Implementing Numerical Optimization Algorithms

MIP Workshop 2025 Summer School, June 2nd, 2025

Philipp M. Christophel
Technical Lead LP/MILP Solver Development,
Scientific Computing R&D, SAS Institute

Agenda

Introduction

Your Project

- Let's assume you want to implement a numerical optimization algorithm!
- Examples:
  - A new algorithm (maybe a heuristic) for a well-known problem
  - A new algorithm to solve a very special problem
  - A new algorithm for a new problem
  - An old algorithm under new circumstances (parallelism)
  - ...
- This talk should give you some ideas what to look out for

Introduction

Why Implement an Algorithm at All?

The proof is in the pudding!

Por la muestra se conoce el paño!

Probieren geht über studieren!

Introduction
Implementation Matters

- Better implementation = better perception of the algorithm
  - Example: Tableau vs. linear algebra in simplex
- Computational complexity vs. empirical research
  - Example: Simplex vs. interior point method
- Usage scenarios
  - Example: Warm starting, special data, ...
- Practical considerations
  - Example: Memory requirements, potential for parallelism
- Research on implementation is research
  - Example: Simplex papers from the 80s, cut papers from the 90s

Requirements

The Algorithm (Design)

- Before you start coding: Try the algorithm!
  - Create a small example and do it on paper
- Write it down in pseudocode
- Think about the theoretical effort each step should take
- Which options do you want to have?
- Think about what to output into the log and where to do that
- Identify sub-algorithms and third-party components
- The algorithm will evolve as you implement it

Requirements

The Data

- What input does your algorithm take?
  - For some problems, various input formats exist
- Implement your own reader or find open-source ones to use
- Identify public libraries of test problems
- Think about how to create your own test problems
- More on test problems when we talk about benchmarking ...

Requirements

The Development Environment

- Programming language
  - Clearly best choice: C (/C++)
  - Possible: FORTRAN, Rust
  - Maybe: Go, Julia
  - Python only useful as "glue" or for prototyping
- Tools
  - Integrated development environment (IDE): VS Code, Visual Studio
  - Compiler: gcc, Intel® compilers
  - Source code management: Git, GitHub
  - Debugger: gdb, UDB
  - Performance profiler: perf, Intel® VTune Amplifier
  - Memory profiler: valgrind, PurifyPlus

Requirements

The Experimentation Environment

- Machine(s) to run experiments on
  - Typically: The more the better for quicker turnaround
- Equal hardware necessary for reliable results
- Even better: Full access to configure things like power saving, turbo, ...
- Cloud computing probably not a good option (shared environment)
- Calibrate your environment and measure its accuracy!
  - Run unchanged code twice, measure the difference

Development Process

The Empirical Research Cycle[1]

Observation

1. Slow speed of some instances

Induction

2. Sparse linear algebra could be faster

Deduction

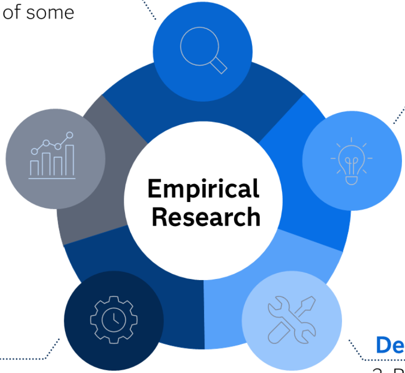
3. Replace all linear algebra with sparse versions

Testing

4. Run a benchmark

Evaluation

5. Did sparse problems get faster and dense problems get slower?



[1] according to de Groot

Development Process

Some More Processes

- Make it work, then make it fast
  1. Handle one specific test problem
  2. Handle normal execution cases
  3. Handle special cases
  4. Improve hotspots one at a time
- Test-driven development
  - Unit tests
  - Sometimes, a bug can make an algorithm very fast ...
- Continuous integration / continuous development (CI/CD)
  - Even if this just means running unit tests once a day
- Agile development
  - Quick turnaround important
  - Always have running code

Benchmarking

Test Problems

- The problems you test with matter (a LOT!)
- More $\neq$ better
- Creating problems yourself is OK
  - Random data is usually not good
  - Better: Random perturbation of real problems
  - Also good: Real data for fake problems
- Use public test problems like MIPLIB, QPLIB, MINLPLIB ...

Benchmarking

Test Setup

- Several machines with the same hardware
- Controlled environment
- Scripts to kick off the runs (tools like HTcondor can help)
- Possibly: Random seeds to counter variability
- Always include checks for correctness!

Benchmarking

Comparison

- Comparing to a previous version
  - Changes might be very small
  - Deterministic implementation needed
- Comparing to another algorithm/implementation
  - Implement yourself or download
  - Make sure published results are replicated!
  - A heuristic or specialized algorithm that beats a general purpose solver is the expected result!

Benchmarking

Evaluation

- Evaluation methods
  - Means of runtime (consider a shifted and/or geometric mean)
  - Other metrics like iterations, nodes, function evaluations
  - Performance profiles (Dolan and Moré 2002)
  - Primal(-Dual) Integral (Berthold 2013)
  - Sub-setting
  - Statistical tests
- Don't shy away from inventing your own evaluation method
- For graphical displays, read *Tufte: The Visual Display of Quantitative Information*
- Like your algorithm, you evaluation methods might evolve
- In the end, you might need to dig into the full results table and logs

It's easy to fool yourself!

Benchmarking

Example - Ratios

problem	old (sec)	new (sec)	ratio
train	0.10	10.00	0.01
car	3.90	2.10	1.86
horse	4.10	4.10	1.00
jim	300.00	112.00	2.68
bob	34.00	7.00	4.86
sally	0.20	0.30	0.67
mean	57.05	22.58	1.84
geomean	3.85	5.22	0.74

Benchmarking

Example - Shifting

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
train	0.10	10.00	0.01	10.10	20.00	0.51
car	3.90	2.10	1.86	13.90	12.10	1.15
horse	4.10	4.10	1.00	14.10	14.10	1.00
jim	300.00	112.00	2.68	310.00	122.00	2.54
bob	34.00	7.00	4.86	44.00	17.00	2.59
sally	0.20	0.30	0.67	10.20	10.30	0.99
mean	57.05	22.58	1.84	67.05	32.58	1.46
geomean	3.85	5.22	0.74	25.51	20.44	1.25
shifted				15.51	10.44	1.49

Benchmarking

Example - Handling Timeouts

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
train	0.10	10.00	0.01	10.10	20.00	0.51
car	3.90	2.10	1.86	13.90	12.10	1.15
horse	4.10	4.10	1.00	14.10	14.10	1.00
hard	3600.00	8.40	428.57	3610.00	18.40	196.20
easy	2800.00	3600.00	0.78	2810.00	3610.00	0.78
jim	300.00	112.00	2.68	310.00	122.00	2.54
bob	34.00	7.00	4.86	44.00	17.00	2.59
sally	0.20	0.30	0.67	10.20	10.30	0.99
mean	842.79	467.99	55.05	852.79	477.99	25.72
geomean	20.64	12.54	1.65	85.27	38.51	2.21
shifted				75.27	28.51	2.64

Benchmarking

Example - Sub-setting: The Wrong Way

Problems where old took more than 10sec:

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
hard	3600.00	8.40	428.57	3610.00	18.40	196.20
easy	2800.00	3600.00	0.78	2810.00	3610.00	0.78
jim	300.00	112.00	2.68	310.00	122.00	2.54
bob	34.00	7.00	4.86	44.00	17.00	2.59
shifted				599.90	98.34	6.10

Benchmarking

Example - Sub-setting: The Right Way

Problems where old OR new took more than 10sec:

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
train	0.10	10.00	0.01	10.10	20.00	0.51
hard	3600.00	8.40	428.57	3610.00	18.40	196.20
easy	2800.00	3600.00	0.78	2810.00	3610.00	0.78
jim	300.00	112.00	2.68	310.00	122.00	2.54
bob	34.00	7.00	4.86	44.00	17.00	2.59
shifted				258.58	67.27	3.84

Implementation

General Advice

- Structure the code well (but not too well)
- Have modules/functions that can be unit tested
- Include logging with various verbosity levels
- If in doubt, performance beats good structure
- Refactoring is part of the process
- LLMs/AI/Copilot can help, but their benefit for optimization algorithms is limited

Implementation

Numerics - The Basics

- Numerical algorithms today mostly use IEEE 754 `double` numbers
  - Good compromise between precision and speed
- The good news about `double` numbers:
  - Integers from -2^{53} to 2^{53} can be exactly represented
- The bad news about `double` numbers:
  - Some frequently arising numbers are not represented well
  - Example:

$$0.3 - 0.2 < 0.1$$

- The classic reference:

David Goldberg

What every computer scientist should know about floating-point arithmetic

`https://dl.acm.org/doi/10.1145/103162.103163`

Implementation

Numerics - What to Do About It

- Apply limits, e.g.
  - Define 1e20 as infinity
- Apply tolerances, e.g.
  - Zero tolerance: 1e-14
  - Feasibility tolerance: 1e-6
  - Integer tolerance: 1e-5
- Rule of thumb: When comparing double numbers, use at least the zero tolerance
- Tolerances are applied to *computed* values, avoid rounding input data
- The order of operations has an impact on precision
- Higher accuracy available but slow
  - quad precision, long double, double double, GNU Multiple Precision

Implementation

Numerics - Example

```c
#include<stdio.h>

int main() {
    double x = 0.0;
    int i;
    for (i = 0; i < 10; i++) {
        x += 0.1;
        printf("x = %.16g\n", x);
    }
    printf("0.1 * 10.0 = %.16g\n",
            0.1 * 10.0);
    return 0;
}
```

Output

```
x = 0.1
x = 0.2
x = 0.3
x = 0.4
x = 0.5
x = 0.6
x = 0.7
x = 0.7999999999999999
x = 0.8999999999999999
x = 0.9999999999999999
0.1 * 10.0 = 1
```

Implementation

Speed - Thinking About Fast Code

- Speed is just one aspect of performance
  - memory, accuracy, solution quality, ...
- Reason for better speed not always obvious
- Factors to consider: cache, locality of data, non-uniform memory access (NUMA), special CPU instructions (AVX), ...
- Every bit of work counts, e.g. zeroing out memory
- Trade-off between speed vs. memory, speed vs. responsiveness, ...
- Experiments are necessary

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

- Donald Knuth

Implementation

Speed - Writing Fast Code

- Reduce the number of loops
- Reduce what is looped over
- Reduce the operations in each pass
- Exploit special structures/data
- Keep the big picture in mind
- Use libraries, e.g.
  - Intel® MKL - implements BLAS/LAPACK

Implementation

Speed - Operations

Integer addition or subtraction

Good memory access

Floating-point addition or subtraction

Integer or Floating-point multiplication

Integer or Floating-point division

Function call

Bad memory access

Memory allocation or deallocation

Thread context switch

Implementation

Speed - Testing Fast Code

- Timing in modern CPUs is not very accurate
- Get the right time (CPU time usually not useful)
  - In C, use `time()` not `clock()`
  - Or time the whole program using the UNIX `time` command
- Tiny differences can be due to previous load on the machine
- Always use compiler optimization
- Use a profiler to identify hotspots

Implementation

Memory

- Be aware of your machine's memory hierarchy
- Consider reusing memory
  - Work arrays of a specific length (kept 0)
  - Temporary memory pools shared by parts of the algorithm
- Avoid frequent allocations and deallocations
- Be aware of `size_t` and integer overflow
- Be aware of alignment and its implications (AVX instructions)
- Be aware of the the cost of `calloc`

Implementation

Parallel Programming

- A good algorithm beats parallelism
- Linear speedup is the exception
- For an optimization algorithm, $\sqrt{\text{no. of threads}}$ speedup is already good
- But: *Better to parallelize* is a feature in an algorithm that has some value
- Deterministic parallel execution is expected
- All of this holds for GPUs as well
  - Most GPUs are not made for double precision
  - Memory bandwidth is frequently an issue

Implementation

Example - The Initial Setup

Input: Given are m sparse vectors x_i, $i = 0 \dots m-1$, with at most k nonzeros, indices ranging from $0 \dots n-1$, in packed sparse form, sorted by index.

Output: The m packed sparse vectors y_i where $y_i = \sum_{j=0}^{i} x_j$ for all $i = 0 \dots m-1$.

```
// Get input data
x = ...

for (i = 0; i < m; i++) {
  // Allocate output memory
  y_i = malloc(...)

  // Call function to compute y_i
  computeYi(i,x,y_i)

}
```

Implementation

Example - In- and Output

In- and Output

```
0:x[3]= 0.01 x[7]=-0.15
1:x[2]= 0.42 x[6]=-0.29 x[9]=-0.23
2:x[0]=-0.15 x[3]=-0.24
==========================================
0:x[3]= 0.01 x[7]=-0.15
1:x[3]= 0.01 x[7]=-0.15 x[2]= 0.42 x[6]=-0.29 x[9]=-0.23
2:x[3]=-0.23 x[7]=-0.15 x[2]= 0.42 x[6]=-0.29 x[9]=-0.23 x[0]=-0.15
```

Implementation

Example - A First (Bad) Version 1/2

```c
int computeYi(int len, int n,
              double** inValues, int** inIndices, int* inLen,
              double* outValues, int* outIndices, int* outLen) {
    int i, j; double* workValues = NULL;

    // Allocate a work array
    workValues = calloc(n, sizeof(double));
    if (!workValues) return STATUS_OOM;

    for (i = 0; i <= len; i++) {
        // Scatter
        for (j = 0; j < inLen[i]; j++)
            workValues[inIndices[i][j]] += inValues[i][j];
    }
```

Implementation

Example - A First (Bad) Version 2/2

```
    // Dense gather
    *outLen = 0;
    for (j = 0; j < n; j++) {
        if (fabs(workValues[j]) >= ZERO_TOL) {
            outValues[*outLen] = workValues[j];
            outIndices[*outLen] = j;
            (*outLen)++;
        }
    }

    // Free work array
    free(workValues);

    return 0;
}
```

Implementation

Example - A Simple Sparse Version 1/2

```c
flag = calloc(n, sizeof(int));
if (!flag) return STATUS_OOM;

for (i = 0; i <= len; i++) {
    // Sparse scatter
    for (j = 0; j < inLen[i]; j++) {
        ind = inIndices[i][j];
        if (flag[ind])
            workValues[ind] += inValues[i][j];
        else {
            workValues[ind] = inValues[i][j];
            outIndices[(*outLen)++] = ind;
            flag[ind] = 1;
        }
    }
}
```

Implementation

Example - A Simple Sparse Version 2/2

```
    // Sparse gather
    j = 0;
    for (i = 0; i < (*outLen); i++) {
        ind = outIndices[i];
        if (fabs(workValues[ind]) >= ZERO_TOL) {
            outValues[j] = workValues[ind];
            outIndices[j] = ind;
            j++;
        }
    }
    *outLen = j;

    free(flag);
```

Implementation

Example - Results Dense vs. Sparse

n	m	k	dense (sec)	sparse (sec)
1000	10	10	0.02	0.00
10000	10	10	0.00	0.01
100000	10	10	0.00	0.00
1000000	10	10	0.03	0.01
10000000	10	10	0.29	0.02
100000000	10	10	28.59	0.02

Implementation

Example - Reuse Memory

```
// The array flag needs to be 0 for len=0
// After that, it does not have to be zeroed
for (i = 0; i <= len; i++) {
  // Scatter
  for (j = 0; j < inLen[i]; j++) {
    ind = inIndices[i][j];
    // flag[ind] <= len indicates data from previous iteration
    if (flag[ind] > len)
      workValues[ind] += inValues[i][j];
    else {
      workValues[ind] = inValues[i][j];
      outIndices[(*outLen)++] = ind;
      flag[ind] = len+1;
    }
  }
} // Sparse gather stays the same
```

Implementation

Example - Local vs. Reuse

n	m	k	local (sec)	reuse (sec)
1000000	10	10	0.02	0.04
1000000	100	10	0.09	0.01
1000000	1000	10	4.23	0.22
1000000	10000	10	27.61	17.17
1000000	100000	10	OOM	OOM

Implementation

Example - Allocating Output Memory: In a Loop

```
for (i = 0; i < m; i++) {
  size += inLen[i];

  outIndices[i] = malloc(size * sizeof(int));
  if (!outIndices[i])
    goto CLEAN_EXIT;

  outValues[i] = malloc(size * sizeof(double));
  if (!outValues[i])
    goto CLEAN_EXIT;
}
```

Implementation

Example - Allocating Output Memory: One Allocation

```
outIndices[0] = malloc((size_t) k * m * m * sizeof(int));
if (!outIndices[0])
  goto CLEAN_EXIT;

outValues[0] = malloc((size_t) k * m * m  * sizeof(double));
if (!outValues[0])
  goto CLEAN_EXIT;

for (i = 1; i < m; i++) {
  outIndices[i] = outIndices[i-1] + k*m;
  outValues[i] = outValues[i-1] + k*m;
}
```

Implementation

Example - In-loop vs. One Allocation

n	m	k	in-loop (sec)	one allocation (sec)
100000	1000	10	0.18	0.14
100000	2500	10	0.91	0.85
100000	5000	10	3.26	3.10
100000	7500	10	6.80	6.42
100000	10000	10	11.33	10.80
100000	12500	10	OOM	OOM

Implementation

Example - A Much Better Version 1/2

```
#define NOT_ZERO_VALUE 2.2250738585072014e-308
// No function call!
for (i = 0; i < m; i++) {
    // Add the i's array the work array
    for (j = 0; j < inLen[i]; j++) {
        ind = inIndices[i][j];
        if (workValues[ind] != 0.0) {
            workValues[ind] += inValues[i][j];
            // Don't allow values that are nonzero to become 0.0
            if (workValues[ind] == 0.0)
                workValues[ind] = NOT_ZERO_VALUE;
        } else {
            workValues[ind] = inValues[i][j];
            workIndices[workLen++] = ind;
        }
    }
```

Implementation

Example - A Much Better Version 2/2

```
    // Gather work array using sparse indices
    for (j = 0; j < workLen; j++) {
        ind = workIndices[j];
        if (fabs(workValues[ind]) > ZERO_TOL) {
            outValues[i][outLen[i]] = workValues[ind];
            outIndices[i][outLen[i]] = ind;
            outLen[i]++;
        }
    }
}
```

Implementation

Example - function vs. single loop

n	m	k	function (sec)	single loop (sec)
100000	1000	10	0.15	0.08
100000	1000	100	1.08	0.46
100000	1000	1000	4.48	1.22
100000	1000	10000	27.43	2.65
100000	1000	100000	OOM	OOM
1000000	10000	10	17.17	7.08

Implementation

Example - A Memory Saving Version 1/2

```
for (i = 0; i < m; i++) {
  s = 0;t = 0;
  while (s < workLen || t < inLen[i]) {
      workInd = (s < workLen) ? workIndices[s] : INT_MAX;
      iInd = (t < inLen[i]) ? inIndices[i][t] : INT_MAX;
      if (workInd == iInd) {
        if (fabs(workValues[s] + inValues[i][t]) > ZERO_TOL) {
          outIndices[i][outLen[i]] = workInd;
          outValues[i][outLen[i]] = workValues[s] + inValues[i][t];
          outLen[i]++;
        }
        s++; t++;
      }
```

Implementation

Example - A Memory Saving Version 2/2

```
        else if (workInd < iInd) {
          outIndices[i][outLen[i]] = workInd;
          outValues[i][outLen[i]] = workValues[s];
          outLen[i]++; s++;
        }
        else if (workInd > iInd) {
          outIndices[i][outLen[i]] = iInd;
          outValues[i][outLen[i]] = inValues[i][t];
          outLen[i]++; t++;
        }
    }
    for (j = 0; j < outLen[i]; j++) { // Not needed for i = m-1
        workIndices[j] = outIndices[i][j];
        workValues[j] = outValues[i][j];
    }
    workLen = outLen[i];
}
```

Implementation

Example - High Memory vs. Low Memory

n	m	k	high mem (sec)	low mem (sec)
1000000	1000	10	0.11	0.08
1000000	1000	100	0.75	0.67
1000000	1000	1000	5.55	5.04
1000000	1000	10000	14.31	14.02

Conclusions

Some Final Advice

Perfect is the enemy of good

Le mieux est le mortel ennemi du bien

Better a diamond with a flaw than a pebble without one

Conclusions

Some Things to Keep in Mind

- This sounds daunting!
- To justify research, small improvements are good enough
- Algorithms evolve from failure
- Programming is learned by doing it
- I never regretted coding something up
- I only regretted NOT coding something up (earlier)

Thank you for your attention.

Philipp.Christophel@sas.com