suppose numbers **would be rounded**:

a = 1234568.0
b = 1234568.0
c = a − b = 0.0

⇨ $Relative\ error = \frac{|0.0\ - 0.001|}{|0.001|} = 1.0$

Floating-Point Exceptions

- Exceptions are the ***smoke before the fire***
- Indicate a computation is about to **"blow up"**

Type	Explanation
Division by Zero	• Result in **INF** (infinity)
Overflow	• The result is too large to be represented • Results in **INF**
Invalid Operation	• Math is undefined • Examples: `sqrt(-1), 0/0` • Result in **NaN**
Underflow	• Result is approaching zero • Smaller than smallest normal number
Inexact	• Result must be rounded • We don't worry about this one in HPC

- Most important
- Result in **INF** or **NAN**

When to Worry about Exceptions?

- Exceptions propagate as a **virus**
  - Once a **NaN** or **INF** enters a calculation, almost every subsequent operation will also result in **NaN** or **INF**
- By default, CPUs don't stop the program when these happen
  - FPChecker will detect them for you
- Dynamic range is reduced in lower precision formats: FP32, FP16, …
  - Likely to get more **overflows**

Module's Review

- Relative rounding error is bounded by machine epsilon
  - For FP32: $\epsilon \approx 10^{-7}$
  - For FP64: $\epsilon \approx 10^{-16}$
- **Increase** precision if relative error is **large**
- Cancellation relative error is **not bounded** by machine epsilon
  Cancellation error could be 100% or more
- Exceptions can introduce **INF** and **NaN**
  Exceptions matter when reducing precision

Questions

Agenda

	Module	Duration	Time (AM)
☑	Welcome	10	9:00 – 9:10
☑	Motivation & Overview	5	9:10 – 9:15
☑	Floating-Point Basics	20	9:15 – 9:35
	FPChecker's Workflow	15	9:35 – 9:50
	Installation, Access to Examples	15	9:50 – 10:05
	Guided Practice 1: *Exceptions (linear solver)*	15	10:05 – 10:20
	Guided Practice 2: *Dynamic Range Analysis*	15	10:20 – 10:35
	Guided Practice 3: *Simple Mixed-Precision Example*	25	10:35 – 11:00
	Break	30	11:00 – 11:30
	Guided Practice 4: *Mixed-Precision in CG*	20	11:30 – 11:50
	Demo: *Single-Line Errors in CG (Cancellation)*	10	11:50 – 12:00
	Independent Practice: *Hypre linear solver package*	30	12:00 – 12:30
	Recap & Next Steps	30	12:30 – 1pm

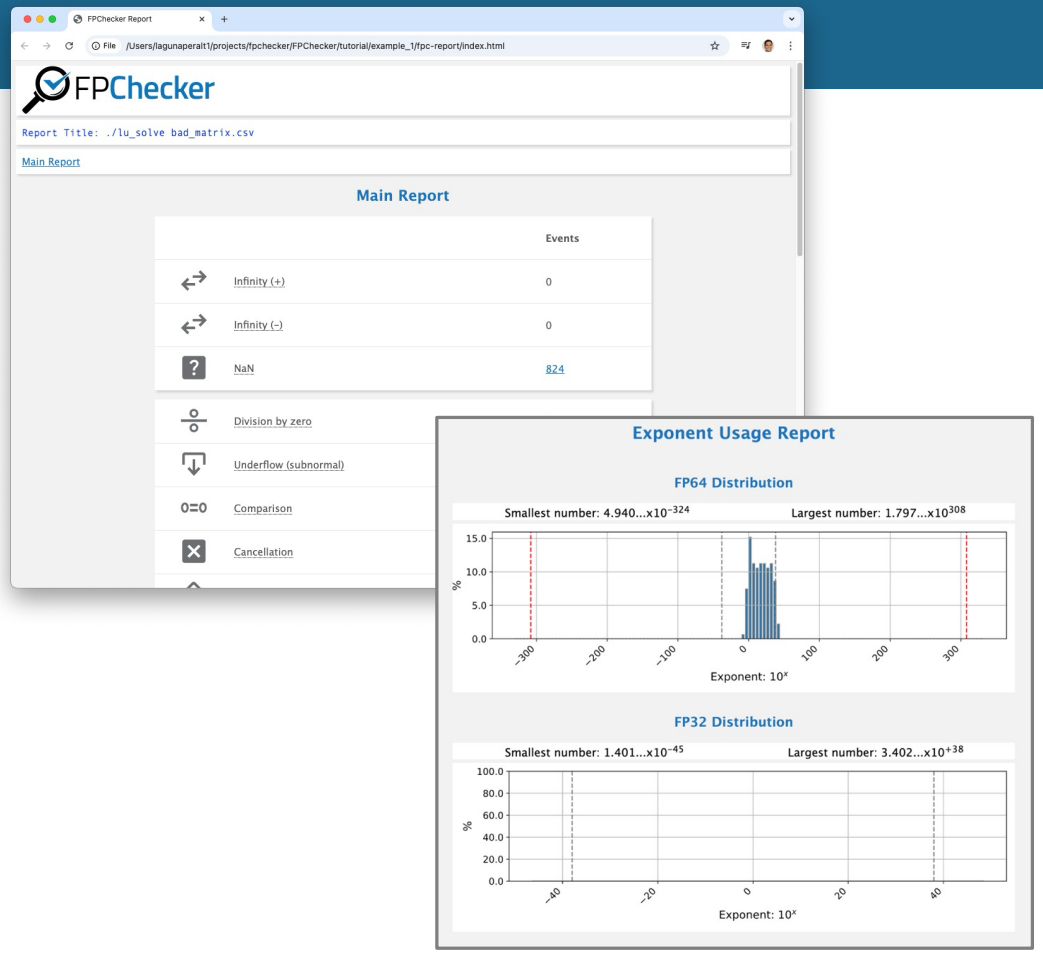
Module 3

FPChecker's Workflow

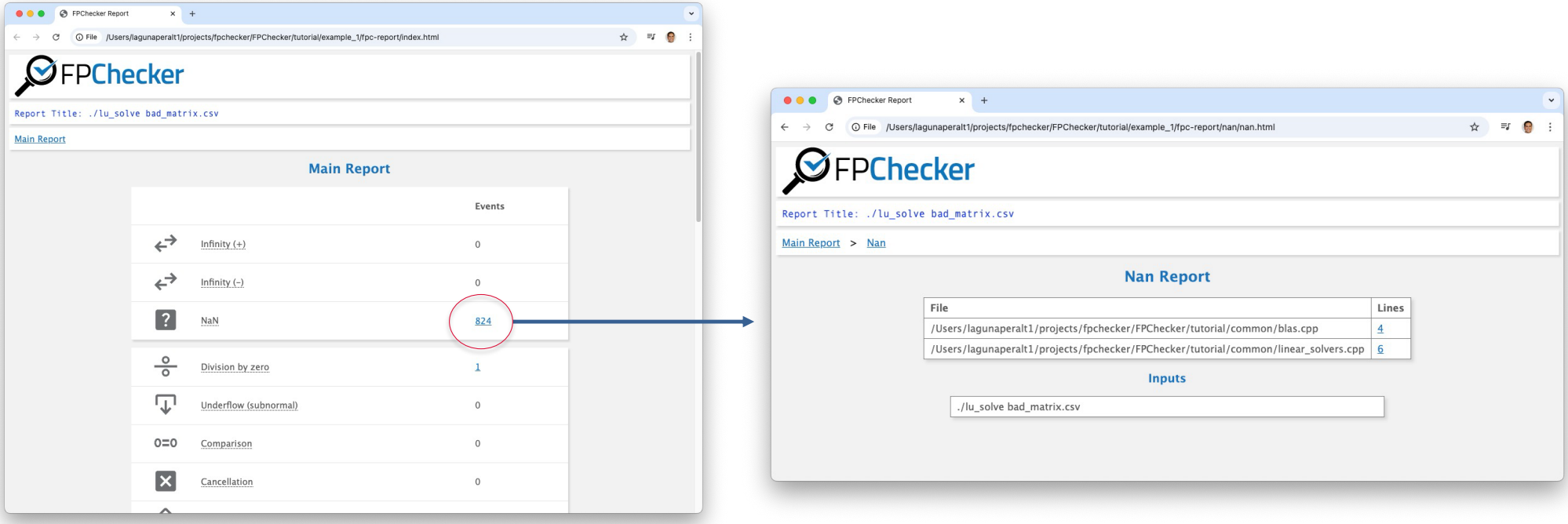
- Tool's workflow
- Compiler instrumentation concepts
- Keeping track of exceptions and rounding error
- Error reports (per line); what does it mean?

FPChecker Overview

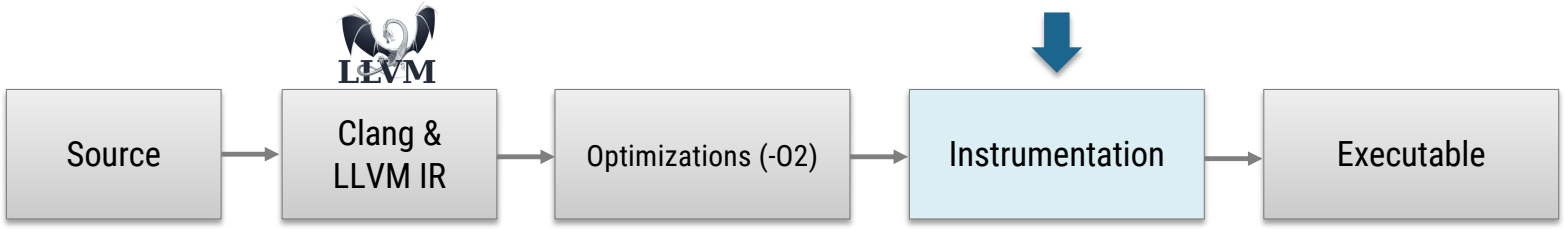
- Detects floating-point exceptions
  - NaN, Infinity
- Shows impacted *lines of code*
- Shows other "code smells"
  - Cancellations, underflows
- Analyzes dynamic range
  - Is FP32 or FP64 enough?
- Computes accumulated rounding error
  - Helpful for *mixed-precision*
- Works with compiler instrumentation
  - Relies on Clang/LLVM
- Documentation: https://fpchecker.org/



Report Example



LLVM Instrumentation Process



1. Make changes to your Makefile or build system (Cmake)
   - Example: add instrumentation pass to your CFLAGS and/or CXXFLAGS
   - See next slides
2. Compile code (*instrument code*)
3. Run executable
4. Create report

Two Ways to Instrument Applications

Method 1

Pass extra flags to the compiler

```
$ clang++ -c file.cpp –O2 \
-g \
-include /path/.../Runtime_cpu.h \
-fpass-plugin=/path/.../libfpchecker_cpu.dylib
```

To find the appropriate flags:

```
$ fpchecker-show
```

Method 2

Use the FPChecker compiler wrappers:

- `clang-fpchecker`
- `clang++-fpchecker`
- `mpicc-fpchecker`
- `mpicxx-fpchecker`

```
$ FPC_INSTRUMENT=1 clang++-fpchecker -c file.cpp –O2
```

Env variables must be enabled to instrument:

- FPC_INSTRUMENT, **or**
- FPC_INSTRUMENT_ERR_TRACKING

Two Classes of Env Variables: *Compile-time & Run-time*

✓ Covered in the Tutorial

Compile-time Variables

Variable	Type	Description
✓ FPC_INSTRUMENT	Compile-time	Instruments the application for exceptions/event checking
✓ FPC_INSTRUMENT_ERR_TRACKING	Compile-time	Instruments the application for rounding error tracking
FPC_ANNOTATED	Compile-time	Indicates that the program is annotated

Run-time Variables

Variable	Type	Description
✓ FPC_EXPONENT_USAGE	Run-time	Profiles exponent usage for FP32/FP64
FPC_TRAP_INFINITY_POS	Run-time	Program exits when Infinity positive is found
FPC_TRAP_INFINITY_NEG	Run-time	Program exits when Infinity negative is found
FPC_TRAP_NAN	Run-time	Program exits when NaN is found
FPC_TRAP_DIVISION_ZERO	Run-time	Program exits when division-by-zero is found
FPC_TRAP_CANCELLATION	Run-time	Program exits when cancellation is found
FPC_TRAP_COMPARISON	Run-time	Program exits when Comparison is found
FPC_TRAP_UNDERFLOW	Run-time	Program exits when underflow is found
FPC_TRAP_LATENT_INF_POS	Run-time	Program exits when Latent Infinity positive is found
FPC_TRAP_LATENT_INF_NEG	Run-time	Program exits when Latent Infinity negative is found
FPC_TRAP_LATENT_UNDERFLOW	Run-time	Program exits when Latent Underflow is found

Dynamic range analysis
from FP64 → FP32 → FP16 …

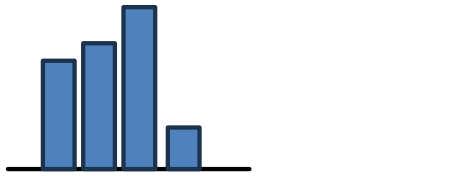
- Identifying **underflow** and **overflow** risks:
  - See if values fit within the format
    - If values ranges between 10^{-20} and 10^{20}, safely use FP32
    - FP32 covers roughly: 1.18×10^{-38} and 1.18×10^{38}

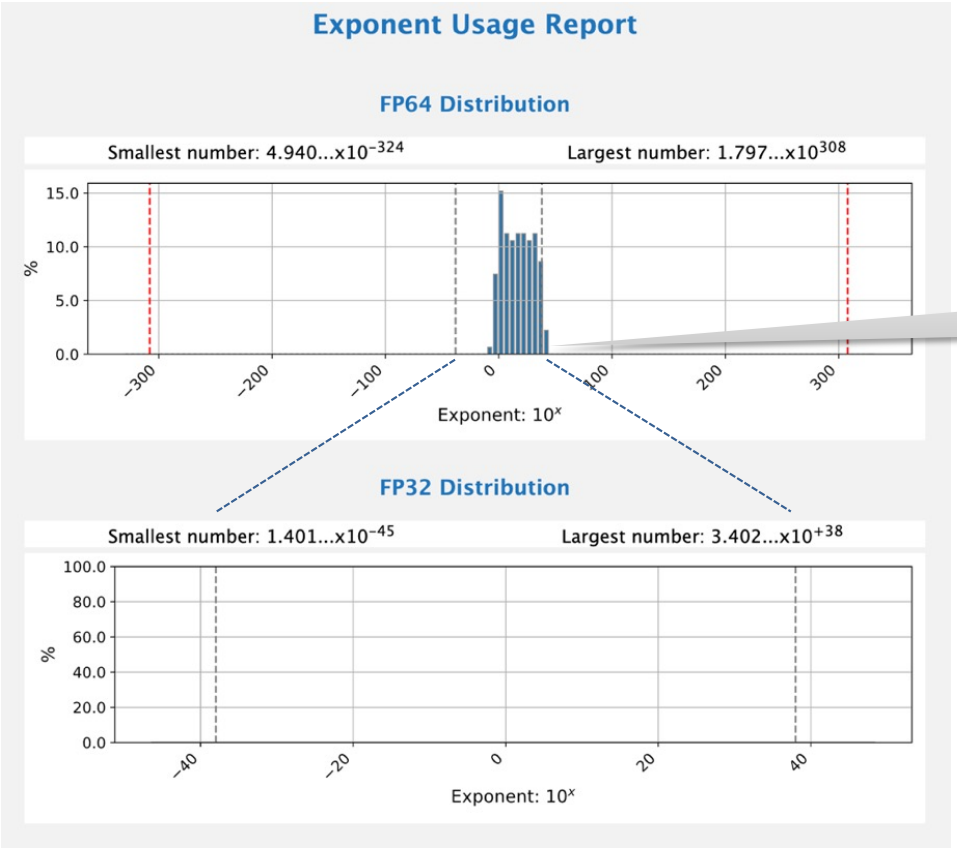
- **Targeted** precision reduction
  - Not every part of your code needs the same level of precision

Exponent Usage Analysis:
Dynamic Range of Computed Numbers

1. Observes computed floating-point numbers
2. FPChecker builds a **histogram** of exponent values

$$variable = 1.23 \times 10^{200}$$





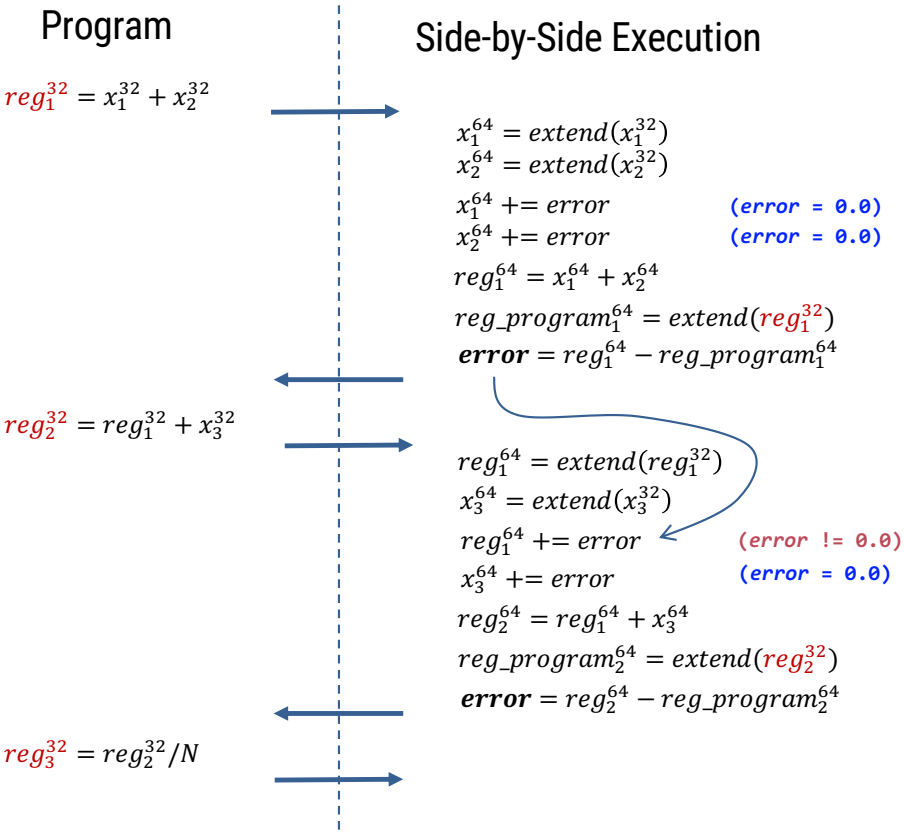
Out of range for FP32

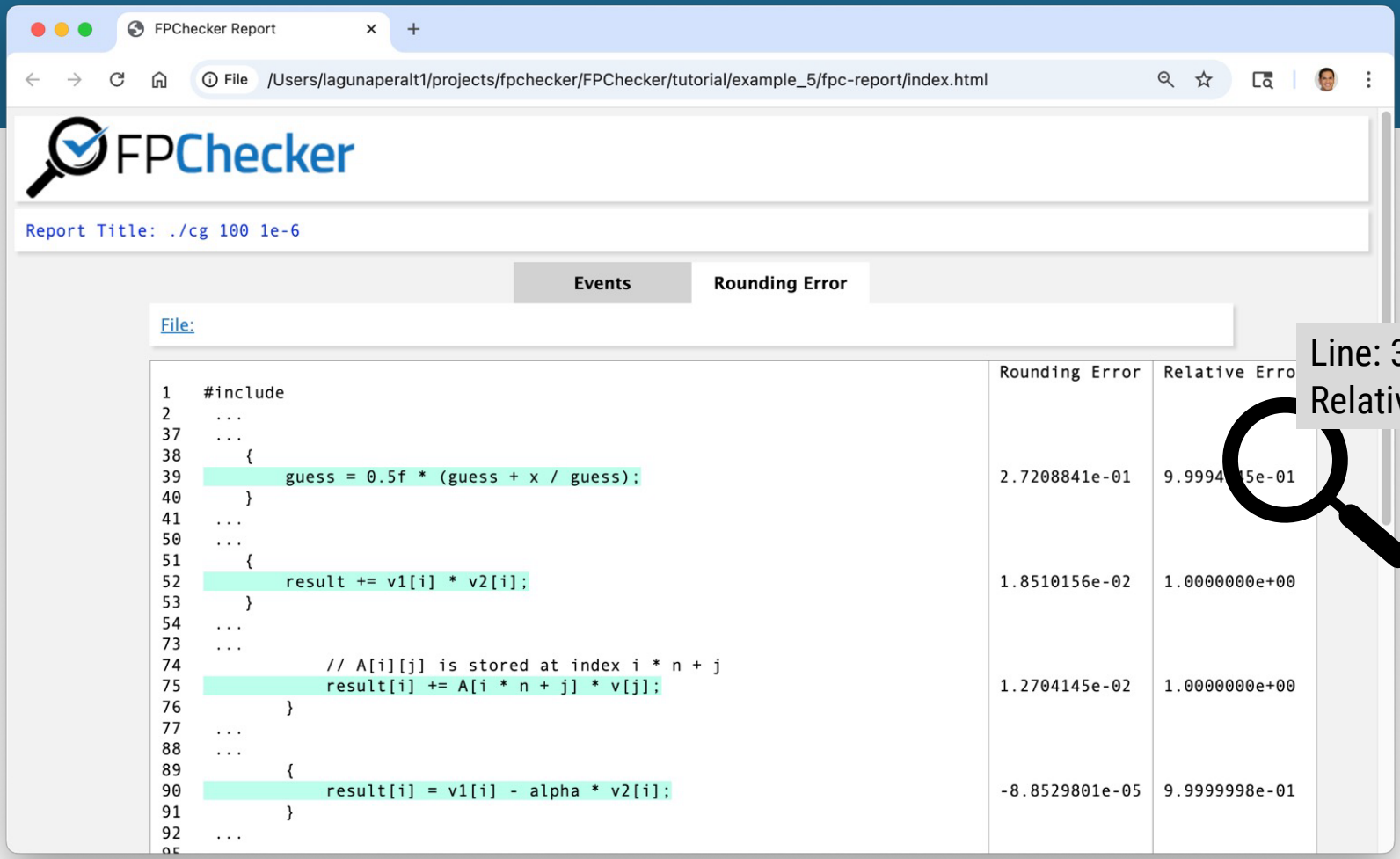
Side-by-Side Execution in Higher Precision

Example (average): $y = \frac{x_1 + x_2 + x_3}{N}$

```
1 void average(float *array, int N, float &average)
2 {
3     float temp = 0.0f;
4     for (int i = 0; i < N; ++i)
5     {
6         temp += array[i];
7     }
8     average = sum_f / N;
9 }
```

	Relative Rounding Error
`3 float temp = 0.0f;`	0.000000
`6 temp += array[i];`	7.12845e-7
`8 average = sum_f / N;`	7.12345e-9




FPChecker Report
File
/Users/lagunaperalt1/projects/fpchecker/FPChecker/tutorial/example_5/fpc-report/index.html
FPChecker
Report Title: ./cg 100 1e-6
Events
Rounding Error
File:
Rounding Error
Relative Erro
1 #include
2 ...
37 ...
38 {
39 guess = 0.5f * (guess + x / guess);
2.7208841e-01
9.9994 5e-01
40 }
41 ...
50 ...
51 {
52 result += v1[i] * v2[i];
1.8510156e-02
1.0000000e+00
53 }
54 ...
73 ...
74 // A[i][j] is stored at index i * n + j
75 result[i] += A[i * n + j] * v[j];
1.2704145e-02
1.0000000e+00
76 }
77 ...
88 ...
89 {
90 result[i] = v1[i] - alpha * v2[i];
-8.8529801e-05
9.9999998e-01
91 }
92 ...
Line: 39
Relative error: 9.99e-01

Accumulated Rounding Error Per Line:
"Last Seen" Error

```
1   float dot_product(
2       const vector<float> &v1, const vector<float> &v2)
3   {
4       float result = 0.0f;
5       for (size_t i = 0; i < v1.size(); ++i)
6       {
7           result += v1[i] * v2[i];
8       }
9       return result;
10  }
```

Example: v1 = {0.1, 0.2}
v2 = {0.3, 0.4}

Question:
How many arithmetic operations will we observe at runtime in line 7? *(assume no FMA)*

(a) 2 (b) 4 (c) 6 (d) 7

At runtime:

$temp_1 = v_{1,1} \times v_{2,1}$ Error 1

$result = temp_1 + 0.0$ Error 2

$temp_2 = v_{1,2} \times v_{2,2}$ Error 3

$result = temp_2 + result$ Error 4

"Last seen" error for line 7

Auto-Vectorizaiton is disabled

- We disable auto-vectorize
- Added flags: -fno-vectorize -fno-slp-vectorize
- To keep source-line attribution accurate

Software Requirements

- Linux, Mac OS
- LLVM/Clang 19.1.7
- Cmake
- Recent Python (e.g., python 3.12 or higher)
- Matplotlib
- Optional for parallel code:
  - MPI
  - OpenMP

Questions

Agenda

	Module	Duration	Time (AM)
☑	Welcome	10	9:00 – 9:10
☑	Motivation & Overview	5	9:10 – 9:15
☑	Floating-Point Basics	20	9:15 – 9:35
☑	FPChecker's Workflow	15	9:35 – 9:50
	Installation, Access to Examples	15	9:50 – 10:05
	Guided Practice 1: *Exceptions (linear solver)*	15	10:05 – 10:20
	Guided Practice 2: *Dynamic Range Analysis*	15	10:20 – 10:35
	Guided Practice 3: *Simple Mixed-Precision Example*	25	10:35 – 11:00
	Break	30	11:00 – 11:30
	Guided Practice 4: *Mixed-Precision in CG*	20	11:30 – 11:50
	Demo: *Single-Line Errors in CG (Cancellation)*	10	11:50 – 12:00
	Independent Practice: *Hypre linear solver package*	30	12:00 – 12:30
	Recap & Next Steps	30	12:30 – 1pm

Module 4

Installation & Access to AWS

- Conda installation guide
- Accessing AWS instances
- Installing FPChecker

How to install Anaconda on Mac or Linux

For Mac, watch this YouTube video:

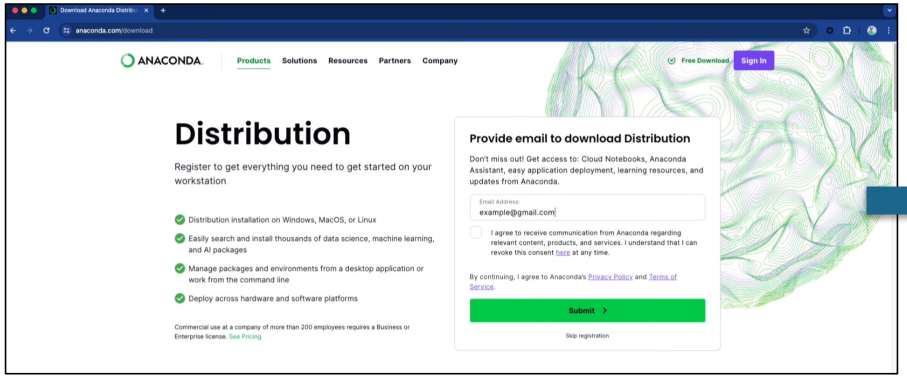
https://youtu.be/DNu8pQOYRGg

For Linux:

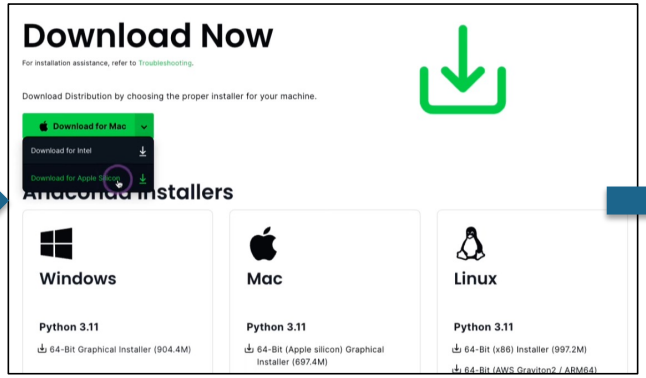
https://docs.conda.io/projects/conda/en/stable/user-guide/install/linux.html

Steps to Install Conda in Mac

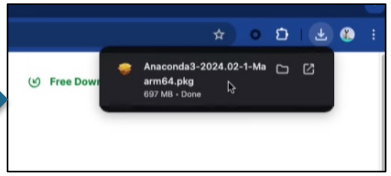
Got https://www.anaconda.com/ -> Download



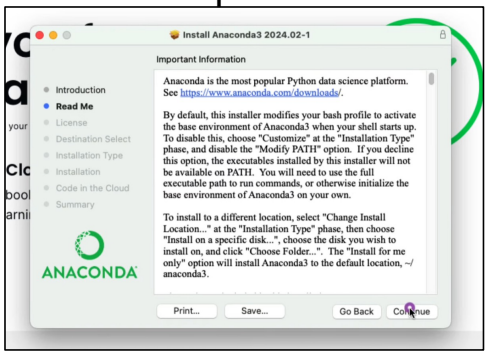
Download



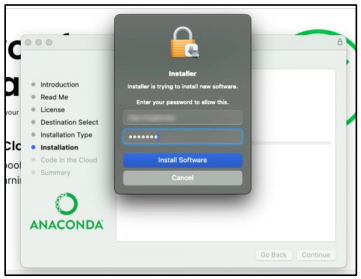
Open Installer



Follow the steps



Provide password if asked



In the terminal, you should see the word **base**:

```
(base) [user@system] $
```

Access to AWS instances

1. Add your name to a username here: https://fpchecker.org/usernames

2.
   - Connect to the AWS instance via ssh
   - Ask instructor for the password

```
$ ssh user[N]@IP_ADDRESS
```

IP Addresses:

```
35.157.176.43
51.102.86.169
52.58.211.80
```

Install Directories

Locally in the laptop:
`/tmp`

In AWS instances:
`/home/user[number]`

Install LLVM and Python with Conda

Not needed for AWS instances

```
   # Create a conda env
$  conda create --name tutorial_env
$  conda activate tutorial_env

   # Install dependencies: cmake, LLVM, clang++, python
$  conda install cmake
$  conda install make -c conda-forge
$  conda install llvmdev=19.1.7 -c conda-forge
$  conda install clangxx=19.1.7 -c conda-forge
$  conda install python=3.12.9
$  conda install -c conda-forge git

   # Install matplotlib. Not required to build FPChecker, but needed for reports
$  pip3 install matplotlib

   # MPI for some examples, but not required to build for FPChecker
$  conda install openmpi=5.0.7 -c conda-forge
```

Optional: Test your Clang/LLVM installation

Not needed for AWS instances

Create a file **`hello.cpp`** with this:

```
#include <iostream>

int main() {
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

Compile and run:

```
$  clang --version

$  clang++ -o hello -O2 -g hello.cpp
$  ./hello
```

Get FPChecker

This tutorial is based on release v0.6

```
   # We will install all in /tmp
$  cd /tmp
$  mkdir tutorial
$  cd tutorial

   # Clone FPChecker
$  git clone https://github.com/LLNL/FPChecker.git
$  cd FPChecker
```

In AWS instance, go to home dir:

```
$ cd /home/user1
```

Install FPChecker

```
$ mkdir build
$ cd build/
$ cmake -DCMAKE_INSTALL_PREFIX=../../install ..
$ make && make install
```

In the laptop:

```
  # Export installation path
$ export PATH=/tmp/tutorial/install/bin:$PATH
```

In AWS instance:

```
$ export PATH=/home/user[NUMBER]/tutorial/install/bin:$PATH
```

Troubleshooting

```
# If the right python3 is not found, set the DPython3_ROOT_DIR
$ cmake -DCMAKE_INSTALL_PREFIX=../../install -DPython3_ROOT_DIR=/opt/anaconda3 ..
```

fpchecker-show

```
$ fpchecker-show
  ========================================
           FPChecker Configuration
  ========================================

  Installation path: /home/user1/tutorial/install

  Add the following to CFLAGS and/or CXXFLAGS:

  (1) For exceptions checking:
  -g -include /home/user1/tutorial/install/src/Runtime_cpu.h -fpass-
  plugin=/home/user1/tutorial/install/lib/libfpchecker_cpu.so

  (2) For rounding error tracking:
  -g -fno-vectorize -fno-slp-vectorize -include /home/user1/tutorial/install/src/Runtime_error.h
  -fpass-plugin=/home/user1/tutorial/install/lib/libfpchecker_error.so

  Wrappers are located here:
  /home/user1/tutorial/install/bin/clang-fpchecker
  /home/user1/tutorial/install/bin/clang++-fpchecker
  /home/user1/tutorial/install/bin/mpicc-fpchecker
  /home/user1/tutorial/install/bin/mpicxx-fpchecker
```

Module 5

Guided Practice 1: Exceptions

Location: tutorial/example_1

- The first win: easy to follow example
- Exceptions in simple linear solver (NaN, Infinity)
- Creating traces (JSON)
- Generating reports
- Show first report of errors

First, build the *common* library
It will be used in other examples

- Common library:
  - BLAS operations
  - Linear solvers (LU, CG)
  - Matrix & vector printing
  - Other functionalities

In your laptop:

```
   # Compile library
$  cd /tmp/tutorial/FPChecker/tutorial/common/
$  FPC_INSTRUMENT=1 make
```

In AWS instance:

```
   # Compile library
$  cd /home/user[NUMBER]/tutorial/FPChecker/tutorial/common/
$  FPC_INSTRUMENT=1 make
```

Example 1:
NaN & Infinity exception in linear solver

- Location:

  `tutorial/example_1/lu_solve.cpp`

- Program description
  - Linear solver $Ax = b$
  - Solves $Ax = 1$
  - LU decomposition: $PA = LU$
  - Partial pivoting
  - Solve by forward/backward substitution

FPChecker's Use Case

- Use an ill-condition problem (matrix)
- Produces NaN and Infinity
  - U factor ends up with zero diagonal
  - Division by zero
- FPChecker locates the exceptions

JSON Trace Files

After an instrumented program runs, it should generate a trace file in .fpc_logs

```
$ ./application input1 input1
  ...
  ...

$ ls .fpc_logs/
  fpc_king01_27794.json
```

JSON file format:

`fpc_[hostname]_[PID].json`

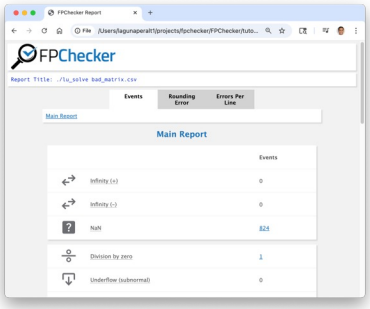
Generating Reports

- To generate an HTML report, run **`fpc-create-report`**
- The parameter –t is optional to define a title

```
$ fpc-create-report -t "report title"
  Generating FPChecker report...
  ...
  Report created: ./fpc-report/index.html
```

- Reports are created in **`./fpc-report`**
- In Mac, the report can be open easily:

```
$ open ./fpc-report/index.html
```



- The –s option shows a report in the shell
- It does not create an HTML report

```
$ fpc-create-report -s
  Generating FPChecker report...

  ========= Main Report ===========
  positive_infinity               0
  negative_infinity               0
  nan                             824
  division_by_zero                1
  cancellation                    0
  comparison                      0
  underflow                       0
  latent_positive_infinity        0
  latent_negative_infinity        0
  latent_underflow                0

  ===== Rounding Error Report =====
  No files with rounding errors.
```

Answer these Questions

5 minutes

Follow the README.md

- Which code lines produce NaN values?
- Where does division by zero (or near-zero pivot) occur?

Example 1: Script (Good Matrix)

```
  # Go back to example 1
$ cd ../example_1/

  # Compile and run problem
$ FPC_INSTRUMENT=1 make
$ ./lu_solve matrix.csv

  # List trace files (there is one)
$ ls -l .fpc_logs/

  # Create report
$ fpc-create-report -t "./lu_solve matrix.csv"
$ open fpc-report/index.html


$
```

Nothing interesting in the report

Optional: Transfer HTML Report to your Laptop

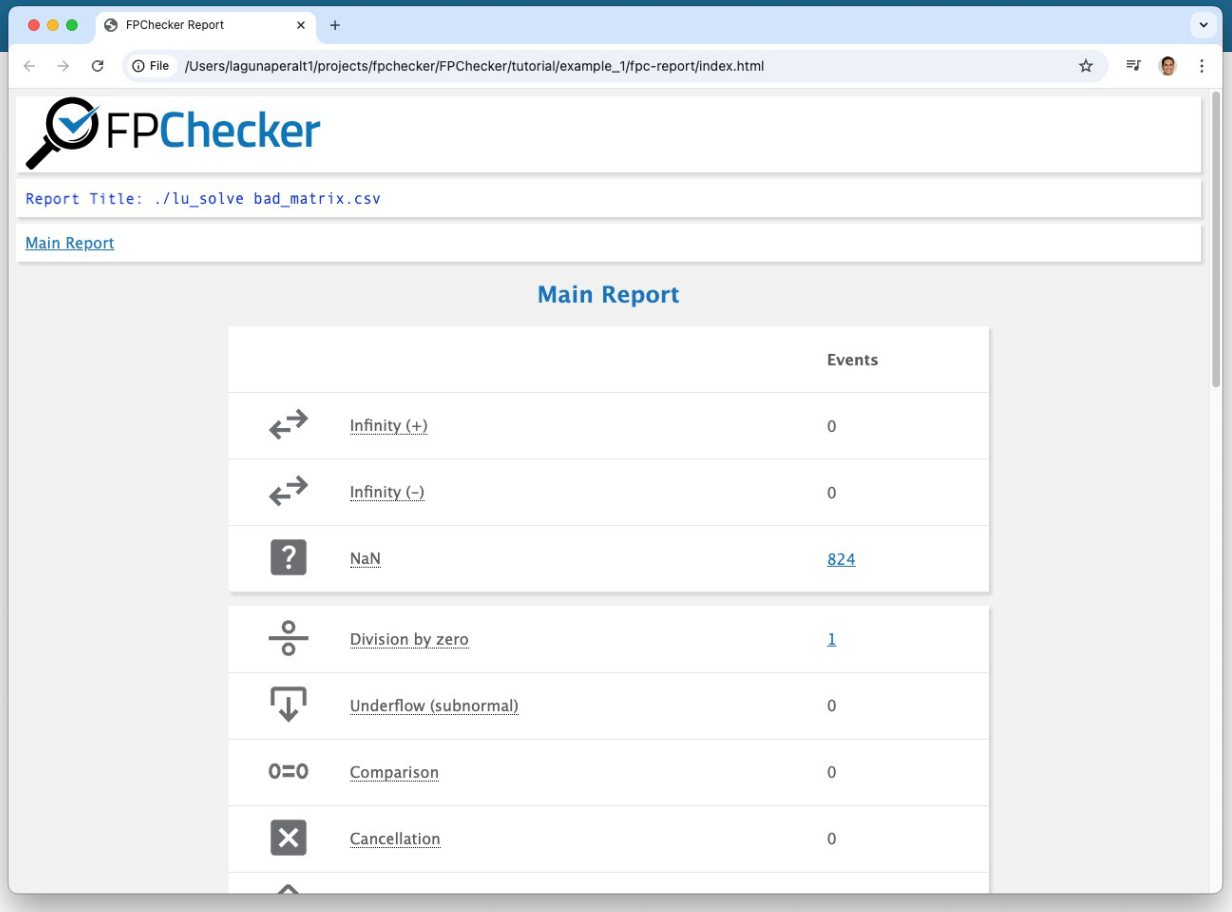
```
$  scp -r user1@54.152.122.0:/home/user2/tutorial/FPChecker/tutorial/example_1/fpc-report .
   ...
   ...

$  open fpc-report/index.html
```

Example 1: Script (Bad Matrix)

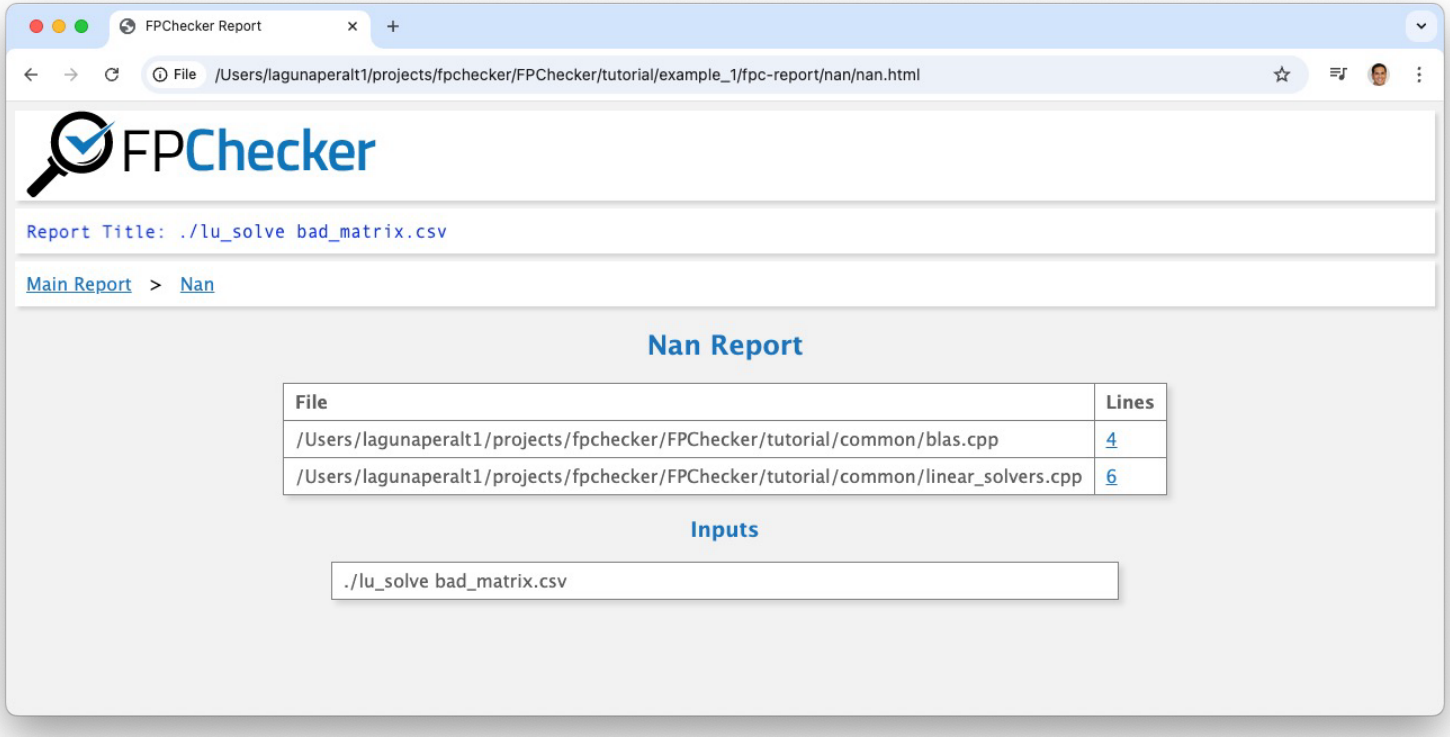
```
  # Clear logs and remove report
$ fpc-create-report -rc

  # Run problem
$ ./lu_solve bad_matrix.csv

  # Create report
$ fpc-create-report -t "./lu_solve bad_matrix.csv"
$ open fpc-report/index.html
```

- NaN
- Division by zero

Report of Example 1



Report of Example 1



Module 6

Guided Practice 2: Dynamic Range Analysis

Location: tutorial/example_2

- Dynamic range analysis for FP32 and FP64
- Example with small PDE solver
- FP32 goes out of range
- Requires higher precision

Example 2:
Exponent Usage on FP64-to-FP32 porting

- Location:

  `tutorial/example_2/reaction_diffusion.cpp`

- Program description
  - 1D linear **reaction-diffusion equation**
  - PDE: $\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + \lambda u$
  - Explicit finite difference method
  - Large λ provides positive feedback
    - Over time, it leads to **exponential growth**

FPChecker's Use Case

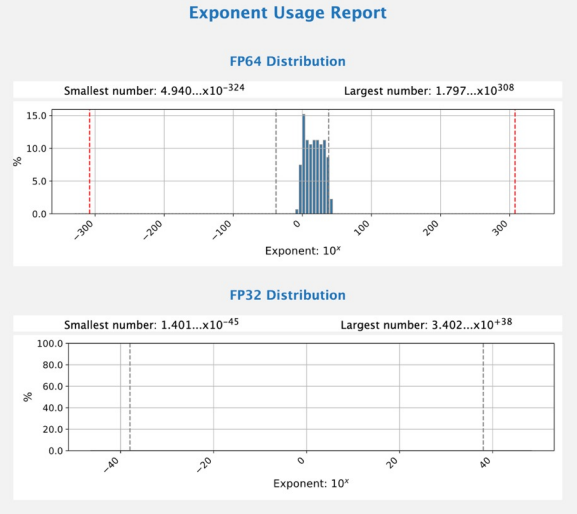
- Run simulation in FP64 and FP32
- Visualize exponent usage
- In FP32, values are "*out-of-range*"
  - Produce exceptions
  - *Should increase precision to FP64*

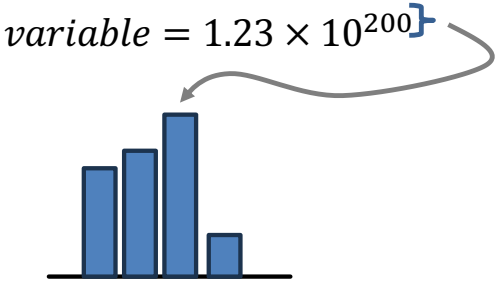
Runtime Variable to Allow Dynamic Range Profiling

- Enable this variable at **runtime**:

  FPC_EXPONENT_USAGE=1

- Allows keeping track of magnitudes (exponents)





Answer these Questions

5 minutes

Follow the README.md

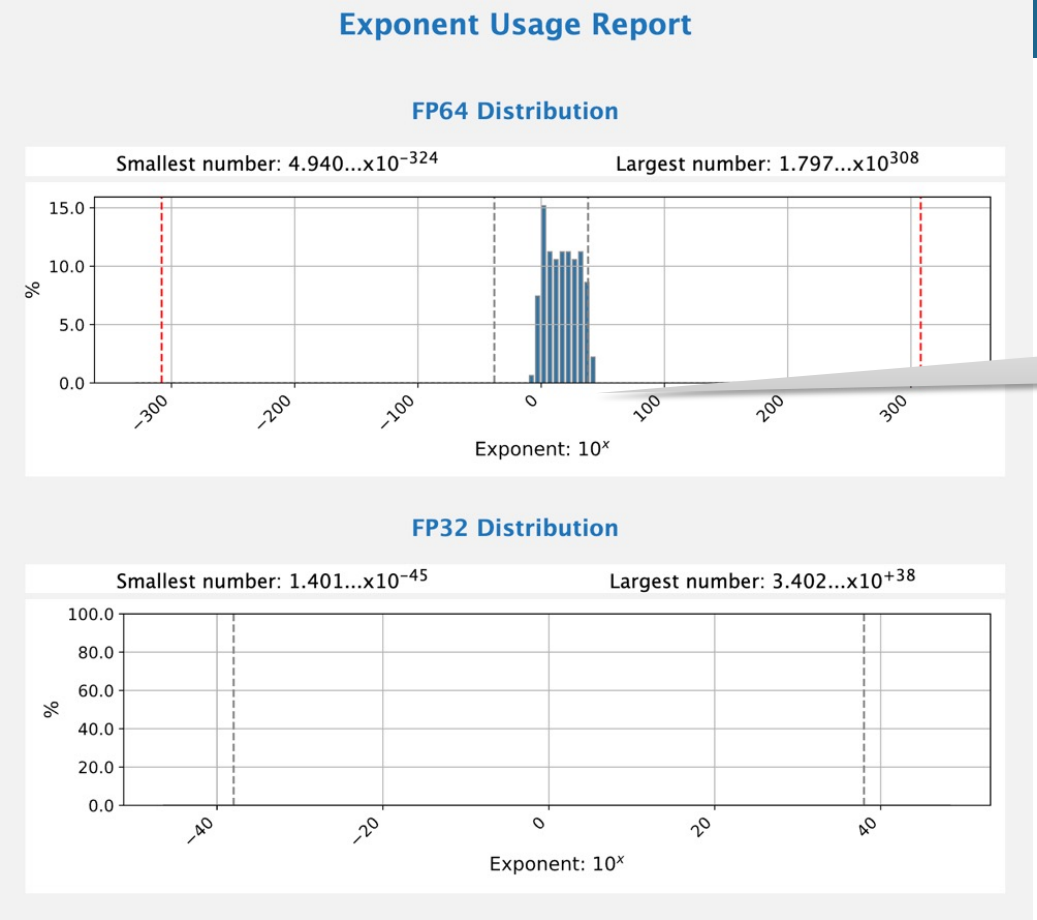
1. At what magnitude does FP32 become unsafe in this example?
2. Which report sections indicate FP32 overflow risk?
3. Does this code and input require FP64?

Example 2: Script

FP64 Version

```
  # Compile and run problem
$ FPC_INSTRUMENT=1 make
$ FPC_EXPONENT_USAGE=1 ./reaction_diffusion

  # List trace files (there are two)
$ ls -l .fpc_logs/

  # Create report
$ fpc-create-report
$ open fpc-report/index.html

  # Clear logs and remove report
$ fpc-create-report -rc
```

Report (Example 1, FP64)



Out of range for FP32

Example 2: Script

First Step:

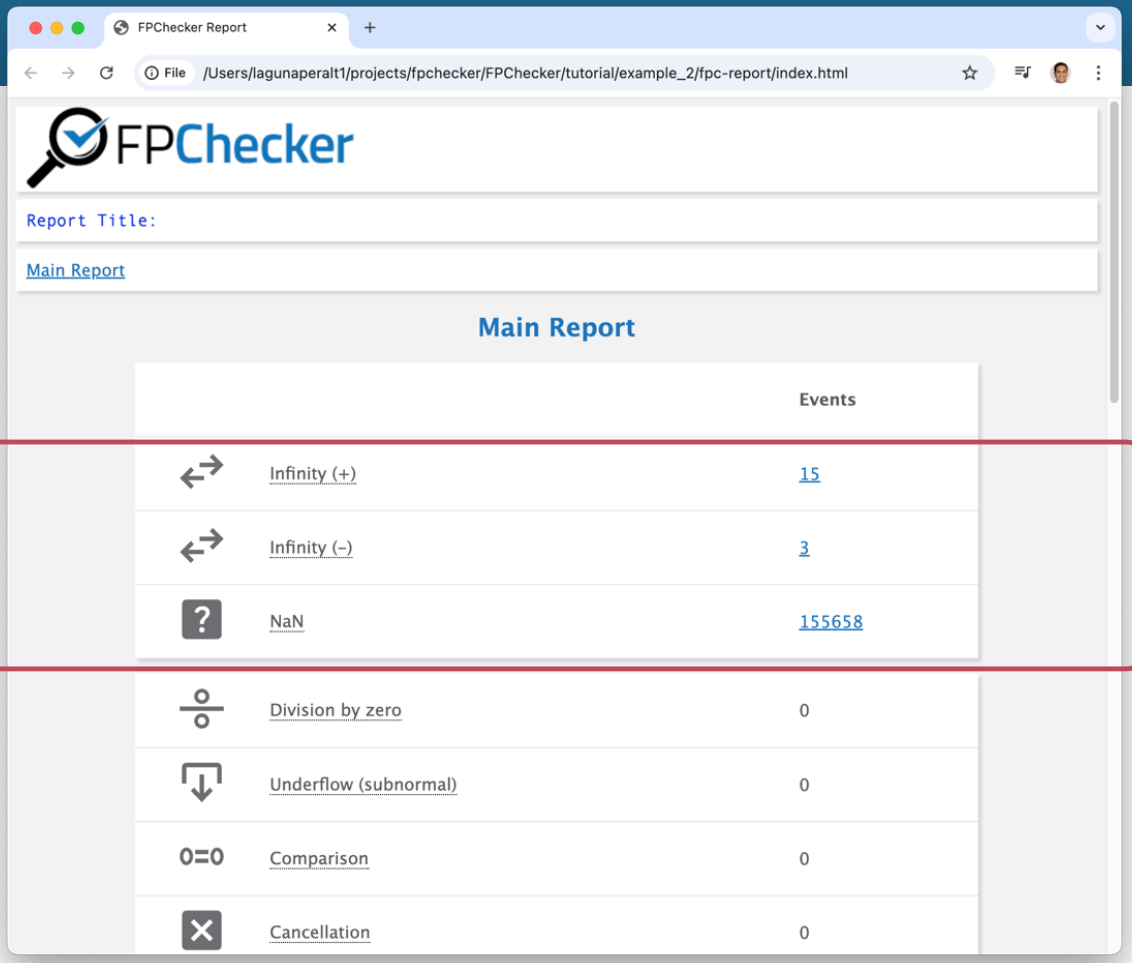
- Open `reaction_diffusion.cpp`
- Modify lines 7-8 to use FP32

```
typedef double Real_t;
// typedef float Real_t;
```

FP32 Version

```
   # Compile and run problem
$  FPC_INSTRUMENT=1 make
$  FPC_EXPONENT_USAGE=1 ./reaction_diffusion

   # List traces files (there are two)
$  ls -l .fpc_logs/

   # Create report
$  fpc-create-report
$  open fpc-report/index.html

   # Clear logs and remove report
$  fpc-create-report -rc
```

Report (Example 2, FP32)



Module 7

Guided Practice 3: Simple Selective Mixed Precision

Location: tutorial/example_6

- Rounding error profiling
- Program: *variance and quadratic equation*
- See rounding error reports
- Mixed-precision using error reports

Report-Driven Mixed-Precision Workflow

1. Run a **baseline FP32** implementation with FPChecker rounding-error instrumentation
2. Use the report to identify **numerically unstable lines**
3. Promote only the sensitive parts to FP64 (mixed precision)
4. Compare FP32 and mixed-precision against FP64 reference accuracy

To enable error tracking:

FPC_INSTRUMENT_ERR_TRACKING=1
(see Makefile)

What the Example Includes

`fp32.cpp`	Baseline FP32 implementation
`mixed.cpp`	Selective FP64 promotion and algorithmic improvements
`fp64.cpp`	FP64 reference implementation
`compare_accuracy.py`	Compares numerical error for FP32 and mixed vs FP64

Program Computes Two Quantities

Quantity 1

Small root of quadratic equation

$$ar^2 + br + c = 0$$

FP32 baseline (unstable form):

$$\Delta = b^2 - 4ac,$$

$$r_{\text{small}}^{(\text{fp32})} = \frac{-b + \sqrt{\Delta}}{2a}.$$

When $b \gg c$ and $a \approx 1$, we get $\sqrt{\Delta} \approx b$.

Cancellation in $-b + \sqrt{\Delta}$

Quantity 2

Variance of data

$$S_1 = \sum_{i=0}^{N-1} x_i\,, \qquad S_2 = \sum_{i=0}^{N-1} x_i^2\,, \qquad \mu = \frac{S_1}{N}$$

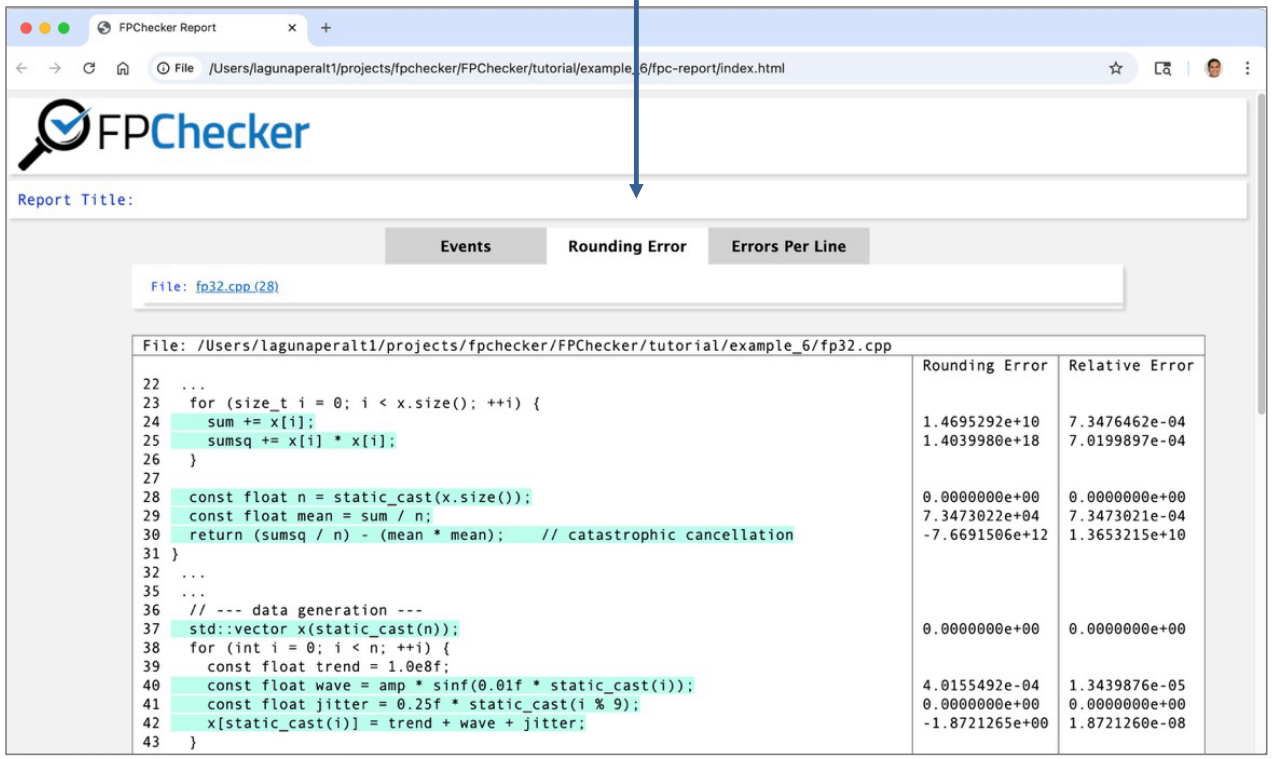
The FP32 baseline returns:

$$\sigma^2_{\text{one-pass}} = \frac{S_2}{N} - \mu^2$$

- For large-magnitude data (e.g., 10^8), both terms are large and close
- Subtracting them causes severe cancellation

Rounding Error Reports

Click on “Rounding Error”



Alternatively, generate report in the terminal:

```
$ fpc-create-report –s rounding
```

Answer these Questions

5 minutes

- Which lines show the largest relative error?
- Which error pattern looks like catastrophic cancellation?
- Which lines appear safe and can likely remain FP32?

Run the Baseline FP32 & Inspect Rounding Errors

```
  # Compile and run problem
$ make clean
$ make fp32
$ ./fp32 200000 32 1.0 1.0e8 1.0

  # Create report
$ fpc-create-report -s rounding
```

⬅️ Can take 2-3 minutes in the AWS instance

Rounding Error Report

High-error hotspot: 24-30

High-error hotspot: 49-52

```
--- File: /Users/lagunaperalt1/projects/fpchecker/FPChecker/tutorial/example_6/fp32.cpp
Line    | Code                                                                |         Error |    Rel. Error
-------------------------------------------------------------------------------------------------------------
24      | sum += x[i];                                                        |  1.469529e+10 |  7.347646e-04
25      | sumsq += x[i] * x[i];                                               |  1.403998e+18 |  7.019990e-04
28      | const float n = static_cast<float>(x.size());                       |  0.000000e+00 |  0.000000e+00
29      | const float mean = sum / n;                                         |  7.347302e+04 |  7.347302e-04
30      | return (sumsq / n) - (mean * mean);    // catastrophi...          | -7.669151e+12 |  1.365321e+10
37      | std::vector<float> x(static_cast<size_t>(n));                       |  0.000000e+00 |  0.000000e+00
40      | const float wave = amp * sinf(0.01f * static_cast<flo...            |  4.015549e-04 |  1.343988e-05
41      | const float jitter = 0.25f * static_cast<float>(i % 9);             |  0.000000e+00 |  0.000000e+00
42      | x[static_cast<size_t>(i)] = trend + wave + jitter;                  | -1.872127e+00 |  1.872126e-08
46      | const float a = coeff.a;                                            |  0.000000e+00 |  0.000000e+00
47      | const float b = coeff.b;                                            |  0.000000e+00 |  0.000000e+00
48      | const float c = coeff.c;                                            |  0.000000e+00 |  0.000000e+00
49      | const float disc = b * b - 4.0f * a * c;                            | -2.725642e+08 |  2.725642e-08
50      | const float sqrt_d = sqrtf(disc);                                   | -1.490116e-08 |  1.490116e-16
51      | const float numer = -b + sqrt_d;                          /...       | -1.490116e-08 |  1.000000e+00
52      | const float root_small = numer / (2.0f * a);                        | -7.450581e-09 |  1.000000e+00
55      | const float variance = variance_one_pass_fp32(x);                   |  0.000000e+00 |  0.000000e+00
57      | return {root_small, variance};                                      |  0.000000e+00 |  0.000000e+00
62      | const float  amp = (argc > 2) ? std::strtof(argv[2], ...            |  0.000000e+00 |  0.000000e+00
65      | coeff.a = (argc > 3) ? std::strtof(argv[3], nullptr) ...            |  0.000000e+00 |  0.000000e+00
66      | coeff.b = (argc > 4) ? std::strtof(argv[4], nullptr) ...            |  0.000000e+00 |  0.000000e+00
67      | coeff.c = (argc > 5) ? std::strtof(argv[5], nullptr) ...            |  0.000000e+00 |  0.000000e+00
69      | const Results r = run_problem_fp32(n, amp, coeff);                  |  0.000000e+00 |  0.000000e+00
73      | std::cout << "n=" << n << " amp=" << amp << "\n";                   |  0.000000e+00 |  0.000000e+00
74      | std::cout << "root_small=" << r.root_small << "\n";                 |  0.000000e+00 |  0.000000e+00
75      | std::cout << "variance="   << r.variance   << "\n";                 |  0.000000e+00 |  0.000000e+00
137     | (line not found)                                                    |  0.000000e+00 |  0.000000e+00
1305    | (line not found)                                                    |  0.000000e+00 |  0.000000e+00
```

Mixed-Precision Algorithmic Changes

Quantity 1

Small root of quadratic equation

Mixed and FP64 programs compute the large root first:

$$r_{\text{large}} = \frac{-b - \sqrt{\Delta}}{2a},$$

Then use Vieta's product relation: $r_{\text{small}}\, r_{\text{large}} = c/a$:

$$r_{\text{small}} = \frac{c}{a, r_{\text{large}}}$$

- Avoids the unstable subtraction: $\left(-b + \sqrt{\Delta}\right)$
- Numerically much more robust

Quantity 2

Variance of data

First pass:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

Second pass:

$$\sigma^2_{\text{two-pass}} = \frac{1}{N}\sum_{i=0}^{N-1} (x_i - \mu)^2$$

- Better conditioned for the centered residuals
- Avoids subtracting two large nearly equal moments

Example Combines Code + Algorithmic Changes

- Precision promotion:
  - Cancellation identification
  - Precision promotion at sensitive operations

- Algorithmic changes:
  - Algorithmic improvement
  - Numerically stable formulas

Compile mixed and FP64 versions

```
  # Compile mixed and FP64 versions
$ make mixed fp64
$ make compare
```

Accuracy Improvements

```
./compare_accuracy.py ./fp32 ./mixed ./fp64 200000 32 1.0 1.0e8 1.0
=== FP32 output ===
mode=fp32
n=200000 amp=32
root_small=0
variance=7.669150646e+12

=== Mixed precision output ===
mode=mixed
n=200000 amp=32
root_small=-1e-08
variance=530.58767247211722

=== FP64 reference output ===
mode=fp64
n=200000 amp=32
root_small=-1e-08
variance=512.5014460013125

=== Relative error (fp32 vs fp64) ===
root_small: 1.000000e+00
variance: 1.496415e+10

=== Relative error (mixed vs fp64) ===
root_small: 0.000000e+00
variance: 3.529010e-02
```

Questions

Agenda

	Module	Duration	Time (AM)
☑	Welcome	10	9:00 – 9:10
☑	Motivation & Overview	5	9:10 – 9:15
☑	Floating-Point Basics	20	9:15 – 9:35
☑	FPChecker's Workflow	15	9:35 – 9:50
☑	Installation, Access to Examples	15	9:50 – 10:05
☑	Guided Practice 1: *Exceptions (linear solver)*	15	10:05 – 10:20
☑	Guided Practice 2: *Dynamic Range Analysis*	15	10:20 – 10:35
☑	Guided Practice 3: *Simple Mixed-Precision Example*	25	10:35 – 11:00
	Break	30	11:00 – 11:30
	Guided Practice 4: *Mixed-Precision in CG*	20	11:30 – 11:50
	Demo: *Single-Line Errors in CG (Cancellation)*	10	11:50 – 12:00
	Independent Practice: *Hypre linear solver package*	30	12:00 – 12:30
	Recap & Next Steps	30	12:30 – 1pm

Module 8

Guided Practice 4: Data-Driven Mixed-Precision Conjugate Gradient

Location: tutorial/example_7

- CG example code (functions, precision, etc)
- Rounding error analysis for a given input
- Data-driven mixed-precision version

Example Objectives

1. Run the baseline FP32 solver with instrumentation
2. Identify high-error lines from the report
3. Compare FP32, mixed, and FP64 behavior
4. Explain why targeted mixed precision improves convergence and residual

Algorithm in FP32

Algorithm 1 Conjugate Gradient (FP32 baseline)

Require: SPD matrix A, RHS b, tolerance τ, max iterations $k_{\max}$

$x_0 \leftarrow 0$, $r_0 \leftarrow b - Ax_0$, $p_0 \leftarrow r_0$
$\rho_0 \leftarrow r_0^T r_0$, $b_n \leftarrow \sqrt{b^T b}$
for $k = 0, 1, \ldots, k_{\max} - 1$ **do**
    $q_k \leftarrow Ap_k$
    $\alpha_k \leftarrow \rho_k / (p_k^T q_k)$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k q_k$
    $\rho_{k+1} \leftarrow r_{k+1}^T r_{k+1}$
    if $\sqrt{\rho_{k+1}}/b_n < \tau$ **then**
        return x_{k+1}
    end if
    $\beta_k \leftarrow \rho_{k+1}/\rho_k$
    $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
end for
return $x_{k_{\max}}$

- No preconditioner
- Example solves $Ax = 1$
- Script to generate matrices provided
- Tridiagonal Toeplitz SPD matrices
- Varying condition numbers

What the Example includes

`cg.cpp`	FP32 solver (will be instrumented with FPChecker rounding-error tracking)
`cg_mixed.cpp`	Mixed-precision solver (targeted FP64 intermediates)
`cg_FP64.cpp`	FP64 reference solver
`generate_cg_matrices.py`	Matrix generation utility
`matrices/`	Pre-generated SPD matrices used in the tutorial
`Makefile`	Build rules for all variants

Challenge & Questions

5 minutes

- Follow README.md
- Run FP32 and inspect rounding errors

Questions:

1. Which lines have the largest relative error?
2. Which operations are likely causing cancellation or unstable accumulation?
3. Which parts can remain FP32?
4. Which variables should move to FP64?

Script 1: Getting Rounding Errors in FP32

- Check in Makefile we are using: FPC_INSTRUMENT_ERR_TRACKING=1

```
   # Compile and run problem
$  make clean
$  make cg
$  ./cg matrices/5_matrix_3.998e+04.csv 500 1e-6

   # Create report in the terminal
$  fpc-create-report -s rounding
```

Rounding Error Report

Line	Code	Error	Rel. Error
39	guess = 0.5f * (guess + x / guess);	2.148692e+03	9.999707e-01
52	result += v1[i] * v2[i];	1.154287e+06	1.000000e+00
75	result[i] += A[i * n + j] * v[j];	-9.567699e+01	1.010557e+00
90	result[i] = v1[i] - alpha * v2[i];	9.567699e+01	9.999952e-01
97	result[i] = v1[i] + alpha * v2[i];	-2.400936e+02	1.448971e+00
111	result[i] = r[i] + beta * p[i];	2.366245e+03	1.000000e+00
146	row.push_back(atof(token));	0.000000e+00	0.000000e+00
165		0.000000e+00	0.000000e+00
227		0.000000e+00	0.000000e+00
229	float alpha = rs_old / dot_product(p, Ap);	8.886324e-01	4.678308e+05
239	float relative_residual = my_sqrt(rs_new) / relative_b;	9.806319e+01	1.000000e+00
248	float beta = rs_new / rs_old;	2.993113e-01	2.947664e-01
265		-7.821318e+01	1.000000e+00
287	float tolerance = (argc > 3) ? atof(argv[3]) : 1e-6f;	0.000000e+00	0.000000e+00
645	(line not found)	0.000000e+00	0.000000e+00

Lines 39:
FP32 Newton-style sqrt routine

Line 52:
Dot product accumulation

Lines 75:
Matrix-vector row accumulation

Lines 90, 97:
Vector fused updates

Lines 111:
Search-direction update

Line 229:
Division with high error

Line 239:
Division with high error

Errors might be slightly different between AWS instances and Mac

Changes Performed for Mixed-Precision

Observation: FP32 accumulation and sensitive divisions amplify error

Main changes:

- We target FP64 intermediate operations to reduce rounding error propagation
- No need to convert all storage to FP64 (***most storage remains FP32***)
- We promote to FP64 only ***sensitive arithmetic***
- *No changes performed to the algorithm*

Script 2: Mixed and FP64

```
   # Compile and run mixed and FP64
$  make cg_mixed cg_FP64

$  printf "\n=== FP32 ===\n"
$  ./cg matrices/5_matrix_3.998e+04.csv 500 1e-6

$  printf "\n=== Mixed ===\n"
$  ./cg_mixed matrices/5_matrix_3.998e+04.csv 500 1e-6

$  printf "\n=== FP64 ===\n"
$  ./cg_FP64 matrices/5_matrix_3.998e+04.csv 500 1e-6
```

Numerical Results for FP32, Mixed, and FP64

=== FP32 ===

```
Converged in 148 iterations. Residual Norm: 7.414996e-06
Average time per iteration: 2.775208e-02 seconds
Execution time: 4.167169e+00 seconds
Final Residual Norm (||Ax - b||): 3.148735e-02
```

=== Mixed ===

```
Converged in 77 iterations. Residual Norm: 7.945719e-06
Average time per iteration: 6.839299e-06 seconds
Execution time: 5.606250e-04 seconds
Final Residual Norm (||Ax - b||): 5.171927e-03
```

=== FP64 ===

```
Converged in 50 iterations. Residual Norm: 1.907349e-06
Average time per iteration: 1.892252e-05 seconds
Execution time: 1.064250e-03 seconds
Final Residual Norm (||Ax - b||): 1.907349e-06
```

	FP32	Mixed	FP64
Iterations	148	77	50
||Ax - b||	3.148735e-02	5.171927e-03	1.907349e-06

Questions

Module 9

Demo: Single-Line Errors in Conjugate Gradient

Location: tutorial/example_5

- Demo example of errors in a single line
- Cancellation errors in CG (in solution update)
- Full functionality for future release v0.7

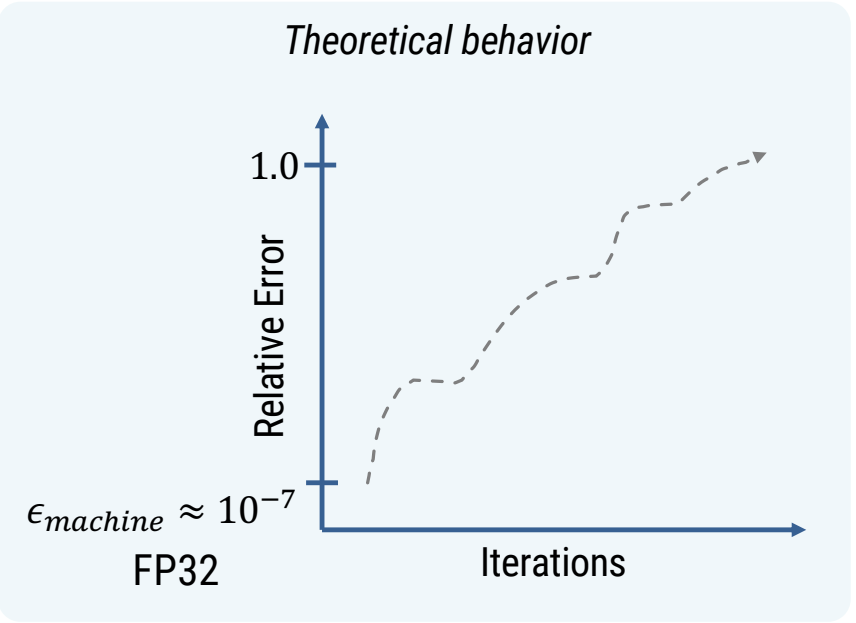
Case Study:
Catastrophic Cancellation in CG

- As the solution x_k approaches the true solution x, the two vectors in the subtraction become increasingly nearly equal in magnitude:

$$r_{k+1} = r_k - \alpha(Ap_k)$$

Become nearly equal

- Catastrophic **cancellation** occurs



Cancellation Leads to Loss of Orthogonality

Residual update in CG

$$r_{k+1} = r_k - \alpha(Ap_k)$$

In exact math, this must hold:

$$r_{k+1}^T r_k = 0$$

- Must be true due to Krylov space
- Entire set of residuals are mutually orthogonal

Let's assume for a given iteration k:

$$r_k = \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix} \qquad \alpha(Ap_k) = C = \begin{matrix} c_1 \\ c_2 \\ \vdots \end{matrix}$$

$$r_{k+1}^T r_k = [r_1 - c_1, r_2 - c_2, \cdots] \cdot \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix}$$

$$= (r_1 - c_1)r_1 + (r_2 - c_2)r_2 + \cdots = 0$$

With cancellation, we introduce an error e

$$r_{k+1}^T r_k = [r_1 - c_1 + e_1, r_2 - c_2 + e_2, \cdots] \cdot \begin{matrix} r_1 \\ r_2 \\ \vdots \end{matrix}$$

$$= (r_1 - c_1 + e_1)r_1 + (r_2 - c_2 + e_2)r_2 + \cdots = 0$$

Large error, not bounded by machine epsilon ϵ

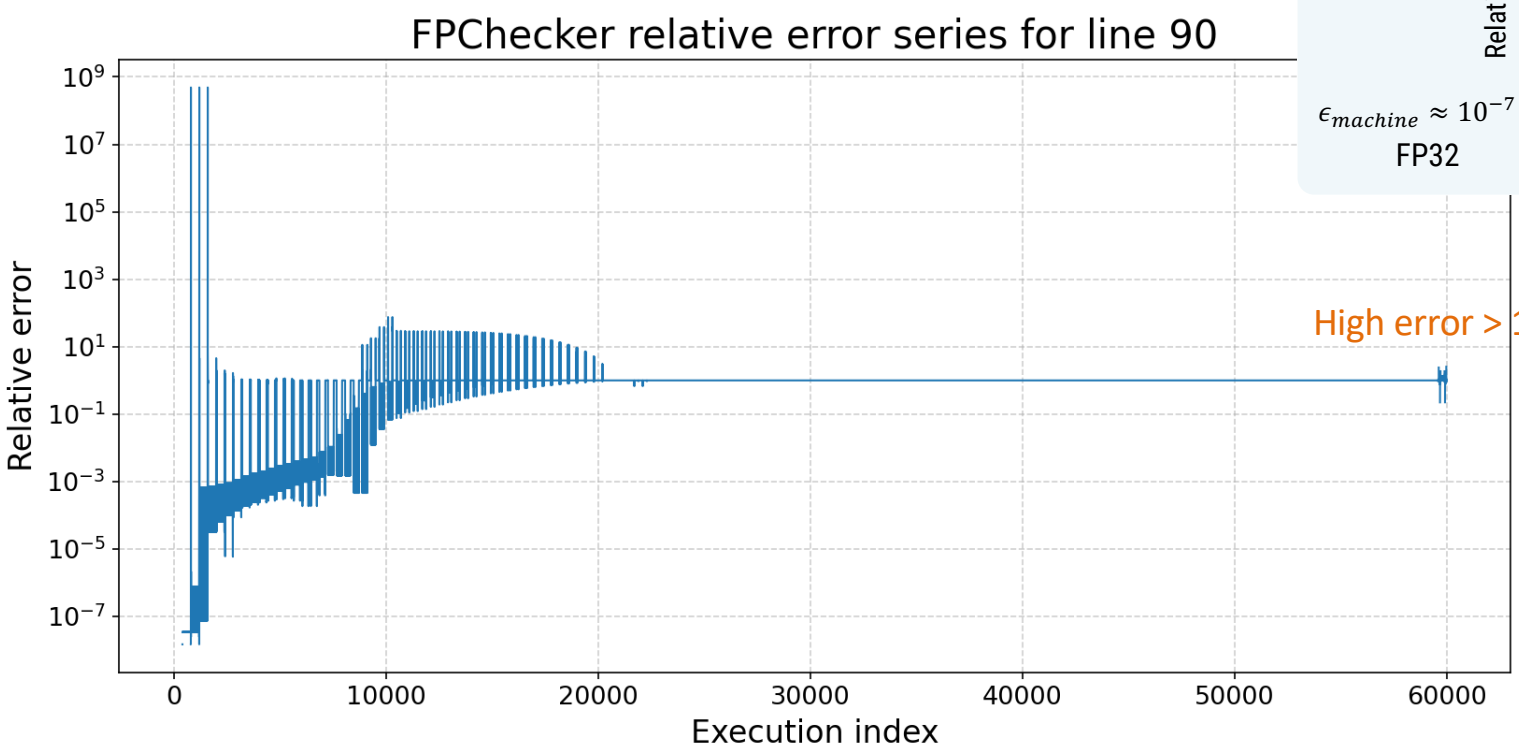
Code Region of Interest (line 90)

```
// Vector Addition/Subtraction (v1 - alpha * v2 or v1 + alpha * v2)
vector<float> vec_add_mult(const vector<float> &v1, const
vector<float> &v2, float alpha, bool subtract = false)
{
  vector<float> result(v1.size());
  if (subtract)
  {
...
89    for (size_t i = 0; i < v1.size(); ++i)
90    {
91      result[i] = v1[i] - alpha * v2[i];
    }
...
  }
...
  return result;
}
```

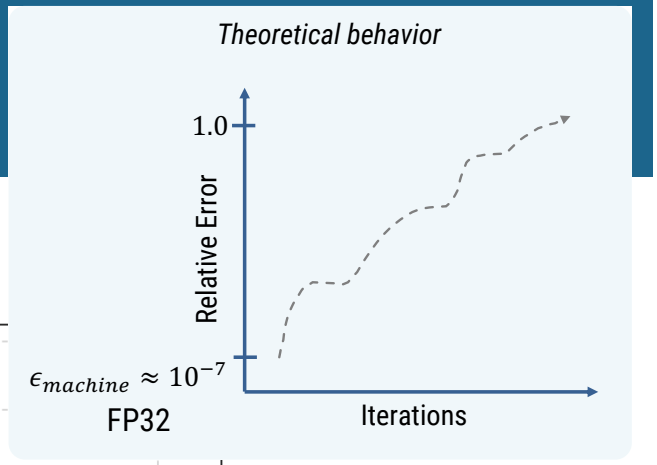
```
    # Compile and run problem
$   make clean
$   FPC_INSTRUMENT_ERR_TRACKING=1 make cg
$   FPC_SAVE_LINE_ERRORS=90 ./cg matrices/5_matrix_3.998e+04.csv 500 1e-6

    # Create plot
$   python3 plot_error_values.py .fpc_logs/errors_per_line_*.json --line 90 --show
```

Tool's Output for CG at Line 90

$$r_{k+1} = r_k - \alpha(Ap_k)$$



High error > 100%



Questions

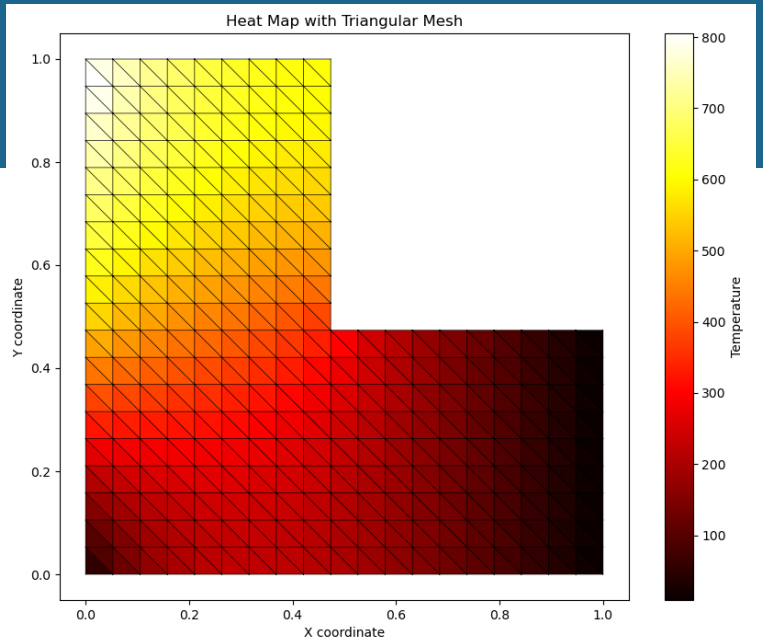
Optional Examples

Not ran in this tutorial

- Controlling slowdown: example_3
- MPI code: example_4

Example 3: Controlling slowdown

Example 3:
Annotations to Control Slowdown



- Location:

  `tutorial/example_3/heat_PDE_finite_elements.cpp`

- Program description
  - **2D Heat conduction equation** with a source term
    - Steady state
  - PDE: $\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = s(x, y)$
  - Finite elements method
    - Triangular shapes
  - $T(x, y)$: temperature distribution
  - Domain: L shape
  - Source $s(x, y) = 0$
  - Boundary conditions: temp applied on the sides

FPChecker Use Case

- Slowdown can be high (for a large problem)
- Reduce slowdown by annotating the code

Example 3: Script

- Run small problem (10 nodes)
- Should take less than 1 second

```
    # Compile and run problem
$   FPC_INSTRUMENT=1 make
$   time FPC_EXPONENT_USAGE=1 ./heat_PDE_finite_elements 10
$
    # Optional: Vizualize heat map
$   python3 plot.py
```

- Run a larger problem (50 nodes)
- It takes about 20 seconds in my laptop

```
    # Compile and run problem
$   time FPC_EXPONENT_USAGE=1 ./heat_PDE_finite_elements 50
```

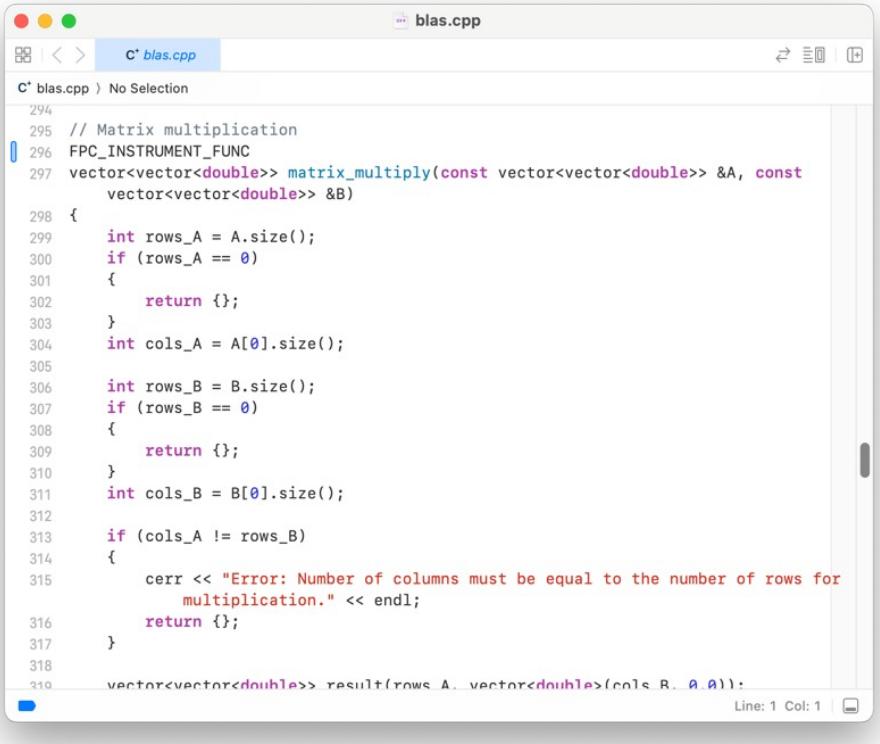
Code Annotations

- Let's annotate the matrix-multiply function
  - That is: we only instrument and analyze that function
  - Should reduce overhead significantly
- Location:

  `tutorial/common/blas.cpp`
- Search for:

  `vector<vector<double>> matrix_multiply(…`
- Add (or uncomment):

  `FPC_INSTRUMENT_FUNC`



Example 3: Script

Recompile the common library

```
$  cd ../common/
$  make clean
$  FPC_INSTRUMENT=1 FPC_ANNOTATED=1 make

$  cd ../example_3/
$  FPC_INSTRUMENT=1 FPC_ANNOTATED=1 make
$  time FPC_EXPONENT_USAGE=1 ./heat_PDE_finite_elements 50
   ...
   ...
   real    0m1.049s
   user    0m0.729s
   sys     0m0.071s
```

Lower run time

Example 4: MPI solver

Example 4: Analyzing MPI code

- Location:

  `tutorial/example_4/heat_mpi.cpp`

- Program description
  - 1D heat equation
  - PDE: $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$
  - Explicit finite difference method
- MPI domain decomposition:
  - 1D spatial grid divided into NPROC contiguous segments
  - NPROC: number of MPI processes
  - Each process computes temperature in its segment

FPChecker Use Case

- Generates traces for MPI programs
- Combine traces into a single report

Example 4: Script

```
   # Show Makefile uses CXX = mpic++-fpchecker
   # Compile and run problem
   # OMPI_CXX indicates to Open MPI which conda compiler to use
$  OMPI_CXX=clang++ FPC_INSTRUMENT=1 make
$  FPC_EXPONENT_USAGE=1 mpiexec -n 4 ./heat_mpi

   # List trace files (there are 4)
$  ls .fpc_logs/
   fpc_king01_95250.json fpc_king01_95251.json
   fpc_king01_95252.json fpc_king01_95253.json

   # Create report
$  fpc-create-report
$  open fpc-report/index.html
```

Module 10

Independent Practice: *Hypre linear solver package*

Location: tutorial/example_8

- Practice with Hypre
- Build Hypre with FPChecker
- Show error report

Hypre

- Library of scalable linear solvers: AMG, PCG, ILU, …
- Designed to handle discretized partial differential equations (PDEs) at extreme scales
- Optimized to run the world's fastest supercomputers like **El Capitan** and **Frontier**
  - Uses MPI
- Supports FP32, FP64 and mixed-precision

Example Objectives

1. Download and build Hypre without FPChecker
   — Run a Hypre test with an input
2. **Instrument** Hypre and the test with FPChecker
   — Run an instrumented Hypre test with an input
3. Produce a **rounding error report**

What the Example Includes

`README.md`	Instructions to build Hypre and test without FPChecker
`Makefile`	Makefile to build the test (hypre_test.c)
`hypre_test.c`	Test that runs three solvers: PCG, AMG, and PCG with AMG preconditioner
`matrix.csv`	Input matrix
`README_solution.md`	Solution instructions (do not read immediately)
`Makefile.solution`	Solution Makefile (do not read immediately)

Independent Practice

- **10 minutes:** build Hypre and test (follow README.md)
- **10 minutes:** Instrument Hypre and test, build error report
- **5 minutes:** Instructor reveals the solution

Do no immediately look at
- README_solution.md, and/or
- Makefile.solution

Agenda

	Module	Duration	Time (AM)
☑	Welcome	10	9:00 – 9:10
☑	Motivation & Overview	5	9:10 – 9:15
☑	Floating-Point Basics	20	9:15 – 9:35
☑	FPChecker's Workflow	15	9:35 – 9:50
☑	Installation, Access to Examples	15	9:50 – 10:05
☑	Guided Practice 1: *Exceptions (linear solver)*	15	10:05 – 10:20
☑	Guided Practice 2: *Dynamic Range Analysis*	15	10:20 – 10:35
☑	Guided Practice 3: *Simple Mixed-Precision Example*	25	10:35 – 11:00
	Break	30	11:00 – 11:30
☑	Guided Practice 4: *Mixed-Precision in CG*	20	11:30 – 11:50
☑	Demo: *Single-Line Errors in CG (Cancellation)*	10	11:50 – 12:00
☑	Independent Practice: *Hypre linear solver package*	30	12:00 – 12:30
	Recap & Next Steps	30	12:30 – 1pm

Recap & Next Steps

Things We didn't Cover…

- Multiple inputs
  - How is error impacted?
- Performance aspects
- Dynamic mixed-precision
  - Switch between FP32 and FP64 dynamically
- Other floating-point formats: FP16, bfloat16, …
- Instrumenting FP64 or mixed code
- GPUs
- Fortran

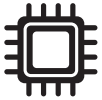
Learning Objectives

Identify numerical instabilities

— Isolate floating-point exceptions (**NaN** and **Infinity**) and **cancellation**

Characterize dynamic range and rounding error

— Generate ***per-line rounding error reports*** and ***dynamic range profiles***



Automate analysis via compiler instrumentation and FPChecker

— Lear to compile and run code with the tool

Implement data-driven mixed-precision strategies

— Use numerical error reports to assign lower or higher precision: **FP32** or **FP64**

Gain experience profiling real-world HPC libraries

ENJOYED THIS TUTORIAL?

Don’t forget to rate it!

Thank you!