Increasing the Performance of the Canadian Hydrological Model using Lookup Tables

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Binben Wang

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

The climate of cold regions is fragile and could be easily threatened by human activities. Hydrological processes play an important role in the climate of cold regions, and using computational models to simulate cold-region hydrological processes helps people understand past hydrological events and predict future ones. With the need to get more accurate simulation results, more complex computational models are often required. However, the complexity of models is often limited by available computational resources. Therefore, improving the computational efficiency of model simulations is an urgent task for hydrological researchers and software developers. The Canadian Hydrological Model (`CHM`) is a modular software package that is used to simulate cold-region hydrological processes. `CHM` uses an efficient surface discretization, unstructured triangular meshes, to reduce the number of discretization elements, which in turn decreases the complexity of cold-region hydrological models. `CHM` also employs parallelization to make models more efficient. By profiling the performance of `CHM`, we find that there are some computationally intensive functions inside `CHM` that are evaluated repeatedly. Lookup tables (LUTs) followed by optional interpolation or Taylor series approximation are common optimizations to replace such direct function evaluations. These optimizations can decrease the complexity of cold-region hydrological models further. The Function Comparator (`FunC`) is a C++ library that can automatically create one-dimensional LUTs for continuous univariate functions on uniformly spaced grids.

In this thesis, we use `FunC` to implement LUTs for two computationally intensive and repeatedly called functions in `CHM`, achieving an improvement of around 20% in the performance of `CHM` in the sense of running time on two cold-region hydrological simulations. In the first step, we identify two computationally intensive and repeatedly called functions by profiling the performance of `CHM`, determine the error tolerances and the ranges of inputs for their LUT implementations, and use `FunC` to implement linear interpolation LUTs for both functions in `CHM`. In the second step, we run `CHM` with and without LUT implementations on a cold-region hydrological simulation with a small domain. We verify that `CHM` with LUT implementations produces correct output and show that there is around an 18% improvement in the performance of `CHM`. In the third step, we run the same `CHM` with and without LUT implementations on a cold-region hydrological simulation with a large domain. We again verify that `CHM` with LUT implementations produces correct output and show that there is around a 21% improvement in the performance of `CHM`.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor, Dr. Raymond J. Spiteri for his great academic support and personal guidance in the past year and a half. It is one of my luckiest things to study under his supervision. I could never finish my M.Sc. degree without his extensive knowledge, patient instructions, and selfless help. I would like to acknowledge to my parents, Mrs. Manzhen Chen and Mr. Changqi Wang, and my elder sister, Mrs. Xinping Wang for their all-aspect support when I am so far away from home. I would like to acknowledge Dr. Kevin R. Green from the Numerical Simulation Laboratory and Dr. Christopher B. Marsh from Global Water Futures for their ingenious software packages and insightful guidance. I would like to thank other colleagues from the Numerical Simulation Laboratory for their generous help. I also would like to thank my dear friends for our lasting friendship.

Contents

List of Tables

List of Figures

List of Abbreviations

AA	Array Access
ARTS	the Atmospheric Radiative Transfer Simulator
ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
Chaste	the Cancer, Heart and Soft Tissue Environment
`CHM`	the Canadian Hydrological Model
CPU	Central Processing Unit
FFTW	Fastest Fourier Transform in the West
FLOP	Floating Point Operation
FPGA	Field Programmable Gate Array
`FunC`	Function Comparator
GB	Gigabyte
IOP	Integer Operation
JSON	JavaScript Object Notation
KB	Kilobyte
MB	Megabyte
NWP	Numerical Weather Prediction
LUT	Lookup Table
NetCDF	Network Common Data Form
ODE	Ordinary Differential Equation
RAM	Random-access Memory
RMSE	Root Mean Square Error
SS	Storge Size
SWE	Snow Water Equivalent

1 Introduction

Using computational models to simulate physical systems, like fluid dynamics, protein structures, hydrological processes, and so on, is a common and effective approach to study them. However, running these simulations becomes more and more expensive and is often limited by available computational resources due to their high complexity. For example, enlarging simulation domains naturally increases the complexity of simulations. Furthermore, more complex models are often required to make simulation results more accurate. In this thesis, we focus our attention on cold-region hydrological process simulations. Cold-region hydrological processes play an important role in the environment of cold regions. Simulations of cold-region hydrological processes help people understand past hydrological events and predict future ones (*Freeze and Harlan*, 1969). Understanding cold-region hydrological processes is important for human beings because the cold-region environment is extremely sensitive to human activities, and the mountain snow in cold regions is an important freshwater source (*Viviroli et al.*, 2007; *Duarte et al.*, 2012). Prediction of future cold-region hydrological processes alerts people to take precautions against possible natural disasters, such as blizzards, avalanches, etc.

The Canadian Hydrological Model (`CHM`) is an innovative open-source software package designed to model hydrological processes with a focus on cold-region hydrological processes (*Marsh et al.*, 2019a). Spatial heterogeneities in surface, surface energy, snow interception by vegetation, etc., impact cold-region hydrological processes significantly, and it is important to include them in simulations (*Marsh et al.*, 2019a). To capture these spatial heterogeneities, a fully distributed model is used in `CHM` (*Marsh et al.*, 2019a). The ability of widely used fully distributed, raster-based models to simulate over large extents is limited by computational resources because fully distributed, raster-based models generally over-represent the surface with unnecessarily many discretization elements in low spatial heterogeneity areas. To resolve these issues, `CHM` employs *unstructured triangular meshes* to reduce the number of discretization elements (*Marsh et al.*, 2019a). Also, Marsh et al. parallelize `CHM` to reduce the running time further (*Marsh et al.*, 2019a). These `CHM` features improve the efficiency of cold-region hydrological simulations so that we can get simulation results within a shorter time period and make possible the adoption of more complex models if necessary. We also find that some computationally intensive functions in `CHM` are evaluated millions of times during a simulation (*Marsh et al.*, 2019b). These function evaluations may consume a significant amount of computational resources. Optimizations called *Lookup Tables* (LUTs) can be used to significantly reduce computational resources required by computationally intensive function evaluations and to further improve efficiency of cold-region hydrological simulations.

A LUT is a list of key and value pairs. Values can be retrieved quickly by using keys as indices. LUTs have a wide range of applications in different areas, such as image processing and computer graphics (*Pharr and Fernando*, 2005), hardware neural networks (*Dias et al.*, 2014), and scientific computing (*Wilcox et al.*, 2011). In scientific computing, LUTs are generally used as optimizations to replace direct function evaluations. Specifically, people sample the input space of a function, precompute function values of all the sample points, and save sample point and function value pairs to LUTs. LUTs can be directly used to evaluate function values at sample points or can be followed by interpolation or Taylor series approximation to evaluate function values at other points. Both cases use many fewer *floating point operations* (FLOPs) than direct function evaluations for computationally intensive functions with the cost of some extra cache/memory accesses. Green et al. show that, in practice, adopting LUTs is an efficient and effective optimization for computationally intensive and repeatedly called functions (*Green et al.*, 2019), and Wilcox et al. show that this optimization is compatible with another optimation *parallelization* (*Wilcox et al.*, 2011). However, many programmers employ LUT optimization manually. This not only requires a lot of extra development work (*Wilcox et al.*, 2011), but it also makes programs more difficult to maintain (*Loh et al.*, 2005). To resolve these issues, Wilcox et al. create a tool called `Mesa` that can automatically generate constant interpolation LUTs for user-specified functions and calculate errors of generated LUTs (*Wilcox et al.*, 2011). Later, Green et al. develop a C++ library called *Function Comparator* (`FunC`) with a similar purpose but more functionality (*Green et al.*, 2019). `FunC` offers various types of LUTs for people to choose from and can generate LUTs by error tolerances (*Green et al.*, 2019).

In this thesis, we use `FunC` to implement LUTs for computationally intensive and repeatedly called functions in `CHM` and demonstrate that there is a significant improvement in the performance of `CHM`. Our contributions are as follows: (1) We identify two computationally intensive and repeatedly called functions in `CHM` and determine the error tolerances, ranges, and LUT types for them according to the properties and characteristics of these functions and two cold-region hydrological process simulations. (2) We use `FunC` to implement linear interpolation LUTs for identified functions and gain around 20% performance improvement in both cold-region hydrological process simulations. (3) We provide a systematic procedure of implementing LUTs for other programmers so that they can implement LUTs in their programs in a similar way.

1.1 Structure of the Thesis

In Chapter 2, we introduce the mathematical theory of LUTs and nine different kinds of LUTs with the corresponding storage strategies employed by `FunC`. In Chapter 3, we introduce applications of LUTs in different areas of Computer Science, including Scientific Computing. We also describe two open-source software packages, `FunC` and `CHM`, in depth. In Chapter 4, we use `FunC` to implement LUTs in `CHM` and evaluate the performance improvement introduced by LUT implementations. In Chapter 5, we summarize the conclusions and provide some possible future works.

2 Theory of Lookup Tables

Given a univariate function $f(x)$ defined on a closed interval $[a, b] \in \mathbb{R}$ and $n+1$ distinct points $\{x_i\}_{i=0}^n$ on $[a, b]$, the essence of LUTs is to use space to trade off increased cache usage for decreased evaluation time. Cache is a piece of small but fast storage. It stores a fraction of code and data of the running program that the *central processing unit* (CPU) is going to execute and use. CPUs can access cache in a few clock cycles to dozens of clock cycles. If we have no limitation on cache size, we can store all possible values of $f(x)$. However, the typical size of cache currently varies between a few *kilobytes* (KB) to a few *megabytes* (MB). Alternatively, we can store only $n+1$ pairs $\{(x_i, f(x_i))\}_{i=0}^n$, and evaluate $f(x)$ at any point in $[a, b]$ with the help of those pairs. One intuitive way is to find a polynomial $p_n \in \mathbb{P}_n$ such that $p_n(x_i) = f(x_i),\ i = 0, 1, \cdots, n$, where $\mathbb{P}_n$ is the set of all polynomials with degree at most n. We call $p_n(x)$ the *interpolating polynomial* for data $\{(x_i, f(x_i))\}_{i=0}^n$.

For simplicity, we assume that all functions in the thesis are continuous and have all orders of derivatives on its domain.

We recall the following theorem about existence and uniqueness of such an interpolating polynomial and the interpolation error of it. The specific version in below and its proof can be found in (*Quarteroni et al.*, 2010).

Theorem 1 *Given $n+1$ distinct points $x_0, x_1, \cdots, x_n$ and $n+1$ corresponding values $f(x_0), f(x_1), \cdots, f(x_n)$, there exists a unique polynomial $p_n(x) \in \mathbb{P}_n$, such that $p_n(x_i) = f(x_i)$ for $i = 0, 1, \cdots, n$. The polynomial $p_n(x)$ has the following form*

$$p_n(x) = \sum_{i=0}^{n} f(x_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}. \tag{2.1}$$

If $f(x)$ has a continuous derivative of order $n+1$ on $[a, b]$, the interpolation error at point x within the domain is given by:

$$E_n(x) = f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i), \tag{2.2}$$

where $\xi \in I_x$, which is the smallest interval containing $\{x_i\}_{i=0}^n$ and x. We call the absolute value of the interpolation error $|E_n(x)|$ the absolute interpolation error.

Equation (2.1) shows that $p_n(x)$ requires $\mathcal{O}(n^2)$ FLOPs per evaluation. We can rewrite Equation (2.1) to reduce the FLOPs per evaluation to $\mathcal{O}(n)$. Let us define $\bar{w}_{n+1} = \prod_{i=0}^{n}(x - x_i)$ and $w_i = \prod_{j=0, j\neq i}^{n}(x_i - x_j)^{-1}$,

then we can rewrite $p_n(x)$ as

$$\begin{aligned} p_n(x) =& \sum_{i=0}^{n} f(x_i) \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} \\ =& \sum_{i=0}^{n} \frac{f(x_i)\bar{w}_{n+1}}{x - x_i} \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j} \\ =& \bar{w}_{n+1} \sum_{i=0}^{n} \frac{w_i}{x - x_i} f(x_i). \end{aligned}$$

This is called *first form of the barycentric interpolation formula* (*Quarteroni et al.*, 2010). We can compute $\{w_i\}_{i=0}^{n}$ beforehand. Then we only use $\mathcal{O}(n)$ FLOPs to compute $\bar{w}_{n+1}$ and $\sum_{i=0}^{n} \frac{w_i}{x-x_i} f(x_i)$s in each evaluation.

Also, it is shown that there is no guarantee that $p_n(x)$ has smaller absolute interpolation errors when n tends to infinity for an arbitrary function $f(x)$; see for example (*Quarteroni et al.*, 2010). One well-known counter-example is the Runge function $f(x) = 1/(1+x^2)$. Figure 2.1 shows the Runge function on $[-5, 5]$ and its interpolating polynomials $p_5(x)$ and $p_{10}(x)$. The interpolating polynomial $p_5(x)$ is of degree five for points $\{(-5, \frac{1}{26}),\ (-3, \frac{1}{10}),\ (-1, \frac{1}{2}),\ (1, \frac{1}{2}),\ (3, \frac{1}{10}),\ (5, \frac{1}{26})\}$ and is shown with an orange line. The interpolating polynomial $p_{10}(x)$ is of degree ten for points $\{(-5, \frac{1}{26}),\ (-4, \frac{1}{17}),\ (-3, \frac{1}{10}),\ (-2, \frac{1}{5}),\ (-1, \frac{1}{2}),\ (0, 1),\ (1, \frac{1}{2}),\ (2, \frac{1}{5}),\ (3, \frac{1}{10}),\ (4, \frac{1}{17})\ (5, \frac{1}{26})\}$ and is shown with a green line. The interpolating polynomial $p_{10}(x)$ has smaller absolute interpolation errors than $p_5(x)$ at points within $[-1, 1]$, but it has much larger absolute interpolation errors than $p_5(x)$ at points around $x = \pm 4.5$. This demonstrates that absolute interpolation errors do not always decrease when we increase the degree of interpolating polynomials.

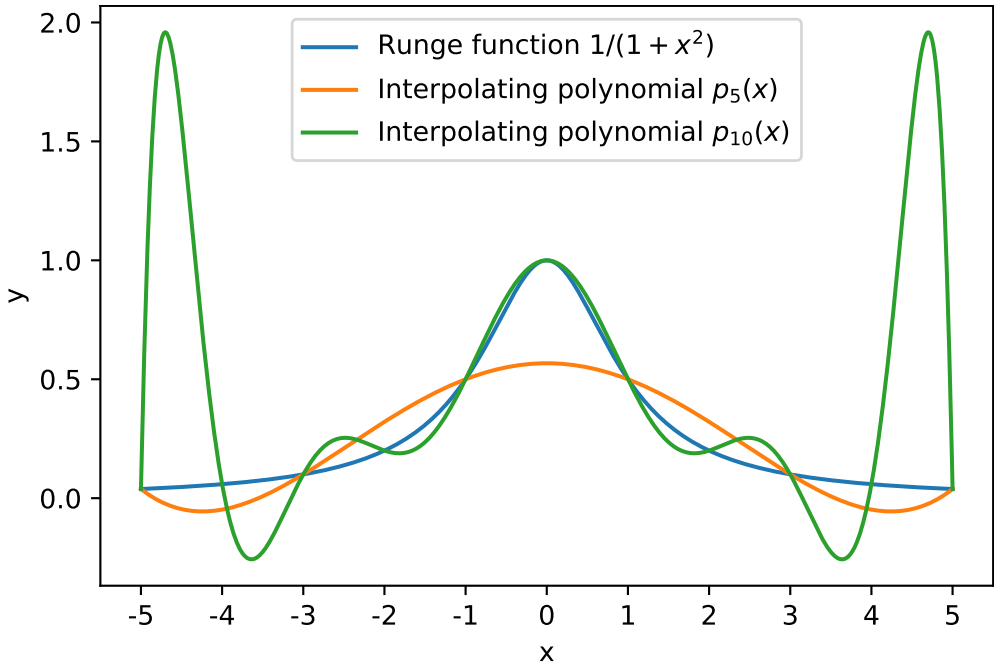

Figure 2.1: The Runge function and its interpolating polynomials $p_5(x)$ and $p_{10}(x)$ on $[-5, 5]$.

Another way to interpolate a function is to partition the domain into subintervals and apply different polynomials on each subinterval. This is called *piecewise polynomial interpolation*, also known as *spline interpolation*. In theory, the lengths of subintervals can vary. In practice, we often partition the domain into subintervals with equal lengths because (1) it is convenient for programming and (2) the theoretical upper bound of absolute errors happens in the longest subinterval with a higher probability than in other subintervals if we interpolate all subintervals in the same way.

2.1 Piecewise Polynomial Interpolation

Let us select N evenly spaced points $\{x_i\}_{i=0}^N$ on the closed interval $[a, b]$ as

$$x_i = a + i \cdot \frac{b-a}{N},\ i = 0, 1, \cdots, N.$$

All subintervals $\{[x_{i-1}, x_i]\}_{i=1}^N$ have a length of $\frac{b-a}{N}$, which is denoted by h for convenience. Because all subintervals are equivalent here, we just consider $f(x)$ and its interpolating polynomial on $[x_{i-1}, x_i]$ without loss of generality. Let $\tilde{p}_{n,i}(x)$ denote the interpolating polynomial of $f(x)$ on $[x_{i-1}, x_i]$, where n is the degree of this interpolating polynomial and i is the index of the subinterval $[x_{i-1}, x_i]$. Theorem 1 guarantees the existence and the uniqueness of $\tilde{p}_{n,i}(x)$. Let $\tilde{p}_n(x)$ denote the piecewise interpolating polynomial of $f(x)$ on $[x_0, x_n] = [a, b]$, where n has the same meaning as before. The polynomials $\tilde{p}_n(x)$ and $\tilde{p}_{n,i}(x)$ satisfy the relation

$$\tilde{p}_n(x) = \tilde{p}_{n,i}(x),\ \forall x \in [x_{i-1}, x_i],\ i = 1, 2, \cdots, N.$$

One drawback of piecewise polynomial interpolation is that $\tilde{p}_n(x)$ is not always continuous at $\{x_i\}_{i=1}^{N-1}$. But if the two endpoints of all the subintervals are used in the interpolation, we can guarantee that $\tilde{p}_n(x)$ is continuous on $[a, b]$. This is the case in all the specific piecewise polynomial interpolations with different degrees introduced next.

According to Theorem 1, the absolute interpolation error of $\tilde{p}_{n,i}(x)$ at point $x \in [x_{i-1}, x_i]$ is given by

$$|f(x) - \tilde{p}_{n,i}(x)| = \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \prod_{i=0}^{n} |(x - x_i)| \leq \frac{\max\limits_{x \in [x_{i-1}, x_i]} |f^{(n+1)}(x)|}{(n+1)!} h^{n+1} = \mathcal{O}(h^{n+1}). \tag{2.3}$$

Because $f^{(n+1)}(x)$ is continuous, so is $|f^{(n+1)}(x)|$. The maximum value of $|f^{(n+1)}(x)|$ on $[x_{i-1}, x_i]$ exists according to *the Extreme Value Theorem*; see for example (*Rudin et al.*, 1964). Similarly, the absolute interpolation errors of $\tilde{p}_{n,i}(x), i = 1, 2, \cdots, N$, are all $\mathcal{O}(h^{n+1})$. Therefore, the absolute interpolation error $|E_n(x)|$ of $\tilde{p}_n(x)$ on $[a, b]$ is $\mathcal{O}(h^{n+1})$ as well. We observe that we can make $|E_n(x)|$ as small as possible by decreasing the length of subintervals h, whereas increasing the degrees of interpolating polynomials may introduce larger absolute interpolation errors.

2.1.1 Piecewise Linear Interpolation

Because a straight line is determined by two points, we use the two endpoints, x_{i-1} and x_i, of the subinterval $[x_{i-1}, x_i]$ to formulate the linear interpolating polynomial:

$$\begin{aligned}\tilde{p}_{1,i}(x) &= f(x_{i-1}) \cdot \frac{x - x_i}{x_{i-1} - x_i} + f(x_i) \cdot \frac{x - x_{i-1}}{x_i - x_{i-1}} \\ &= f(x_{i-1}) + (f(x_i) - f(x_{i-1})) \frac{x - x_{i-1}}{x_i - x_{i-1}}.\end{aligned} \tag{2.4}$$

The absolute interpolation error of $\tilde{p}_{1,i}(x)$ on $[x_{i-1}, x_i]$ is $\mathcal{O}(h^2)$ according to Equation (2.3). Because of the arbitrariness of the subinterval we are considering, we conclude that the absolute interpolation error of $\tilde{p}_1(x)$ on $[a, b]$ is also $\mathcal{O}(h^2)$. Figure 2.2 shows a function $f(x) = \sin(x) + 0.8x$ and its linear interpolating polynomials $\tilde{p}_{1,i-1}(x)$, $\tilde{p}_{1,i}(x)$, and $\tilde{p}_{1,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise linear polynomial $\tilde{p}_{1,i}(x)$ has the same values as $f(x)$ at points x_{i-1} and x_i. The same is true for the other subintervals.

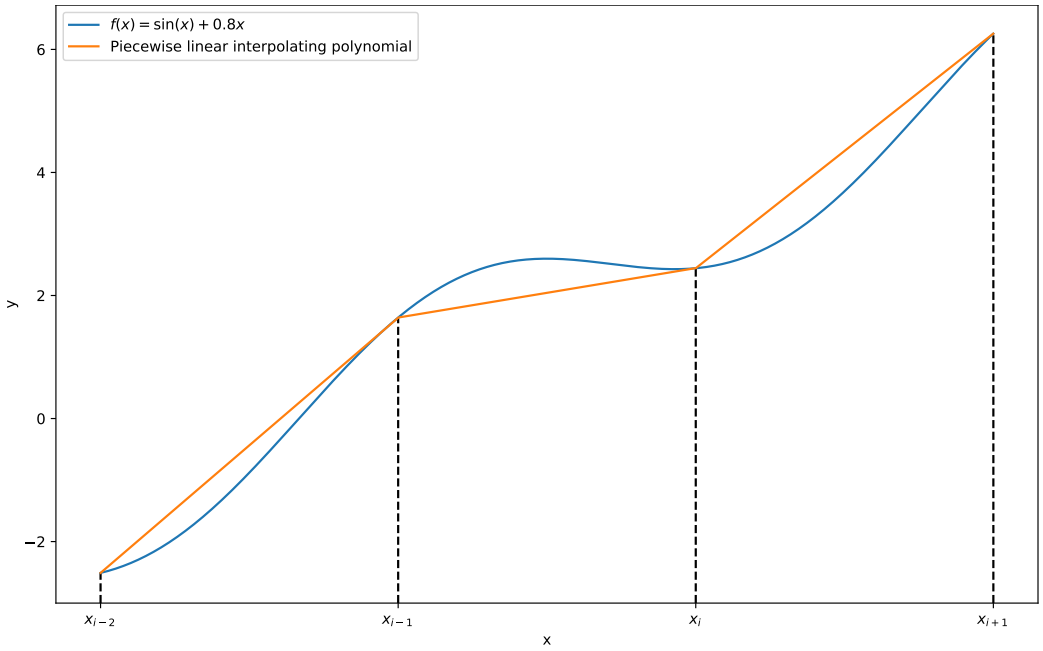

Figure 2.2: $f(x) = \sin(x) + 0.8x$ and its linear interpolating polynomials $\tilde{p}_{1,i-1}(x)$, $\tilde{p}_{1,i}(x)$, and $\tilde{p}_{1,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformLinearInterpolationTable` of `FunC` implements piecewise linear interpolation. It uses the storage strategy demonstrated in Figure 2.3 to store $\{f(x_i)\}_{i=0}^{N}$ in an array of size $N + 1$. Because

$$\frac{x - x_0}{h} = \frac{x - x_{i-1} + x_{i-1} - x_0}{x_i - x_{i-1}} = \frac{x - x_{i-1}}{x_i - x_{i-1}} + i - 1 = \frac{x - x_{i-1}}{x_i - x_{i-1}} + \left\lfloor \frac{x - x_0}{h} \right\rfloor$$

$$\Downarrow$$

$$\frac{x - x_0}{h} - \left\lfloor \frac{x - x_0}{h} \right\rfloor = \frac{x - x_{i-1}}{x_i - x_{i-1}},$$

we know the array indices of $f(x_{i-1})$ and $f(x_i)$, $\lfloor (x-x_0)/h \rfloor$ and $\lfloor (x-x_0)/h \rfloor + 1$, and $(x - x_{i-1})/(x_i - x_{i-1})$. Then the class `UniformLinearInterpolationTable` uses Equation (2.4) to return the estimated value of $f(x)$ at point x. The class `UniformLinearInterpolationTable` uses six FLOPs and one C++ explicit type conversion per evaluation. Specifically, it uses two FLOPs to calculate $(x - x_0)/h$, one C++ explicit type conversion to calculate $\lfloor (x-x_0)/h \rfloor$, one FLOP to calculate $(x-x_0)/h - \lfloor (x-x_0)/h \rfloor$, and three FLOPs to calculate the estimated value of $f(x)$ from Equation (2.4).

Value:	$f(x_0)$	$f(x_1)$	$\cdots$	$f(x_{i-1})$	$f(x_i)$	$f(x_{i+1})$	$\cdots$	$f(x_{N-1})$	$f(x_N)$
Index:	0	1	$\cdots$	$i-1$	i	$i+1$	$\cdots$	$N-1$	N

Figure 2.3: The storage strategy of the class `UniformLinearInterpolationTable`.

Because we already know $\{f(x_i)\}_{i=0}^{N}$, we can compute all the differences $\{f(x_i) - f(x_{i-1})\}_{i=1}^{N}$ beforehand and store them. The class `UniformLinearPrecomputedInterpolationTable` of `FunC` implements this precomputed piecewise linear interpolation. It stores $\{f(x_i)\}_{i=0}^{N-1}$ and $\{f(x_i) - f(x_{i-1})\}_{i=1}^{N}$ in an array of size $2N$ as expressed in Figure 2.4. Similarly, $2\lfloor (x-x_0)/h \rfloor$ and $2\lfloor (x-x_0)/h \rfloor + 1$ are array indices to access $f(x_{i-1})$ and $f(x_i) - f(x_{i-1})$. The class `UniformLinearPrecomputedInterpolationTable` uses five FLOPs, one integer operation (IOP), and one C++ explicit type conversion per evaluation. Specifically, it uses two FLOPs to calculate $(x - x_0)/h$, one C++ explicit type conversion to calculate $\lfloor (x-x_0)/h \rfloor$, one FLOP to calculate $(x-x_0)/h - \lfloor (x-x_0)/h \rfloor$, one IOP to calculate $2\lfloor (x-x_0)/h \rfloor$, and two FLOPs to calculate the estimated value of $f(x)$.

Value:	$f(x_0)$	$f(x_1)$ $-f(x_0)$	$f(x_1)$	$f(x_2)$ $-f(x_1)$	$\cdots$	$f(x_i)$	$f(x_{i+1})$ $-f(x_i)$	$\cdots$	$f(x_{N-1})$	$f(x_N)-$ $f(x_{N-1})$
Index:	0	1	2	3	$\cdots$	$2i$	$2i+1$	$\cdots$	$2N-2$	$2N-1$

Figure 2.4: The storage strategy of the class `UniformLinearPrecomputedInterpolationTable`.

2.1.2 Piecewise Quadratic Interpolation

Normally, cache is not big enough to store an entire LUT. Only a fraction of the LUT stays in cache. The rest of it stays in main memory. If an evaluation of $f(x)$ needs data that are not in cache, the CPU frees some cache and copies a new fraction containing needed data from main memory to cache. This copy operation takes much longer than accessing data in cache directly. In order to reduce the number of copy operations from main memory to cache, we need to make LUTs smaller. The size of a LUT can be approximated by the

following formula

$$\begin{aligned}\text{size of a LUT} \approx\ &\text{number of subintervals}\\ &\times \text{number of stored values per subinterval}\\ &\times \text{number of bytes for double-precision floating-point number.}\end{aligned}$$

The number of stored values per subinterval is a small positive integer. For example, the class `UniformLinearInterpolationTable` and the class `UniformLinearPrecomputedInterpolationTable` store one and two values per subinterval, respectively. In principle, the number of bytes for double-precision floating-point numbers varies according to hardware. In this thesis, we assume that a double-precision floating-point number takes eight bytes, which is the typical size that C++ uses to store double-precision floating-point numbers. To reduce the size of a LUT significantly, we need to reduce the number of subintervals. Because the number of subintervals is proportional to the reciprocal of h and the absolute interpolation error of a piecewise polynomial interpolation of degree n is $\mathcal{O}(h^{n+1})$, we can increase the degree of a piecewise polynomial interpolation to make h larger for a given error.

A quadratic interpolation needs three distinct points. We already have two endpoints. The most common candidate for the third node is the midpoint $(x_{i-1} + x_i)/2$, denoted by $x_{i-\frac{1}{2}}$. The quadratic interpolating polynomial has the following form:

$$\begin{aligned}\tilde{p}_{2,i}(x) = {} & f(x_{i-1})\frac{\left(x - x_{i-\frac{1}{2}}\right)(x - x_i)}{\left(x_{i-1} - x_{i-\frac{1}{2}}\right)(x_{i-1} - x_i)} + f\left(x_{i-\frac{1}{2}}\right)\frac{(x - x_{i-1})(x - x_i)}{\left(x_{i-\frac{1}{2}} - x_{i-1}\right)\left(x_{i-\frac{1}{2}} - x_i\right)}\\ & + f(x_i)\frac{(x - x_{i-1})\left(x - x_{i-\frac{1}{2}}\right)}{(x_{i+1} - x_{i-1})\left(x_{i+1} - x_{i-\frac{1}{2}}\right)} && \text{(2.5a)}\\ = {} & a^{(0)}_{q,i-1} + a^{(1)}_{q,i-1}\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) + a^{(2)}_{q,i-1}\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^2 && \text{(2.5b)}\\ = {} & a^{(0)}_{q,i-1} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\cdot\left(a^{(1)}_{q,i-1} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\cdot a^{(2)}_{q,i-1}\right), && \text{(2.5c)}\end{aligned}$$

where $a^{(0)}_{q,i-1} = f(x_{i-1})$, $a^{(1)}_{q,i-1} = -3f(x_{i-1}) + 4f\left(x_{i-\frac{1}{2}}\right) - f(x_i)$ and $a^{(2)}_{q,i-1} = 2f(x_{i-1}) - 4f\left(x_{i-\frac{1}{2}}\right) + 2f(x_i)$. Equation (2.5a), Equation (2.5b), and Equation (2.5c) are three different forms of $\tilde{p}_{2,i}(x)$. Equation (2.5a) is called *Lagrange's form*. It is easy to verify that $\tilde{p}_{2,i}(x)$ has the same values as $f(x)$ at points x_{i-1}, $x_{i-1/2} = (x_{i-1} + x_i)/2$, and x_i, in this form. Equation (2.5b) is called *power form* and written in ascending order of the degrees of x. Equation (2.5c) is called *Horner's form*. It reduces the number of FLOPs and rounding errors when evaluating a polynomial. The absolute interpolation errors of $\tilde{p}_{2,i}(x)$ and $\tilde{p}_2(x)$ are both $\mathcal{O}(h^3)$. Figure 2.5 shows $f(x) = \sin(x) + 0.8x$ and its quadratic interpolating polynomials $\tilde{p}_{2,i-1}(x)$, $\tilde{p}_{2,i}(x)$, and $\tilde{p}_{2,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise quadratic polynomial $\tilde{p}_{2,i}(x)$ has the same values as $f(x)$ at points x_{i-1}, $x_{i-1/2} = (x_{i-1} + x_i)/2$, and x_i. The same is true for the other subintervals.

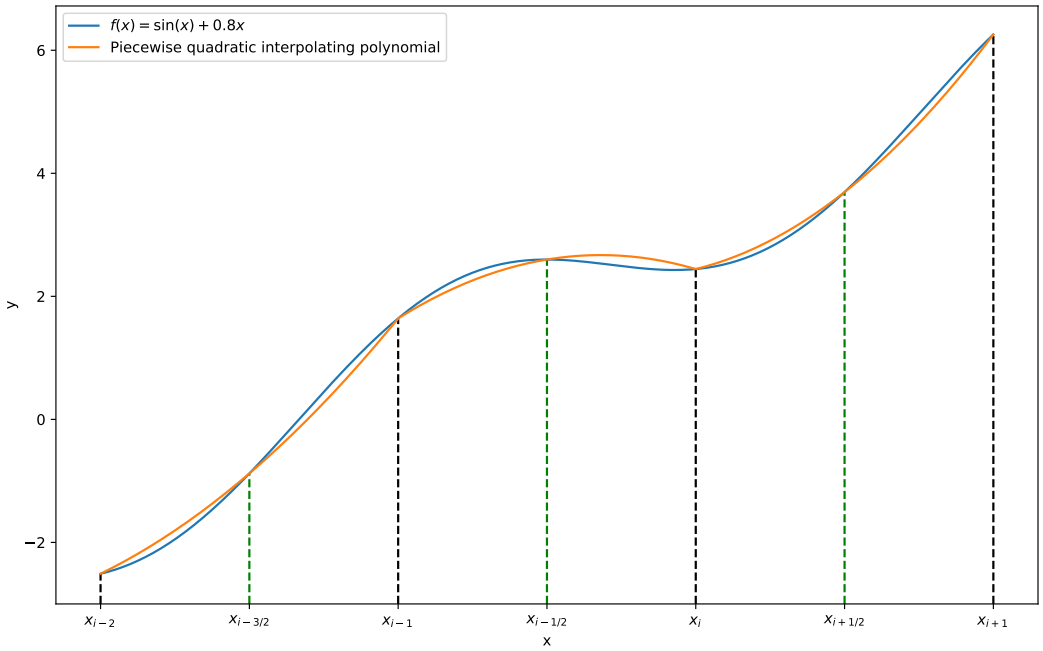

Figure 2.5: $f(x) = \sin(x) + 0.8x$ and its quadratic interpolating polynomials $\tilde{p}_{2,i-1}(x)$, $\tilde{p}_{2,i}(x)$, and $\tilde{p}_{2,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformQuadraticPrecomputedInterpolationTable` of `FunC` stores $\{a_{q,i}^{(0)}, a_{q,i}^{(1)}, a_{q,i}^{(2)}\}_{i=0}^{N-1}$ in an array of size $3N$ as expressed in Figure 2.6. We can use $3\lfloor(x - x_0)/h\rfloor$, $3\lfloor(x - x_0)/h\rfloor+1$, and $3\lfloor(x - x_0)/h\rfloor+2$ as indices to access $a_{q,i-1}^{(0)}$, $a_{q,i-1}^{(1)}$, and $a_{q,i-1}^{(2)}$. The class `UniformQuadraticPrecomputedInterpolationTable` uses seven FLOPs, one IOP, and one C++ explicit type conversion per evaluation. Specifically, it uses two FLOPs to calculate $(x - x_0)/h$, one C++ explicit type conversion to calculate $\lfloor(x - x_0)/h\rfloor$, one FLOP to calculate $(x - x_0)/h - \lfloor(x - x_0)/h\rfloor$, one IOP to calculate $3\lfloor(x - x_0)/h\rfloor$, and four FLOPs to calculate the estimated value of $f(x)$ by Equation (2.5c).

Value:	$a_{q,0}^{(0)}$	$a_{q,0}^{(1)}$	$a_{q,0}^{(2)}$	$\cdots$	$a_{q,i}^{(0)}$	$a_{q,i}^{(1)}$	$a_{q,i}^{(2)}$	$\cdots$	$a_{q,N-1}^{(0)}$	$a_{q,N-1}^{(1)}$	$a_{q,N-1}^{(2)}$
Index:	0	1	2		$3i$	$3i+1$	$3i+2$		$3N-3$	$3N-2$	$3N-1$

Figure 2.6: The storage strategy of the class `UniformQuadraticPrecomputedInterpolationTable`.

For example, suppose $f(x)$ is a continuous function defined on $[0, 10]$ and has all orders of derivatives. We want a piecewise interpolating polynomial of $f(x)$ with an error tolerance 10^{-12}. If we adopt piecewise linear interpolation, the length of subintervals is $\mathcal{O}(10^{-6})$, and the number of subintervals is $\mathcal{O}(10^7)$. So the class `UniformLinearInterpolationTable` needs $\mathcal{O}(10^7) \times 1 \times 8$ bytes $\approx \mathcal{O}(10^2)$ MB. If we adopt piecewise quadratic interpolation, the length of subintervals is $\mathcal{O}(10^{-4})$, and the number of subintervals is $\mathcal{O}(10^5)$. The class `UniformQuadraticPrecomputedInterpolationTable` needs $\mathcal{O}(10^5) \times 3 \times 8$ bytes $\approx \mathcal{O}(1)$ MB. Although

these only are loose upper bounds, they demonstrate that increasing the degree of piecewise polynomial interpolation reduces the required size of LUTs significantly.

2.1.3 Piecewise Cubic Interpolation

To reduce the size of a LUT even further, we use four points: $x_{i-1},\ x_{i-\frac{2}{3}} = \frac{2}{3}x_{i-1} + \frac{1}{3}x_i,\ x_{i-\frac{1}{3}} = \frac{1}{3}x_{i-1} + \frac{2}{3}x_i$, and x_i to derive the cubic interpolating polynomial on $[x_{i-1}, x_i]$:

$$
\begin{aligned}
\tilde{p}_{3,i}(x) = {} & f(x_{i-1}) \frac{\left(x - x_{i-\frac{1}{3}}\right)\left(x - x_{i-\frac{2}{3}}\right)(x - x_i)}{\left(x_{i-1} - x_{i-\frac{1}{3}}\right)\left(x_{i-1} - x_{i-\frac{2}{3}}\right)(x_{i-1} - x_i)} \\
& + f\left(x_{i-\frac{1}{3}}\right) \frac{(x - x_{i-1})\left(x - x_{i-\frac{2}{3}}\right)(x - x_i)}{\left(x_{i-\frac{1}{3}} - x_{i-1}\right)\left(x_{i-\frac{1}{3}} - x_{i-\frac{2}{3}}\right)\left(x_{i-\frac{1}{3}} - x_i\right)} \\
& + f\left(x_{i-\frac{2}{3}}\right) \frac{(x - x_{i-1})\left(x - x_{i-\frac{1}{3}}\right)(x - x_i)}{\left(x_{i-\frac{2}{3}} - x_{i-1}\right)\left(x_{i-\frac{2}{3}} - x_{i-\frac{1}{3}}\right)\left(x_{i-\frac{2}{3}} - x_i\right)} \\
& + f(x_i) \frac{(x - x_{i-1})\left(x - x_{i-\frac{1}{3}}\right)\left(x - x_{i-\frac{2}{3}}\right)}{(x_i - x_{i-1})\left(x_i - x_{i-\frac{1}{3}}\right)\left(x_i - x_{i-\frac{2}{3}}\right)} \\
= {} & a_{c,i-1}^{(0)} + a_{c,i-1}^{(1)} \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) + a_{c,i-1}^{(2)} \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^2 + a_{c,i-1}^{(3)} \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^3 \\
= {} & a_{c,i-1}^{(0)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) \cdot \left(a_{c,i-1}^{(1)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) \cdot \left(a_{c,i-1}^{(2)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) \cdot a_{c,i-1}^{(3)}\right)\right),
\end{aligned}
\tag{2.6}
$$

where

$$
\begin{aligned}
a_{c,i-1}^{(0)} &= f(x_{i-1}), \\
a_{c,i-1}^{(1)} &= -\frac{11}{2} f(x_{i-1}) + 9f\left(x_{i-\frac{2}{3}}\right) - \frac{9}{2} f\left(x_{i-\frac{1}{3}}\right) + f(x_i), \\
a_{c,i-1}^{(2)} &= 9f(x_{i-1}) - \frac{45}{2} f\left(x_{i-\frac{2}{3}}\right) + 18f\left(x_{i-\frac{1}{3}}\right) - \frac{9}{2} f(x_i), \\
a_{c,i-1}^{(3)} &= -\frac{9}{2} f(x_{i-1}) + \frac{27}{2} f\left(x_{i-\frac{2}{3}}\right) - \frac{27}{2} f\left(x_{i-\frac{1}{3}}\right) + \frac{9}{2} f(x_i).
\end{aligned}
$$

Absolute interpolation errors of $\tilde{p}_{3,i}(x)$ and $\tilde{p}_3(x)$ are both $\mathcal{O}(h^4)$. Figure 2.7 shows $f(x) = \sin(x) + 0.8x$ and its cubic interpolating polynomials $\tilde{p}_{3,i-1}(x)$, $\tilde{p}_{3,i}(x)$, and $\tilde{p}_{3,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise cubic polynomial $\tilde{p}_{3,i}(x)$ has the same values as $f(x)$ at points x_{i-1}, $x_{i-2/3} = 2/3x_{i-1} + 1/3x_i$, $x_{i-1/3} = 1/3x_{i-1} + 2/3x_i$, and x_i. The same is true for the other subintervals.

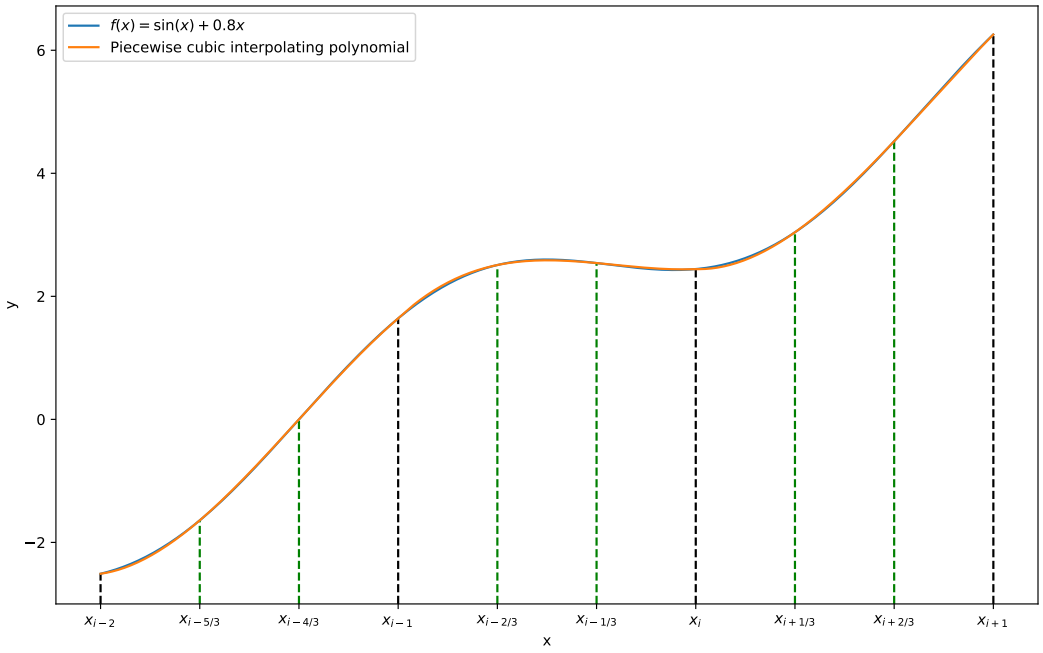

Figure 2.7: $f(x) = \sin(x) + 0.8x$ and its piecewise cubic interpolating polynomials $\tilde{p}_{3,i-1}(x)$, $\tilde{p}_{3,i}(x)$, and $\tilde{p}_{3,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformCubicPrecomputedInterpolationTable` of `FunC` stores $\{a_{c,i}^{(0)}, a_{c,i}^{(1)}, a_{c,i}^{(2)}, a_{c,i}^{(3)}\}_0^{N-1}$ in an array of size $4N$ as expressed in Figure 2.8. We can use $4\lfloor (x - x_0)/h \rfloor$, $4\lfloor (x - x_0)/h \rfloor + 1$, $4\lfloor (x - x_0)/h \rfloor + 2$, and $4\lfloor (x - x_0)/h \rfloor + 3$ as indices to access $a_{c,i-1}^{(0)}$, $a_{c,i-1}^{(1)}$, $a_{c,i-1}^{(2)}$, and $a_{c,i-1}^{(3)}$. The class `UniformCubicPrecomputedInterpolationTable` uses nine FLOPs, one IOP, and one C++ explicit type conversion per evaluation. Specifically, it uses two FLOPs to calculate $(x - x_0)/h$, one C++ explicit type conversion to calculate $\lfloor (x - x_0)/h \rfloor$, one FLOP to calculate $(x - x_0)/h - \lfloor (x - x_0)/h \rfloor$, one IOP to calculate $4\lfloor (x - x_0)/h \rfloor$, and six FLOPs to calculate the estimated value of $f(x)$ by Equation (2.6).

Value:	$a_{c,0}^{(0)}$	$a_{c,0}^{(1)}$	$a_{c,0}^{(2)}$	$a_{c,0}^{(3)}$	$\cdots$	$a_{c,i}^{(0)}$	$\cdots$	$a_{c,N-1}^{(0)}$	$a_{c,N-1}^{(1)}$	$a_{c,N-1}^{(2)}$	$a_{c,N-1}^{(3)}$
Index:	0	1	2	3		$4i$		$4N-4$	$4N-3$	$4N-2$	$4N-1$

Figure 2.8: The storage strategy of the class `UniformCubicPrecomputedInterpolationTable`.

Let us continue with the previous example. We want a piecewise interpolating polynomial of a continuous function $f(x)$ defined on $[0, 10]$ with an error tolerance 10^{-12}. If we adopt piecewise cubic interpolation, the length of subintervals is $\mathcal{O}(10^{-3})$, and the number of subintervals is $\mathcal{O}(10^4)$. The class `UniformCubicPrecomputedInterpolationTable` only needs $\mathcal{O}(10^4) \times 4 \times 8$ bytes $\approx \mathcal{O}(10^{-1})$ MB.

2.1.4 Piecewise Cubic Hermite Interpolation

One drawback of all the previous piecewise polynomial interpolations is that their first derivative is not continuous at $\{x_i\}_{i=1}^{n-1}$. If piecewise interpolations are required to have continuous derivatives, we can use another strategy called *Hermite interpolation*; see for example (*Quarteroni et al.*, 2010). We still consider $[x_{i-1}, x_i]$. First, we denote the piecewise cubic Hermite interpolating polynomial on $[x_{i-1}, x_i]$ as $H_{3,i}(x)$ and the piecewise cubic Hermite interpolating polynomial on $[a, b]$ as $H_3(x)$, where i is the index of the subinterval $[x_{i-1}, x_i]$ and 3 is the degree of the piecewise cubic Hermite interpolating polynomial. The polynomials $H_3(x)$ and $H_{3,i}(x)$ satisfy the relation

$$H_3(x) = H_{3,i}(x),\ \forall x \in [x_{i-1}, x_i],\ i = 1, 2, \cdots, N.$$

Also, $H_{3,i}(x)$ satisfies

$$H_{3,i}(x_{i-1}) = f(x_{i-1}),\ H_{3,i}(x_i) = f(x_i),$$
$$H_{3,i}{}'(x_{i-1}) = f'(x_{i-1}),\ H_{3,i}{}'(x_i) = f'(x_i).$$

$H_{3,i}(x)$ is a cubic polynomial and has the following format:

$$\begin{aligned}
H_{3,i}(x) &= f(x_{i-1})\left(1 + 2\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\left(\frac{x - x_i}{x_{i-1} - x_i}\right)^2 + f'(x_{i-1})\,(x - x_{i-1})\left(\frac{x - x_i}{x_{i-1} - x_i}\right)^2 \\
&\quad + f(x_i)\left(1 + 2\frac{x - x_i}{x_{i-1} - x_i}\right)\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^2 + f'(x_i)\,(x - x_i)\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^2 \\
&= a_{h,i-1}^{(0)} + a_{h,i-1}^{(1)}\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right) + a_{h,i-1}^{(2)}\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^2 + a_{h,i-1}^{(3)}\left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)^3 \\
&= a_{h,i-1}^{(0)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\cdot\left(a_{h,i-1}^{(1)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\cdot\left(a_{h,i-1}^{(2)} + \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\cdot a_{h,i-1}^{(3)}\right)\right),
\end{aligned} \tag{2.7}$$

where

$$\begin{aligned}
a_{h,i-1}^{(0)} &= f(x_{i-1}), \\
a_{h,i-1}^{(1)} &= f'(x_{i-1})(x_i - x_{i-1}), \\
a_{h,i-1}^{(2)} &= -3f(x_{i-1}) + 3f(x_i) - (2f'(x_{i-1}) + f'(x_i))(x_i - x_{i-1}), \\
a_{h,i-1}^{(3)} &= 2f(x_{i-1}) - 2f(x_i) + (f'(x_{i-1}) + f'(x_i))(x_i - x_{i-1}).
\end{aligned}$$

It is shown that the absolute interpolation errors of $H_{3,i}(x)$ and $H_3(x)$ are $\mathcal{O}(h^4)$; see for example (*Quarteroni et al.*, 2010). Figure 2.9 shows $f(x) = \sin(x) + 0.8x$ and its cubic Hermite interpolating polynomials $H_{3,i-1}(x)$, $H_{3,i}(x)$, and $H_{3,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise cubic interpolating polynomial $H_{3,i}(x)$ has the same values and first derivatives as $f(x)$ at points x_{i-1} and x_i. The same is true for the other subintervals.

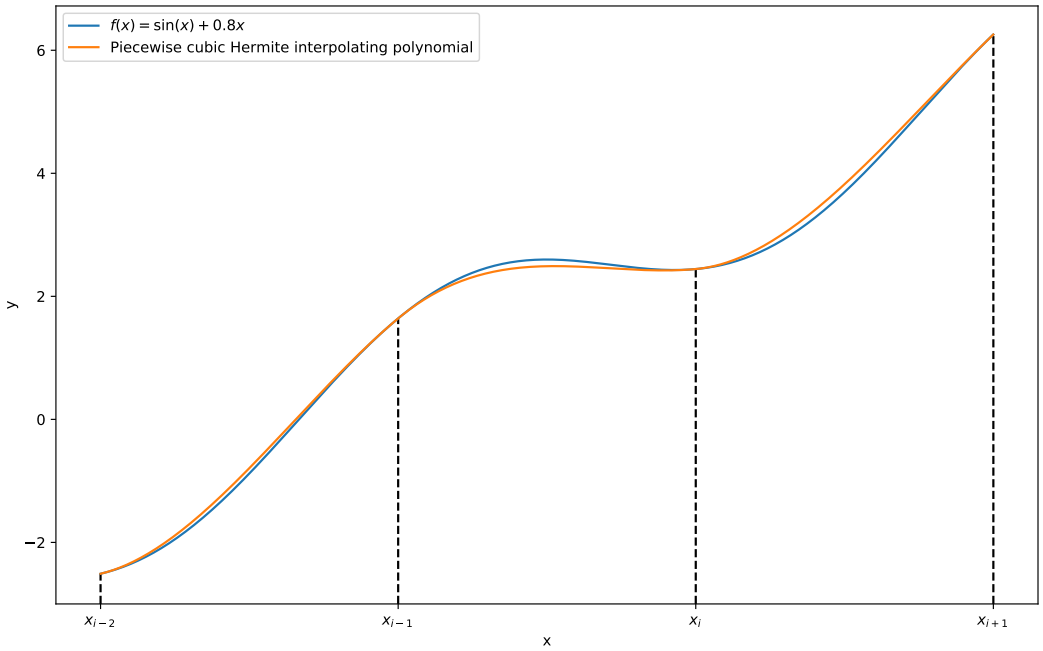

Figure 2.9: $f(x) = \sin(x) + 0.8x$ and its cubic Hermite interpolating polynomials $H_{3,i-1}(x)$, $H_{3,i}(x)$, and $H_{3,i+1}(x)$ on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformCubicHermiteTable` of `FunC` uses the same strategy to store $\{a_{h,i}^{(0)}, a_{h,i}^{(1)}, a_{h,i}^{(2)}, a_{h,i}^{(3)}\}_{i=0}^{N-1}$ as the class `UniformCubicPrecomputedInterpolationTable`. The class `UniformCubicHermiteTable` uses nine FLOPs, one IOP, and one C++ explicit type conversion per evaluation. It uses two FLOPs to calculate $(x - x_0)/h$, one C++ explicit type conversion to calculate $\lfloor (x - x_0)/h \rfloor$, one FLOP to calculate $(x - x_0)/h - \lfloor (x - x_0)/h \rfloor$, one IOP to calculate $4\lfloor (x - x_0)/h \rfloor$, and six FLOPs to calculate the estimated value of $f(x)$ by Equation (2.7). Because $H'_{3,i}(x_i) = H'_{3,i+1}(x_i) = f'(x_i),\ i = 1, 2, \cdots, N-1$ and $H'_{3,i}(x)$ is continuous on the open interval $(x_{i-1}, x_i),\ i = 1, 2, \cdots, N-1$, $H_3{}'(x)$ is continuous on $[a, b]$.

Let us continue with the previous example. We want a piecewise interpolating polynomial of a continuous function $f(x)$ defined on $[0, 10]$ with an error tolerance 10^{-12}. If we adopt piecewise cubic Hermite interpolation, the length of subintervals is $\mathcal{O}(10^{-3})$, and the number of subintervals is $\mathcal{O}(10^4)$. The class `UniformCubicHermiteTable` only needs $\mathcal{O}(10^4) \times 4 \times 8$ bytes $\approx \mathcal{O}(10^{-1})$ MB.

2.2 Piecewise Taylor Series Approximation

Taylor series approximation is another way to approximate function values by using information about the function at points. Specifically, Taylor series approximation uses the function value and the function's derivatives at a point to evaluate function values at the neighbor of the point. Let us recall *Taylor's Theorem* first. Taylor's Theorem is a well-known theorem, and it has many variants. The following statement is from (*Thomas et al.*, 2005).

Theorem 2 (Taylor's Theorem) *If $f \in C^n[a,b]$ and $f^{(n+1)}$ exists on (a,b), then for any points $x,\ x+\Delta x \in [a,b]$ we have*

$$f(x+\Delta x) = \sum_{k=0}^{n} \frac{\Delta x^k}{k!} f^{(k)}(x) + \frac{\Delta x^{n+1}}{(n+1)!} f^{(n+1)}(\xi),$$

where ξ is a point between x and $x+\Delta x$.

We can use $\sum_{k=0}^{n} \Delta x^k f^{(k)}(x)/k!$ to approximate $f(x+\Delta x)$. This approximation is called *the Taylor series approximation of degree n at x*. The absolute error of the approximation is $|\Delta x|^{n+1}/(n+1)!|f^{(n+1)}(\xi)| = \mathcal{O}\left(|\Delta x|^{n+1}\right)$, because $|f^{(n+1)}(\xi)|$ is continuous and has a maximum value between x and $x+\Delta x$. Because the Taylor series approximation only uses values at a single point, we adopt a slightly different strategy to make Δx smaller. The strategy is to evaluate $f(x)$ at a point $x \in [a,b]$ at the nearest point among $\{x_i\}_{i=0}^{N}$. If x is exactly in the middle of two points, for example x_{i-1} and x_i, we evaluate $f(x)$ arbitrarily at the larger point, herein x_i. Because the maximum value of $|\Delta x|$ is $h/2$, the absolute error of the Taylor series approximation can be rewritten as

$$\frac{|\Delta x|^{n+1}}{(n+1)!}|f^{(n+1)}(\gamma)| = \mathcal{O}\left(|\Delta x|^{n+1}\right) = \mathcal{O}\left(h^{n+1}\right). \tag{2.8}$$

2.2.1 Piecewise Constant Taylor Series Approximation

Figure 2.10 shows $f(x) = \sin(x) + 0.8x$ and its constant Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise constant Taylor series approximation on $[x_{i-1}, (x_{i-1}+x_i)/2)$ has the same value as $f(x)$ at the point x_{i-1}, and the piecewise constant Taylor series approximation on $[(x_{i-1}+x_i)/2, x_i]$ has the same value as $f(x)$ at the point x_i. The same is true for the other subintervals. When x is exactly in the middle of two points, for example x_{i-1} and x_i, we evaluate $f(x)$ arbitrarily at the larger point, herein x_i. The absolute error of the constant Taylor series approximation is $\mathcal{O}(h)$ according to Equation (2.8).

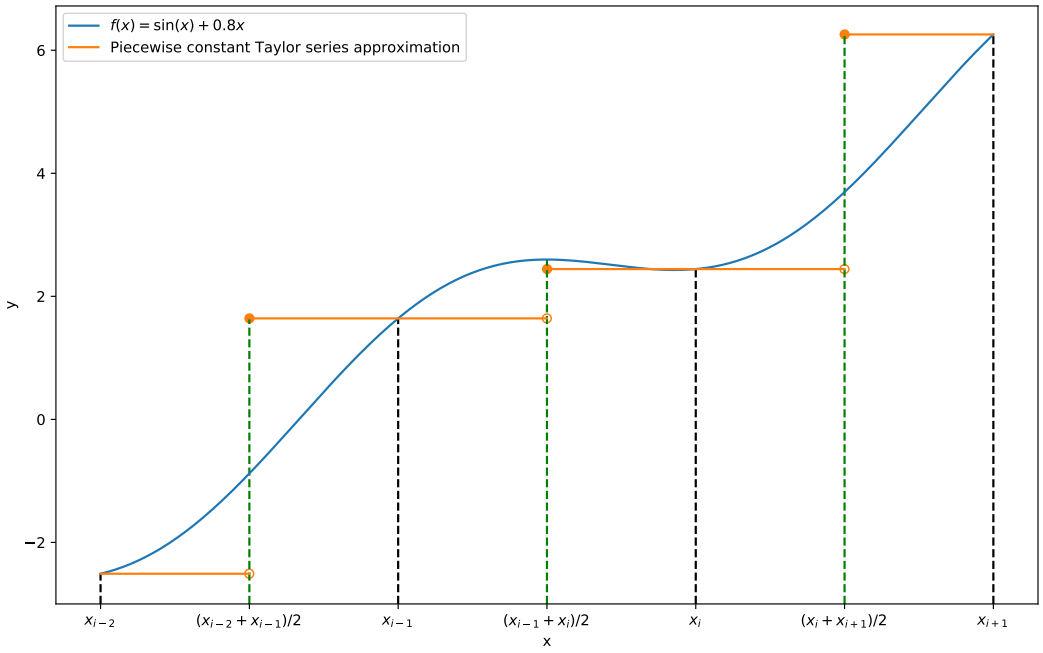

Figure 2.10: $f(x) = \sin(x) + 0.8x$ and its constant Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformConstantTaylorTable` of `FunC` implements piecewise constant Taylor series approximation. It stores the same data as the class `UniformLinearInterpolationTable`. We can use $\lfloor (x - x_0)/h + 0.5 \rfloor$ as the index to access $f(x_i)$. The class `UniformConstantTaylorTable` uses three FLOPs and one C++ explicit type conversion per evaluation. Specifically, it uses three FLOPs to calculate $(x - x_0)/h + 0.5$ and one C++ explicit type conversion to calculate $\lfloor (x - x_0)/h + 0.5 \rfloor$.

This kind of LUT is impractical for small error tolerances or large input ranges because it uses too much space. Let us continue with the previous example. We want a piecewise constant Taylor Series Approximation of a continuous function $f(x)$ defined on $[0, 10]$ with an error tolerance 10^{-12}. The length of subintervals is $\mathcal{O}(10^{-12})$, and the number of subintervals is $\mathcal{O}(10^{13})$. The class `UniformConstantTaylorTable` needs $\mathcal{O}(10^{13}) \times 1 \times 8$ bytes $\approx \mathcal{O}(10^5)$ *gigabytes* (GB).

2.2.2 Piecewise Linear Taylor Series Approximation

According to Theorem 2, the linear Taylor series approximation of $f(x)$, $\forall x \in [\frac{x_{i-1}+x_i}{2}, \frac{x_i+x_{i+1}}{2})$ can be expressed as:

$$f(x) \approx f(x_i) + f'(x_i)(x - x_i). \tag{2.9}$$

It is a straight line having the same value and first derivative as $f(x)$ at the point x_i. The absolute error of the linear Taylor series approximation is $\mathcal{O}\left(h^2\right)$ according to Equation (2.8). Figure 2.11 shows $f(x) = \sin(x) + 0.8x$ and its linear Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise linear Taylor series approximation on $[x_{i-1}, (x_{i-1} + x_i)/2)$ has the

same value and first derivative as $f(x)$ at the point x_{i-1}, and the piecewise linear Taylor series approximation on $[(x_{i-1}+x_i)/2, x_i]$ has the same value and first derivative as $f(x)$ at the point x_i. The same is true for the other subintervals. When x is exactly in the middle of two points, for example x_{i-1} and x_i, we evaluate $f(x)$ arbitrarily at the larger point, herein x_i.

The class `UniformLinearTaylorTable` of `FunC` implements piecewise linear Taylor series approximation. It stores $\{f(x_i)\}_{i=0}^{N}$ and $\{f'(x_i)\}_{i=0}^{N}$ in an array of size $2N+2$ as expressed in Figure 2.12. We can use $2\lfloor (x-x_0)/h + 0.5 \rfloor$ and $2\lfloor (x-x_0)/h + 0.5 \rfloor + 1$ as indices to access $f(x_i)$ and $f'(x_i)$. The class `UniformLinearTaylorTable` uses eight FLOPs, one IOP, and one C++ explicit type conversion per evaluation. Specifically, it uses three FLOPs to calculate $(x-x_0)/h + 0.5$, one C++ explicit type conversion to calculate $\lfloor (x-x_0)/h + 0.5 \rfloor$, one IOP to calculate $2\lfloor (x-x_0)/h + 0.5 \rfloor$, three FLOPs to calculate $x - x_i = (x-x_0) - (2\lfloor (x-x_0)/h + 0.5 \rfloor)h/2$ (noting that $x - x_0$ is calculated in first step), and two FLOPs to calculate the estimated value of $f(x)$ by Equation (2.9).

Let us continue with the previous example. We want a piecewise linear Taylor Series Approximation of a continuous function $f(x)$ defined on $[0, 10]$ with an error tolerance 10^{-12}. The length of subintervals is $\mathcal{O}(10^{-6})$, and the number of subintervals is $\mathcal{O}(10^7)$. The class `UniformLinearTaylorTable` needs $\mathcal{O}(10^7) \times 2 \times 8$ bytes $\approx \mathcal{O}(10^2)$ (MB).

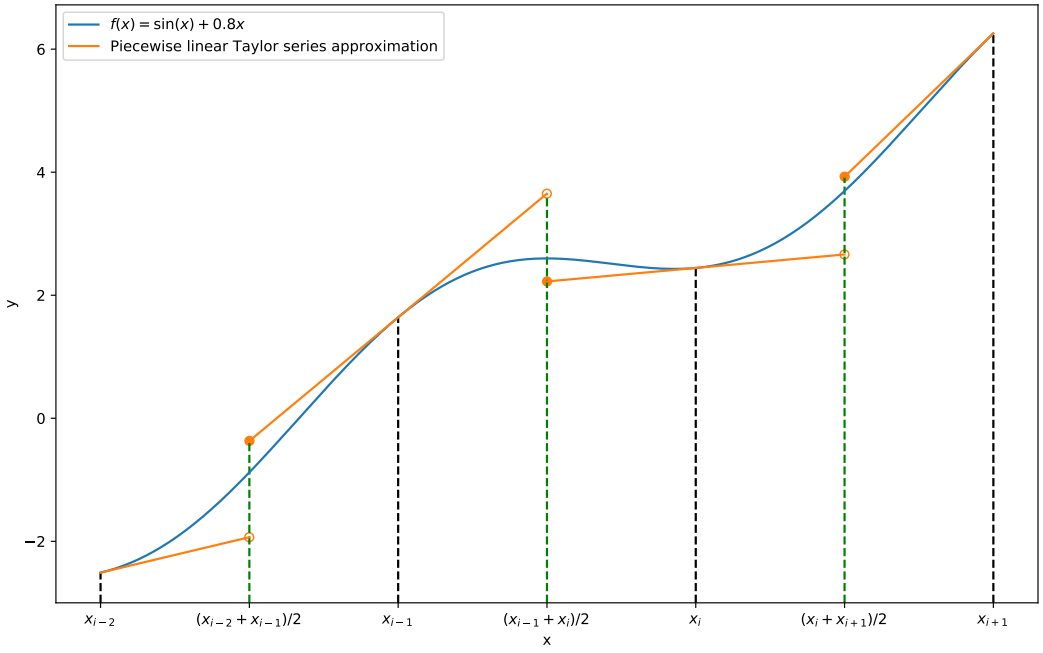

Figure 2.11: $f(x) = \sin(x) + 0.8x$ and its linear Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

Value:	$f(x_0)$	$f'(x_0)$	$f(x_1)$	$f'(x_1)$	$\cdots$	$f(x_i)$	$f'(x_i)$	$\cdots$	$f(x_N)$	$f'(x_N)$
Index:	0	1	2	3		$2i$	$2i+1$		$2N$	$2N+1$

Figure 2.12: The storage strategy of the class `UniformLinearTaylorTable`.

2.2.3 Piecewise Quadratic Taylor Series Approximation

The quadratic Taylor series approximation of $f(x)$, $\forall x \in [\frac{x_{i-1}+x_i}{2}, \frac{x_i+x_{i+1}}{2})$ can be expressed as:

$$\begin{aligned} f(x) &\approx f(x_i) + f'(x_i)(x - x_i) + \frac{f''(x_i)}{2}(x - x_i)^2 \\ &= f(x_i) + (x - x_i)\left(f'(x_i) + (x - x_i)\frac{f''(x_i)}{2}\right). \end{aligned} \tag{2.10}$$

It is a quadratic polynomial having the same value, first derivative, and second derivative as $f(x)$ at x_i. The absolute error of the quadratic Taylor series approximation is $\mathcal{O}(h^3)$ according to Equation (2.8). Figure 2.13 shows $f(x) = \sin(x) + 0.8x$ and its quadratic Taylor series approximations on$[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise quadratic Taylor series approximation on $[x_{i-1}, (x_{i-1} + x_i)/2)$ has the same value, first derivative, and second derivative as $f(x)$ at the point x_{i-1}, and the piecewise quadratic Taylor series approximation on $[(x_{i-1} + x_i)/2, x_i]$ has the same value, first derivative, and second derivative as $f(x)$ at the point x_i. The same is true for the other subintervals. When x is exactly in the middle of two points, for example x_{i-1} and x_i, we evaluate $f(x)$ arbitrarily at the larger point, herein x_i.

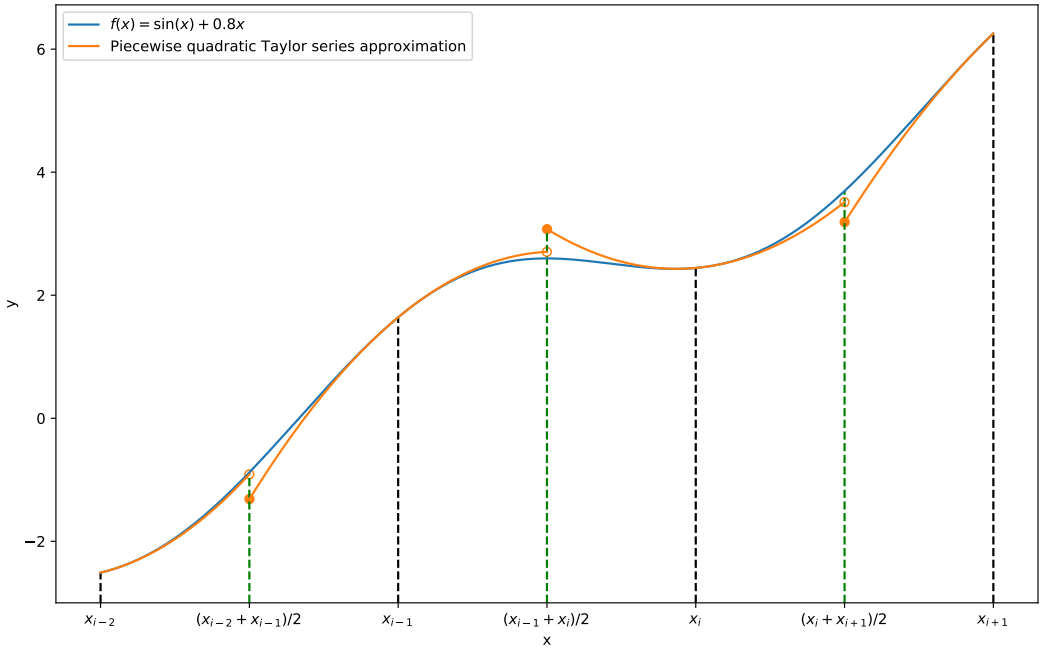

Figure 2.13: $f(x) = \sin(x) + 0.8x$ and its quadratic Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformQuadraticTaylorTable` of `FunC` implements piecewise quadratic Taylor series approximation. It stores $\{f(x_i), f'(x_i), f''(x_i)/2\}_{i=0}^{N}$ in an array of size $3N+3$ as expressed in Figure 2.14. We can use $3\lfloor(x-x_0)/h+0.5\rfloor$, $3\lfloor(x-x_0)/h+0.5\rfloor+1$, and $3\lfloor(x-x_0)/h+0.5\rfloor+2$ as indices to access $f(x_i)$, $f'(x_i)$, and $f''(x_i)/2$. The class `UniformQuadraticTaylorTable` uses ten FLOPs, one IOP, and one C++ explicit type conversion per evaluation. Specifically, it uses three FLOPs to calculate $(x-x_0)/h+0.5$, one C++ explicit type conversion to calculate $\lfloor(x-x_0)/h+0.5\rfloor$, one IOP to calculate $3\lfloor(x-x_0)/h+0.5\rfloor$, three FLOPs to calculate $x-x_i=(x-x_0)-(3\lfloor(x-x_0)/h+0.5\rfloor)h/3$ (noting that $x-x_0$ is calculated in first step), and four FLOPs to calculate the estimated value of $f(x)$ by Equation (2.10).

Let us continue with the previous example. We want a piecewise quadratic Taylor Series Approximation of a continuous function $f(x)$ defined on $[0,10]$ with an error tolerance 10^{-12}. The length of subintervals is $\mathcal{O}(10^{-4})$, and the number of subintervals is $\mathcal{O}(10^5)$. The class `UniformQuadraticTaylorTable` needs $\mathcal{O}(10^5)\times 3\times 8$ bytes $\approx \mathcal{O}(1)$ (MB).

Value:	$f(x_0)$	$f'(x_0)$	$\frac{f''(x_0)}{2}$	$\cdots$	$f(x_i)$	$f'(x_i)$	$\frac{f''(x_i)}{2}$	$\cdots$	$f(x_N)$	$f'(x_N)$	$\frac{f''(x_N)}{2}$
Index:	0	1	2		$3i$	$3i+1$	$3i+2$		$3N$	$3N+1$	$3N+2$

Figure 2.14: The storage strategy of the class `UniformQuadraticTaylorTable`.

2.2.4 Piecewise Cubic Taylor Series Approximation

The cubic Taylor series approximation of $f(x)$, $\forall x \in [\frac{x_{i-1}+x_i}{2}, \frac{x_i+x_{i+1}}{2})$ can be expressed as:

$$\begin{aligned} f(x) &\approx f(x_i)+f'(x_i)(x-x_i)+\frac{f''(x_i)}{2}(x-x_i)^2+\frac{f^{(3)}(x_i)}{6}(x-x_i)^3 \\ &= f(x_i)+(x-x_i)\left(f'(x_i)+(x-x_i)\left(\frac{f''(x_i)}{2}+(x-x_i)\frac{f^{(3)}(x_i)}{6}\right)\right). \end{aligned} \tag{2.11}$$

This is a cubic polynomial having the same value, first derivative, second derivative, and third derivative as $f(x)$ at x_i. The absolute error of the cubic Taylor series approximation is $\mathcal{O}(h^4)$ according to Equation (2.8). Figure (2.15) shows $f(x)=\sin(x)+0.8x$ and its cubic Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$. For the subinterval $[x_{i-1}, x_i]$, the piecewise cubic Taylor series approximation on $[x_{i-1}, (x_{i-1}+x_i)/2)$ has the same value, first derivative, second derivative, and third derivative as $f(x)$ at the point x_{i-1}, and the piecewise cubic Taylor series approximation on $[(x_{i-1}+x_i)/2, x_i]$ has the same value, first derivative, second derivative, and third derivative as $f(x)$ at the point x_i. The same is true for the other subintervals. When x is exactly in the middle of two points, for example x_{i-1} and x_i, we evaluate $f(x)$ arbitrarily at the larger point, herein x_i.

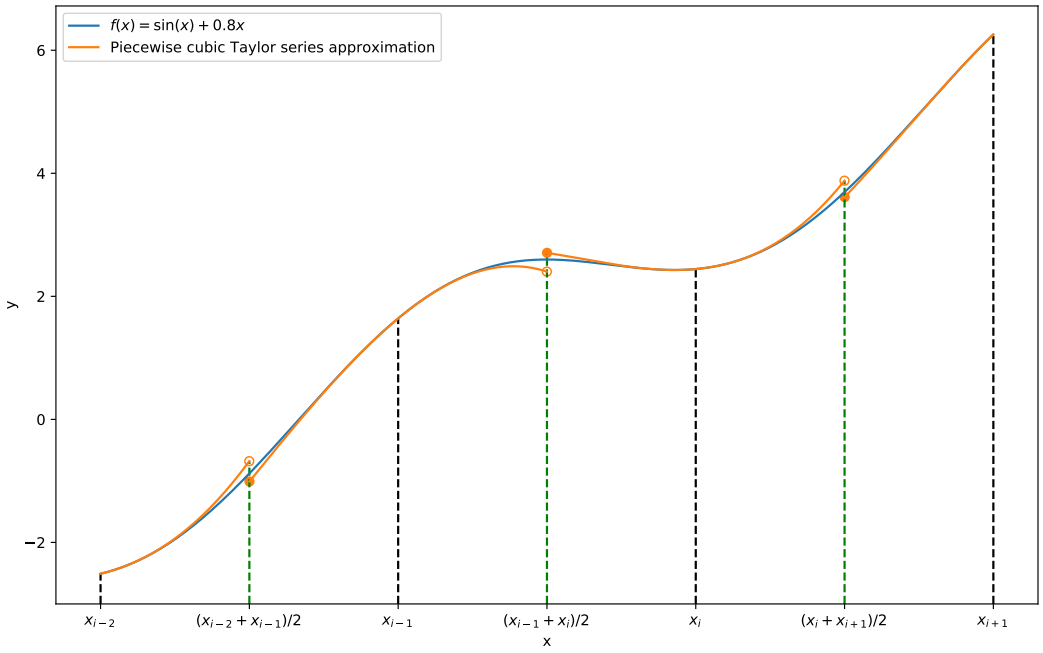

Figure 2.15: $f(x) = \sin(x) + 0.8x$ and its cubic Taylor series approximations on $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, and $[x_i, x_{i+1}]$.

The class `UniformCubicTaylorTable` of `FunC` implements piecewise cubic Taylor series approximation. It stores $\left\{f(x_i), f'(x_i), f''(x_i)/2, f^{(3)}(x_i)/6\right\}_{i=0}^{N}$ in an array of size $4N+4$ as expressed in Figure 2.16. We can use $4\lfloor(x-x_0)/h+0.5\rfloor$, $4\lfloor(x-x_0)/h+0.5\rfloor+1$, $4\lfloor(x-x_0)/h+0.5\rfloor+2$, and $4\lfloor(x-x_0)/h+0.5\rfloor+3$ as indices to access $f(x_i)$, $f'(x_i)$, $f''(x_i)/2$, and $f^{(3)}(x_i)/6$. The class `UniformCubicTaylorTable` uses twelve FLOPs, one IOP, and one C++ explicit type conversion per evaluation. It uses three FLOPs to calculate $(x-x_0)/h+0.5$, one C++ explicit type conversion to calculate $\lfloor(x-x_0)/h+0.5\rfloor$, one IOP to calculate $4\lfloor(x-x_0)/h+0.5\rfloor$, three FLOPs to calculate $x-x_i = (x-x_0) - (4\lfloor(x-x_0)/h+0.5\rfloor)h/4$ (noting that $x-x_0$ is calculated in first step), and six FLOPs to calculate the estimated value of $f(x)$ by Equation (2.11).

Let us continue with the previous example. We want a piecewise cubic Taylor Series Approximation of a continuous function $f(x)$ defined on $[0, 10]$ with an error tolerance 10^{-12}. The length of subintervals is $\mathcal{O}(10^{-3})$, and the number of subintervals is $\mathcal{O}(10^4)$. The class `UniformCubicTaylorTable` needs $\mathcal{O}(10^4) \times 4 \times 8$ bytes $\approx \mathcal{O}(10^{-1})$ (MB).

Value:	$f(x_0)$	$f'(x_0)$	$\frac{f''(x_0)}{2}$	$\frac{f^{(3)}(x_0)}{6}$	$\cdots$	$f(x_i)$	$\cdots$	$f(x_N)$	$f'(x_N)$	$\frac{f''(x_N)}{2}$	$\frac{f^{(3)}(x_N)}{6}$
Index:	0	1	2	3		$4i$		$4N$	$4N+1$	$4N+2$	$4N+3$

Figure 2.16: The storage strategy of the class `UniformCubicTaylorTable`.

Without causing ambiguity, we use class names in `FunC` to refer corresponding LUT types. Table 2.1

is a summary of degree of polynomials (n), absolute error (AE), storage size (SS) in bytes, array accesses per evaluation (AAs/Eval), FLOPs per evaluation (FLOPs/Eval), and IOPs per evaluation (IOPs/Eval) for all LUT types that `FunC` provides, where we assume all LUTs have N subintervals. We do not put C++ explicit type conversions per evaluation in the table because all LUT types use the same C++ explicit type conversion once per evaluation. The h and N satisfy the equation $h = (b - a)/N$.

Lookup Table	n	AE	SS	AAs/Eval	FLOPs/Eval	IOPs/Eval
`UniformLinearInterpolationTable`	1	$\mathcal{O}(h^2)$	$8N + 8$	2	6	0
`UniformLinearPrecomputedInterpolationTable`	1	$\mathcal{O}(h^2)$	$16N$	2	5	1
`UniformQuadraticPrecomputedInterpolationTable`	2	$\mathcal{O}(h^3)$	$24N$	3	7	1
`UniformCubicPrecomputedInterpolationTable`	3	$\mathcal{O}(h^4)$	$32N$	4	9	1
`UniformCubicHermiteTable`	3	$\mathcal{O}(h^4)$	$32N$	4	9	1
`UniformConstantTaylorTable`	0	$\mathcal{O}(h)$	$8N + 8$	1	3	0
`UniformLinearTaylorTable`	1	$\mathcal{O}(h^2)$	$16N + 16$	2	8	1
`UniformQuadraticTaylorTable`	2	$\mathcal{O}(h^3)$	$24N + 24$	3	10	1
`UniformCubicTaylorTable`	3	$\mathcal{O}(h^4)$	$32N + 32$	4	12	1

Table 2.1: Lookup Table Summary.

We note that numbers of subintervals N for the various LUTs are significantly different when they are generated by an error tolerance. Piecewise polynomial interpolations and piecewise Taylor series approximations with higher degree trade off more array accesses and FLOPs per evaluation for decreased storage sizes. They save time by reducing the number of copy operations from main memory to cache. But they also spend more time on array accesses and FLOPs per evaluation. There is no guarantee that piecewise polynomial interpolations and piecewise Taylor series approximations with higher degree have better performance than piecewise polynomial interpolations and piecewise Taylor series approximations with lower degree.

Also, Taylor series approximations require additional derivative information. They are not continuous at $\{x_i\}_{i=1}^{n-1}$. This can be problematic when we want the estimated function values to be continuous. The class `UniformCubicHermiteTable` is the only LUT that has a continuous first derivative. But it requires additional derivative information as well.

3 Literature Review

This chapter provides a brief introduction to the background of LUTs. Section 3.1.1 describes the history of LUTs and applications of LUTs before computers were invented. Section 3.1.2 describes applications of LUTs in areas of computer science outside of scientific computing. Section 3.1.3 describes applications of LUTs in scientific computing. Section 3.2 describes the C++ library `FunC` in depth (*Green et al.*, 2018). It generates different one-dimensional LUTs for continuous univariate functions and compares the performance of LUTs and direct function evaluation. Section 3.3 describes the open-source software package `CHM` in depth (*Marsh et al.*, 2019b). It is designed to simulate hydrological processes, especially cold-region hydrological processes. Finally, we propose a systematic procedure of implementing LUTs that can be easily applied to numerical computing software packages.

3.1 Lookup Tables

3.1.1 Applications before the Advent of Computers

A LUT is a list of key and value pairs. It is used to store paired information or save run-time computations. All the keys are ordered in some way so that we can use indices to access the corresponding values in the LUT quickly. For example, numerical keys are commonly ordered in ascending order or descending order. Before computers were invented, people had to evaluate complicated functions, like the exponential function, trigonometric functions, and probability density functions, by hand. It is generally difficult and time consuming to calculate these functions. Adopting LUTs made these calculations much faster. People sample the domain of a function, calculate all the function values at all the sample points, and create a LUT for the function by storing sample point and value pairs in order. Because it is impossible to store all points in the domain and their values, we cannot use the LUT to directly evaluate a function if the input point is not a sample point. We can use the value corresponding to the nearest sample point as the function estimation of the input or use the approximation methods introduced in Chapter 2 to achieve a better estimation with a few more computations. LUTs are still very useful to evaluate complicated functions today in situations when computers or calculators are not available. Table 3.1 shows a partial LUT for the cumulative distribution function of the standard normal distribution.

First decimal place of x	Second decimal place of x									
	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3.1: A partial LUT for the cumulative distribution function of the standard normal distribution.

3.1.2 Applications outside of Scientific Computing

In the 1950s, IBM introduced two new instructions related to LUTs when they were developing their latest computer system, the IBM 709 (*Amdahl*, 2013). They used LUTs to achieve code conversions between *Binary Coded Decimal* (BCD) and *American Standard Code for Information Interchange* (ASCII). They also realized addition and subtraction between BCD numbers by using LUTs.

LUTs are commonly used in image processing and computer graphics (*Battiato and Lukac*, 2008; *Gonzales and Woods*, 2002; *Pharr and Fernando*, 2005). Color-mapped images use LUTs of colors to compress images (*Battiato and Lukac*, 2008). Every pixel in an RGB image needs 24 bits, eight bits for red, green, and blue, to store color information. By using LUTs of colors, every pixel only needs to store an integer. The number of bits representing the integer is decided by the total number of colors in this image, which generally is less than the largest positive integer that 24 bits can represent. Grayscale images use a single integer to store the color information of each pixel. They can be converted to different color images by using different LUTs of colors (*Gonzales and Woods*, 2002). LUTs also make real-time color transformations of high-resolution imagery possible (*Pharr and Fernando*, 2005). Adopting LUTs simplifies a series of color operators into one single LUT. The LUT has nothing to do with the number of color operators or the complication of color operators. It significantly reduces the time used for color transformations. Both one-dimensional and three-dimensional LUTs are often used in color transformations. The number of inputs decides which type of LUTs are used. In practice, three-dimensional LUTs are used more frequently. In order to get more accurate outputs, a linear interpolation or a *trilinear interpolation*, a generalization of linear interpolation in three-dimensional space, follows one-dimensional LUTs or three-dimensional LUTs, respectively.

LUTs are also heavily used in *field programmable gate arrays* (FPGAs) (*Kuon et al.*, 2008). FPGAs are reprogrammable integrated circuits. They have five kinds of gates (basic logic operations): AND, OR, NOT, XOR, and NAND. Users combine gates arbitrarily to create more complicated logic operations. FPGAs use LUTs to implement these logic operations. A LUT in an FPGA is a truth table that has one or more inputs and one output. All the inputs and the output are binary. Inputs of a LUT are addressable indices, and outputs are results of logic operations. It is noteworthy that LUTs in FPGAs store all possible inputs and do not involve any approximation methods.

Using LUTs in hardware neural networks is also explored in many recent works (*Kumar Meher*, 2010; *Reis et al.*, 2014; *Dias et al.*, 2014). LUTs for activation functions use fewer hardware resources and are faster than real hardware functions (*Dias et al.*, 2014). In order to reduce the sizes of LUTs (hardware areas), Meher comes up with a LUT that maps a range of inputs to the same LUT entry for the hyperbolic tangent sigmoid function $f(x) = (1 + e^{-x})^{-1}$ (*Kumar Meher*, 2010). Reis et al. developed a tool that can generate hardware neural networks automatically (*Reis et al.*, 2014). This tool can generate LUTs with different sizes for the hyperbolic tangent sigmoid function (*Reis et al.*, 2014). Further, Dias et al. propose an automatic method that can generate LUTs with more user-specified parameters for different activation functions in different applications (*Dias et al.*, 2014). This method also adopts a LUT optimization based on the LUT entry access frequency (*Dias et al.*, 2014). The optimization removes table entries with small numbers of accesses and refines table entries with large numbers of accesses (*Dias et al.*, 2014).

3.1.3 Applications in Scientific Computing

Using LUTs is an alternative way to evaluate functions. It is a straightforward and efficient performance optimization. Green et al. show that LUTs are faster than direct function evaluation even for simple functions like the exponential function (*Green et al.*, 2019). So we can adopt LUTs as optimization for many function evaluations. However, LUTs have their own limitations. If a function is not evaluated enough times during the code execution, it may not be worthwhile to implement LUTs with the costs of extra development work. The extra space used by LUTs is also a non-negligible factor in some cases.

LUTs as common optimizations are used in many scientific computing libraries. One significant scientific computing library using LUTs is the *Fastest Fourier Transform in the West* (FFTW). FFTW is a C library to compute discrete Fourier transforms efficiently (*Frigo and Johnson*, 2005). It uses a LUT to keep track of plans of computed problems and returns the solution if a problem is already computed (*Frigo and Johnson*, 2005). This method is also known as *memorization*. Also, twiddle factors in fast Fourier transforms are trigonometric constant coefficients and can be precomputed and stored in LUTs, one-dimensional arrays herein, to improve performance (*Frigo and Johnson*, 2005).

Buehler et al. developed a modular program called *the atmospheric radiative transfer simulator* (ARTS) that is used to simulate atmospheric radiative transfer (*Buehler et al.*, 2005). ARTS uses two LUTs to keep track of *workspace variables* and *workspace methods* separately (*Buehler et al.*, 2005). This mechanism makes it easy to add new workspace methods and to implement online documentation of both workspace variables and workspace methods (*Buehler et al.*, 2005). Absorption cross sections are functions of frequency, pressure, temperature, and the water vapor volume mixing ratio and require a lot of computational resources in radiative transfer models (*Buehler et al.*, 2011). For each frequency, Buehler et al. precompute and store absorption cross sections into reference profiles, a kind of two-dimensional irregular LUT, to save space because not all the input combinations exist in the atmosphere (*Buehler et al.*, 2011). Then, ARTS uses an extraction strategy containing three high-order polynomial interpolations to obtain absorption cross

sections (*Buehler et al.*, 2011). Buehler et al. test ARTS with LUTs with three different satellite remote sensing instrument scenarios and find that adopting LUTs significantly reduces the running time of all scenarios (*Buehler et al.*, 2011). Also, errors introduced by LUTs are small enough that they can be safely ingored (*Buehler et al.*, 2011).

Because using LUTs manually requires a lot of extra development work (*Wilcox et al.*, 2011) and makes programs more difficult to maintain (*Loh et al.*, 2005), Wilcox et al. developed a software package called `Mesa` that can automatically analyze errors and generate the code for LUT creation and approximation (*Wilcox et al.*, 2011). They precompute and store function values at all the midpoints of subintervals and adopt constant or linear interpolation approximations (*Wilcox et al.*, 2011). Then they test `Mesa` on three molecular biology applications, and all three applications get a speed increase with speedup factor around 5 (*Wilcox et al.*, 2011). Here we need to point out that the amount of performance improvement depends on how much the cost of direct function evaluations relative to the entire cost of the code. We can get higher performance improvement when the cost of direct function evaluations is a higher proportion of the entire cost. They also show that LUT optimization works on both single- and multi-core systems (*Wilcox et al.*, 2011). A disadvantage of `Mesa` is that it cannot generate LUTs by error tolerances. Users have to change the step size manually based on the error analysis of previous LUTs. Another disadvantage is that users cannot change the type of interpolation.

3.2 Function Comparator

Function Comparator (`FunC`) is a C++ library that is used to create one-dimensional LUTs for continuous univariate functions on uniformly spaced grids (*Green et al.*, 2018). `FunC` has an abstract base class called `EvaluationImplementation` that only contains meta-information about the function we want to evaluate. It has two derived classes: `DirectEvaluation` and `UniformLookupTableGenerator`. The class `DirectEvaluation` is used to implement the direct function evaluation of a user-specified function. Specific and detailed LUT implementations are achieved by derived classes, which are items listed with an asterisk below, of the `UniformLookupTableGenerator`. The hierarchy of classes in `FunC` is:

- `EvaluationImplementation`
  - `DirectEvaluation`
  - `UniformLookupTableGenerator`
    - `UniformLinearInterpolationTable`
    - `UniformLinearPrecomputedInterpolationTable`
    - `UniformQuadraticPrecomputedInterpolationTable`
    - `UniformCubicPrecomputedInterpolationTable`
    - `UniformCubicHermiteTable`
    - `UniformConstantTaylorTable`

* `UniformLinearTaylorTable`
* `UniformQuadraticTaylorTable`
* `UniformCubicTaylorTable`

Users can use class `ImplementationComparator` to compare the performance of LUTs and direct function evaluation by the minimum running time, the mean running time, and the maximum running time (*Green et al.*, 2019).

`FunC` allows users to specify different parameters to generate desired LUTs. Specifically, users can specify a step size, an error tolerance, or an implementation size. Generating a LUT by a step size or an implementation size is trivial. `FunC` includes error analysis implicitly and can generate a LUT by an error tolerance (*Green et al.*, 2018). This is the most important one because the interpolation error of a LUT directly determines whether the LUT can be used. The error measure adopted in `FunC` is

$$E = \max_x \frac{|f(x) - \tilde{f}(x)|}{\frac{1}{2}(|f(x)| + |\tilde{f}(x)|)}, \tag{3.1}$$

where $f(x)$ is the direct function evaluation and $\tilde{f}(x)$ is the value of a LUT approximation (*Green et al.*, 2019). Because the difference between $f(x)$ and the true function value is assumed to be on the order of machine precision, we can approximately regard $f(x)$ as the true function value and E as a *relative error*. Then, Green et al. assume E can also be approximated by

$$E(h) = Ch^r,$$

where C is a constant, h is the step size of the LUT interpolation, and r is the order of the LUT interpolation (*Green et al.*, 2019). They update the step size to achieve the user-specified error tolerance by a Newton-like iteration in log-log space (*Green et al.*, 2019).

However, error measure (3.1) encounters problems when $f(x)$ is almost zero. In this case, error measure (3.1) approximately equals to 2 and does not change markedly with the value of $\tilde{f}(x)$. A better error measure is

$$E = \max_x \frac{|f(x) - \tilde{f}(x)|}{1 + |f(x)|},$$

where $f(x)$ and $\tilde{f}(x)$ have the same meaning as they do in error measure (3.1). We get an approximation of the relative error of $\tilde{f}(x)$ when $|f(x)|$ is large enough, and we get an approximation of the *absolute error* of $\tilde{f}(x)$ when $|f(x)|$ is almost zero.

From Table 2.1, we know that LUTs with higher degree (for both piecewise polynomial interpolation and Taylor series approximation) require more FLOPs per evaluation but have much smaller storage sizes. More FLOPs result in more time for evaluation, whereas smaller storage sizes result in less time for evaluation because of the higher probability of staying in cache. Green et al. design several experiments to roughly explore the effects of cache on evaluation time and find that the LUT implementation with the fewest FLOPs per evaluation takes the least evaluation time when the LUT stays in cache with a low probability or a high probability (*Green et al.*, 2019).

The Cancer, Heart and Soft Tissue Environment (Chaste) is an open-source C++ library that uses mathematical models to simulate biological and physiological problems with a focus on cardiac electrophysiology simulation, cancer cell and cell population simulation, and lung ventilation simulation (*Mirams et al.*, 2013). Chaste employs linear interpolation LUTs as optimization in cardiac electrophysiology simulations and uses a Python tool PyCml to identify computationally intensive and repeatedly called functions and to generate linear interpolation LUTs for them (*Cooper et al.*, 2006; *Green et al.*, 2019). Green et al. use `FunC` to generate quadratic and cubic interpolation LUTs for those functions with similar errors of original linear interpolation LUTs (*Green et al.*, 2019). They find that all linear interpolation LUTs but one, most quadratic interpolation LUTs, and most cubic interpolation LUTs are faster than direct function evaluations even when all LUTs stay in cache with a low probability (*Green et al.*, 2019). They also run two different cardiac electrophysiology simulations, one small and one large, and show that there are significant speed increases in the *ordinary differential equation* components of both simulations by adopting LUTs (*Green et al.*, 2019). For the large cardiac electrophysiology simulation, both linear and cubic interpolation LUTs yield a performance increase of around 30% in total CPU time for Chaste (*Green et al.*, 2019).

3.3 The Canadian Hydrological Model

The Canadian Hydrological Model (`CHM`) is an innovative open-source software package designed to model hydrological processes with a focus on cold-region hydrological processes (*Marsh et al.*, 2019b). It is about 63,000 lines of C++ code. It employs an efficient surface discretization and uses a fully distributed and modular method to model hydrological processes (*Marsh et al.*, 2019a).

Understanding cold-region hydrological processes is important for humans because the cold-region environment is extremely sensitive to human activities, and the mountain snow in cold regions is an important freshwater source (*Viviroli et al.*, 2007; *Duarte et al.*, 2012). One of the challenges in modeling cold-region hydrological processes is the various spatial heterogeneities in cold regions. Spatial heterogeneities in surface, surface energy, snow interception by vegetation, etc., impact snowmelt spatial heterogeneity, which then impacts streamflow discharge spatial heterogeneity (*Marsh et al.*, 2019a). To capture these spatial heterogeneities, a fully distributed model is used in `CHM` (*Marsh et al.*, 2019a). Fully distributed, raster-based models discretize the surface with cells with the same size and generally cannot capture the spatial heterogeneities very well. To capture the high spatial heterogeneity areas of the surface, fully distributed raster-based models have to reduce the size of cells and over-represent low spatial heterogeneity areas. This increases the computational cost significantly and makes modeling hydrological processes in large extents less possible. To reduce the number of discretization elements as well as the computational cost, Marsh et al. use *unstructured triangular meshes* to discretize a surface and its spatial heterogeneity of topography, vegetation, etc. in `CHM` (*Marsh et al.*, 2019a). Unstructured triangular meshes use triangles with various sizes in a single

discretization of a surface. It uses small triangles to discretize high spatial heterogeneity areas of the surface, like mountains and rivers, and uses large triangles to discretize low spatial heterogeneity areas, like valley bottoms. Marsh et al. also developed a multi-objective mesh generation tool *Mesher* to convert raster data into unstructured triangular meshes. The generated unstructured triangular mesh uses fewer elements than the orignal raster discretization and reduces the uncertainty of distributed models because it reduces elapsed time and the number of model parameters and forcing fields (*Marsh et al.*, 2018).

Also, in order to make full use of raster-based algorithms without much modification, `CHM` uses a k-dimensional (k-d) spatial search tree to accelerate the search of the target triangle (*Marsh et al.*, 2019a). Searching for a target cell by discretized coordinates is faster in raster-based models. However, because discretization triangles in unstructured triangular meshes are irregular, we cannot use the same indexing method to find the target triangle as in raster-based models. Instead, `CHM` uses a k-d spatial search tree to search for the target triangle (*Marsh et al.*, 2019a). As a demonstration, Marsh et al. apply this mechanism to a shadowing algorithm of Dozier and Frew (*Dozier and Frew*, 1990) and find the calculated shadow areas match the observed data well (*Marsh et al.*, 2019a).

Marsh et al. also introduce process modularity in `CHM` so that a cold-region hydrological process can be represented by a model composed of a series of ordered modules (*Marsh et al.*, 2019a). There is uncertainty of cold-region hydrological process modeling, and it is almost impossible to precisely represent a cold-region hydrological process. What we can do is use different models to represent the same cold-region hydrological process and evaluate the accuracy of each model. The process modularity in `CHM` enables users to quickly change the model of a cold-region hydrological process and make the uncertainty analysis of cold-region hydrological process modeling simpler and easier (*Marsh et al.*, 2019a). Users can add or remove modules and change the order of modules in the representation of a cold-region hydrological process (*Marsh et al.*, 2019a). Modules in `CHM` are hydrological process representations with different algorithms. There are two types of modules in `CHM`: *forcing data interpolant* and *standard module* (*Marsh et al.*, 2019b). Forcing data interpolants are interpolation processes and are used to interpolate point-scale input forcing data onto all discretization triangles (*Marsh et al.*, 2019b). A module may provide some variables to other modules and use variables from other modules (*Marsh et al.*, 2019b). This creates dependencies between modules. Modules can be combined in any way to model a cold-region hydrological process. However, if there are cyclic dependencies in the model, users have to manually remove module dependencies to eliminate cyclic dependencies. After that, `CHM` uses a topological sort to decide the execution order of modules so no modules violate their dependencies. Figure 3.1 is an example of the modules of the *Kananaskis snowpack simulation*, which we study in detail in the next chapter, and their dependencies. The execution order of modules is from left to right. Directed lines are module dependencies and show that the right modules use variables from the left modules. In terms of parallelization, all modules are either *data parallel* or *domain parallel* (*Marsh et al.*, 2019a). The difference between domain parallel modules and data parallel modules is whether information about its neighbors is needed when computing values for a triangle (*Marsh et al.*, 2019a). `CHM` groups modules

in the execution order by their parallel types to facilitate parallelization (*Marsh et al.*, 2019a).

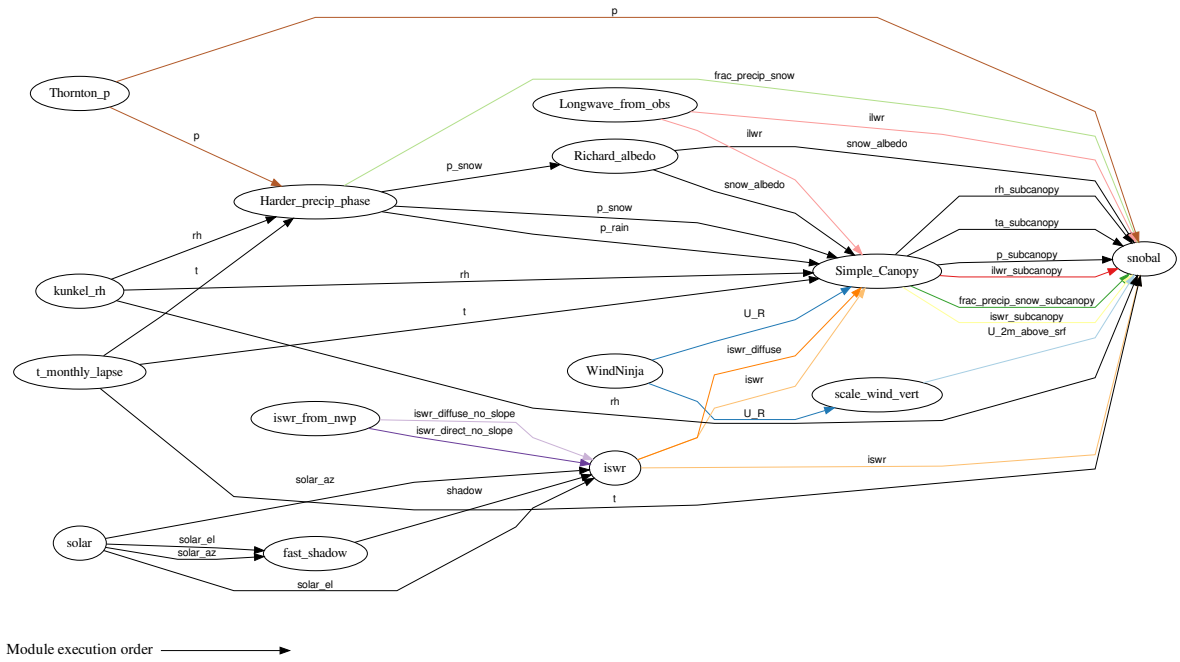

Figure 3.1: The modules and their dependencies of the Kananaskis snowpack simulation.

`CHM` takes text files or *Network Common Data Form* (NetCDF) files as input forcing data (*Marsh et al.*, 2019b). Both types of files contain forcing data from meteorological station measurements or *numerical weather prediction* (NWP) output (*Marsh et al.*, 2019a). Forcing data are in chronological order, and the time difference between any two consecutive data point is the same (*Marsh et al.*, 2019b). `CHM` also allows users to filter forcing data that are out of range or to correct values of forcing data (e.g., correct precipitation for under-catch) before any module is executed (*Marsh et al.*, 2019a). After that, `CHM` uses forcing data interpolants to interpolate forcing data onto all triangles.

All configuration information is stored in a *JavaScript Object Notation* (JSON) file (*Marsh et al.*, 2019b). Users can easily modify model parameters and forcing fields, add or remove modules, change the order of modules, and remove module dependencies by editing the corresponding JSON file or overriding specific configuration information from the command line (*Marsh et al.*, 2019a). This mechanism makes the uncertainty analysis of cold-region hydrological process modeling simpler and easier (*Marsh et al.*, 2019a).

3.4 A Systematic Procedure for Implementing LUTs

The primary goal of this thesis is to use `FunC` to generate LUTs for computationally intensive and repeated called functions in `CHM` so that we can improve the performance of `CHM` in terms of running time. In this

section, we propose a systematic procedure of using `FunC` to implement LUTs. This procedure is used for `CHM` in the next chapter and also can be easily applied to other numerical computing software packages.

The first thing we need to do is to identify computationally intensive and repeated called functions in the numerical computing software package, the performance of which we want to improve. We can look through the source code or use performance analysis software packages to identify such functions. To generate LUTs by using `FunC`, we need to determine the error tolerances and the input ranges for LUT implementations of the identified functions. If the input to a function has a physical meaning, we can first estimate the range of the input. We can then verify the input range of the function through repeated testing. Although the error tolerance is related to the accuracy of the numerical computing software package and we have to determine the magnitude of the error tolerance according to the specific software package, the single-precision error tolerance 10^{-8} and the double-precision error tolerance 10^{-16} are two good candidates that are widely used. Next, we need to make sure that the software package with LUT implementations still yields the same results as the original software package. The final step is to verify whether the LUT implementations improve the performance of the software package. Figure 3.2 shows the workflow of implementing LUTs in numerical computing software packages.

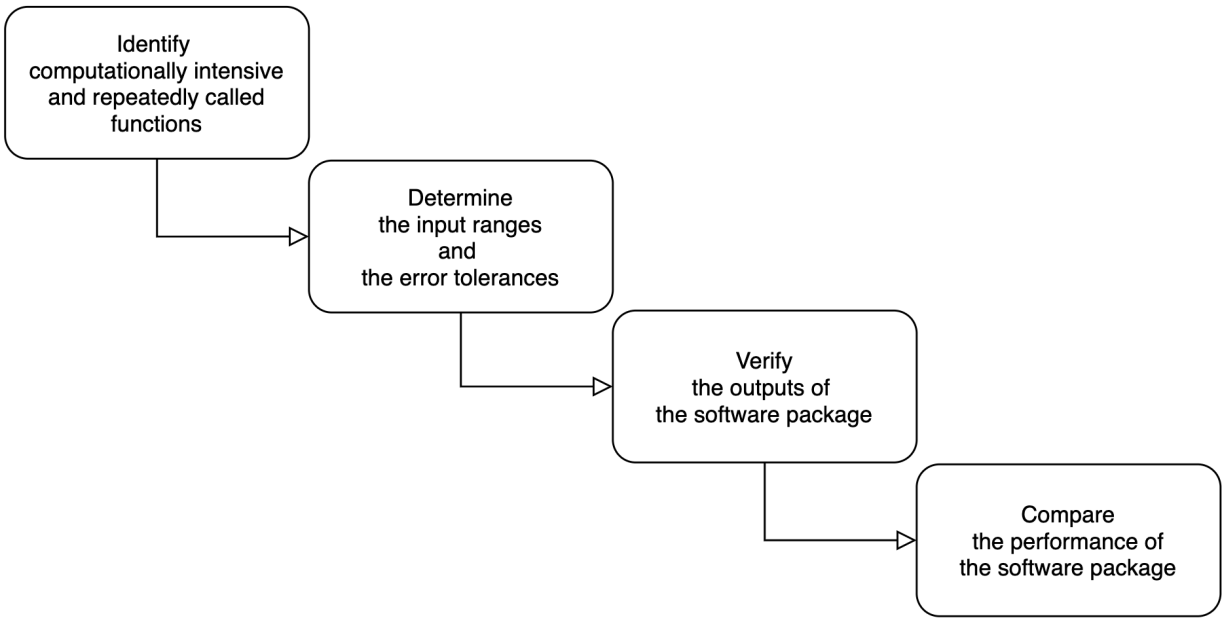

Figure 3.2: The workflow of implementing LUTs in numerical computing software packages.

In this chapter, we first introduced the applications of LUTs in different areas. Then, we described two open-source software packages, `FunC` and `CHM`, that are the foundation for the study performed. Finally, we proposed a systematic procedure of implementing LUTs that can be easily applied to numerical computing software packages.

4 Numerical Results

This chapter presents details of LUT implementations in `CHM` and evaluates the performance improvement introduced by LUT implementations of running `CHM` on cold-region hydrological simulations. In Section 4.1, we describe the procedure of profiling `CHM` and implementing LUTs for computationally intensive and repeatedly called functions in `CHM`. In Section 4.2, we verify the outputs of `CHM` with LUT implementations on a small-extent, cold-region hydrological simulation and show that LUT implementations improve the performance of `CHM`. In Section 4.3, we get results similar to that of Section 4.2 when using a large-extent, cold-region hydrological simulation. All the running times in Section 4.2 and Section 4.3 are measured using an `Intel`© `Core`™ `i7-6700 Processor @ 3.40 GHz` computer with `64 GB DDR4 RAM @ 2133 MHz`. The `Intel`© `Core`™ `i7-6700` has a 32 KB L1 data cache, a 32 KB L1 instruction cache, a 256 KB L2 cache, and a 8192 KB L3 cache.

4.1 LUT implementations in CHM

The *VTune Amplifier*, created by Intel, enables software developers to analyze their programs and detect time-consuming or resource-consuming code sections (*Intel*, 2019). We use the VTune Amplifier to profile the performance of running `CHM` on a problem called the Kananaskis snowpack simulation from September 01, 2017 to August 30, 2018. The simulation uses various meteorology inputs to drive a snowcover module, `snobal` (*Marks et al.*, 1999). The modules and their dependencies of the simulation are detailed in Figure 3.1. Directed lines are module dependencies. The starting points of directed lines are modules that provide variables corresponding to directed lines, and the end points of directed lines are modules that use variables corresponding to directed lines. Variables provided by one module but not used by others are not shown in Figure 3.1.

The domain covered by the simulation has an area of around 1000 km^2 and is shown in blue and yellow (blue = low elevation, yellow = high elevation) in Figure 4.1. The elevation of this domain ranges from 1827 m to 3053 m. The Kananaskis snowpack simulation divides the domain into 93,162 triangles and outputs hourly values of variables provided by all modules for all triangles. We selectively save values of *snow water equivalent* (SWE), *snow depth perpendicular to triangle slope* (snowdepthavg), and *snow depth perpendicular to triangle slope* [*cosine corrected*] (snowdepthavg_vert) into `vtu` files every 24 hours. All three variables are provided by the module `snobal`. We choose this simulation because it spends a lot of running time on the `snobal` module of `CHM` and, in this thesis, we only use LUTs to optimize the `snobal` module.

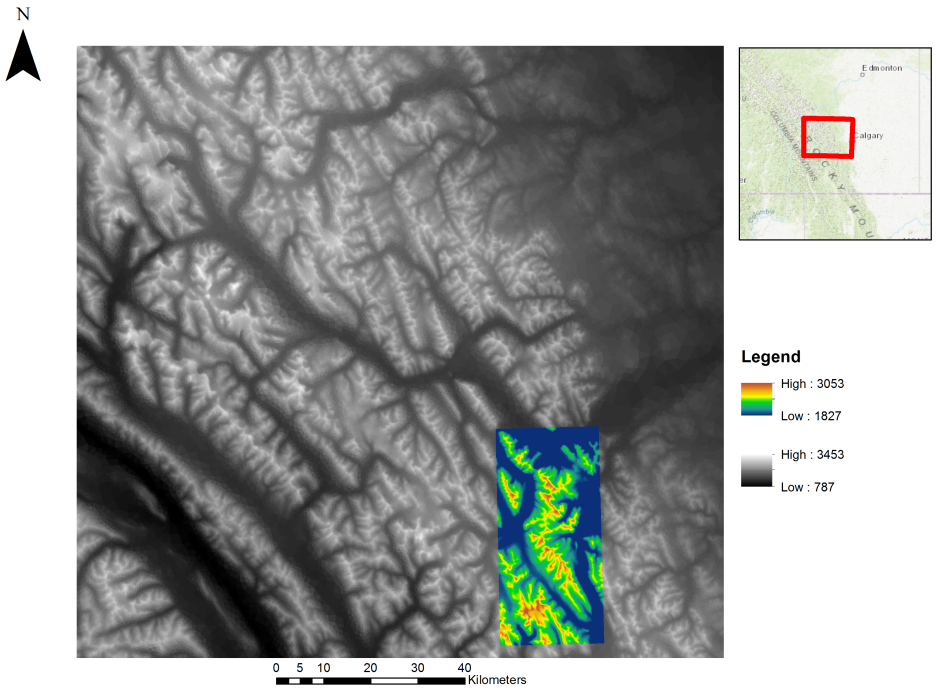

Figure 4.1: The domain covered by the Kananaskis snowpack simulation has an area of around 1000 km^2 and is shown in blue and yellow (blue = low elevation, yellow = high elevation). The elevation of this domain ranges from 1827 m to 3053 m. The domain covered by the SnowCast simulation has an area of around 17,880 km^2 and is shown in black and white (black = low elevation, white = high elevation). The elevation of this domain ranges from 787 m to 3453 m.

It takes the VTune Amplifier 203,540.832 s (around 2.4 days) of CPU time to run `CHM` on the Kananaskis snowpack simulation. Figure 4.2 shows partial VTune Amplifier profiling results of running `CHM` on the simulation. The blue highlights are performance issues that can potentially be improved by implementing LUTs. We notice that there are two functions, `sno::satw` and `sno::sati`, where the letters "w" and "i" stand for water and ice, in the `snobal` module that take up most of the computational resources. Most arithmetic operations of `sno::satw` and `sno::sati` happen in

$$
\begin{aligned}
f(x) =& 10^{Z_{satw}}, \\
Z_{satw} =& -7.90298\left(\frac{373.15}{x}-1\right) \\
&+\frac{5.02808}{\ln 10}\ln\frac{373.15}{x} \\
&-0.00000013816\left(10^{11.344\left(1-\frac{x}{373.15}\right)}-1\right) \\
&+0.0081328\left(10^{-3.49149\left(\frac{373.15}{x}-1\right)}-1\right) \\
&+\frac{\ln 101324.6}{\ln 10},
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
f(x) =& 100 \times 10^{Z_{sati}}, \\
Z_{sati} =& -9.09718 \left(\frac{273.16}{x} - 1 \right) \\
& - \frac{3.56654}{\ln 10} \ln \frac{273.16}{x} \\
& + 0.876793 \left(1 - \frac{x}{273.16} \right) \\
& + \frac{\ln 6.1071}{\ln 10},
\end{aligned}
\tag{4.2}
$$

respectively, where the input x is a temperature in the unit of Kelvin (K). In practice, `CHM` computes and stores $\ln 10$ in the beginning of `sno::satw` and `sno:sati` to reduce computational effort. Although Equation (4.1) is just a part of `sno::satw`, it accounts for most FLOPs of `sno::satw`. We regard `sno::satw` and Equation (4.1) as the same thing in this thesis and refer to Equation (4.1) as `sno::satw`. The same is true for `sno:sati` and Equation (4.2). The VTune Amplifier divides effective times by utilization into five categories: idle, poor, ok, ideal, and over. We focus on poor effective times, which are shown in red horizontal bars in Figure 4.2. The poor effective times for `sno::satw` and `sno::sati` themselves are 323.443 s and 1049.775 s, respectively. From Figure 4.2, we can also observe that functions `log` and `pow` have a large amount of poor effective times. Among those poor effective times, `sno::satw` and `sno::sati` make up the majority. More specifically, the poor effective time for `log` is 3376.663 s, the poor effective time for `sno::satw` calling `log` is 559.764 s, and the poor effective time for `sno::sati` calling `log` is 2019.602 s. The poor effective time for both `sno::satw` and `sno::sati` calling `log` accounts for more than 75% of the poor effective time for `log`. The poor effective for `pow` is 1817.079 s, the poor effective time for `sno::satw` calling `pow` is 603.607 s, and the poor effective for `sno::sati` calling `pow` is 668.086 s. The poor effective time for both `sno::satw` and `sno::sati` calling `pow` accounts for around 70% of the poor effective time for `pow`. All these poor effective times indicate that we can improve the performance of `sno::satw` and `sno::sati` by implementing LUTs for them.

Equation (4.1) and Equation (4.2) show that `sno::satw` calls the `log` function three times (`sno::satw` only calls the `log` function once for two $\ln 10$ evaluations because it computes and stores $\ln 10$ at the beginning of `sno::satw`) and calls `pow` three times per evaluation, and that `sno::sati` calls the `log` function three times (`sno::sati` only calls the `log` function once for two $\ln 10$ evaluations because it computes and stores $\ln 10$ in the beginning of `sno::sati`) and calls `pow` one time per evaluation. From our previous discussion, we know that the poor effective time for `sno::sati` calling `log` is around four times as long as the poor effective time for `sno::satw` calling `log` and that the poor effective time for `sno::sati` calling `pow` is almost the same as the poor effective time for `sno::satw` calling `pow`. Therefore, we can safely infer that the number of times that `sno::sati` is called is three to four times as many as the number of times that `sno::satw` is called.

Function / Call Stack	Effective Time by Utilization				
	Idle	Poor ▼	Ok	Ideal	Over
__ieee754_log_avx	0s	3376.663s	4137.582s	10513.150s	0s
sno::sati	0s	2019.602s	2118.330s	6669.144s	0s
sno::satw	0s	559.764s	585.640s	1823.525s	0s
__pow	0s	1817.079s	2094.529s	7329.552s	0s
sno::sati	0s	668.086s	758.580s	2415.179s	0s
sno::satw	0s	603.607s	652.727s	2038.240s	0s
sno::sati	0s	1049.775s	1102.777s	3464.844s	0s
___vsnprintf_chk	0s	705.260s	1273.428s	1532.544s	0s
face<CGAL::Projection_traits_	0s	616.971s	1825.269s	4711.998s	0s
face<CGAL::Projection_traits_	0.243s	592.584s	96.595s	10.221s	0s
std::__cxx11::basic_istringstre	0s	542.851s	1124.005s	228.928s	0s
boomphf::mphf<unsigned long	0.153s	489.011s	384.429s	248.596s	0s
__GI___pthread_rwlock_unloc	0s	461.047s	2545.988s	20281.100s	0s
sno::hle1	0s	448.850s	483.811s	1523.561s	0s
operator new	0s	344.711s	583.711s	1520.126s	0s
sno::satw	0s	323.443s	340.332s	1056.228s	0s

Figure 4.2: Partial VTune Amplifier profiling results of running `CHM` on the Kananaskis snowpack simulation. The blue highlights are performance issues related to `CHM` and can potentially be improved by implementing LUTs. We notice that there are two functions, `sno::satw` and `sno::sati` in the `snobal` module that perform poorly. There is a large amount of poor effective time for `sno::satw` and `sno::sati` themselves and for `sno::satw` and `sno::sati` calling `log` and `pow`.

In theory, we can implement LUTs for all functions listed in the VTune Amplifier profiling results, but in practice it may not be worth the effort to do so. The reasons why we only implement LUTs for `sno::satw` and `sno::sati` include but are not limited to: (1) Other functions have little poor effective time. (2) Other functions do not perform many arithmetical computations. (3) Other functions are in external libraries. If we want to use LUTs to optimize external libraries, we should treat them as independent software packages and apply the same procedure as here to them.

Let us take a closer look at the curves of `sno::satw` and `sno::sati`. The left subfigure of Figure 4.3 shows the curve of `sno::satw` from $x = 0.01$ to $x = 300$. From Equation (4.1) and the left subfigure of Figure 4.3, we conclude that `sno::satw` has the following properties:

1. the domain of `sno::satw` is $x > 0$,
2. `sno::satw` is continuous and has all orders of derivatives on its domain,
3. `sno::satw` is a strictly increasing function,
4. all values of `sno::satw` are positive,
5. all values of `sno::satw` for $x \leq 150$ are near zero. Specifically, `sno::satw`$(x) < 2.46 \times 10^{-6}$, $\forall x \in (0, 150]$.

The right subfigure of Figure 4.3 shows the curve of `sno::sati` from $x = 0.01$ to $x = 300$. From Equation (4.2) and the right subfigure of Figure 4.3, we conclude that `sno::sati` has all the same properties of `sno::satw`.

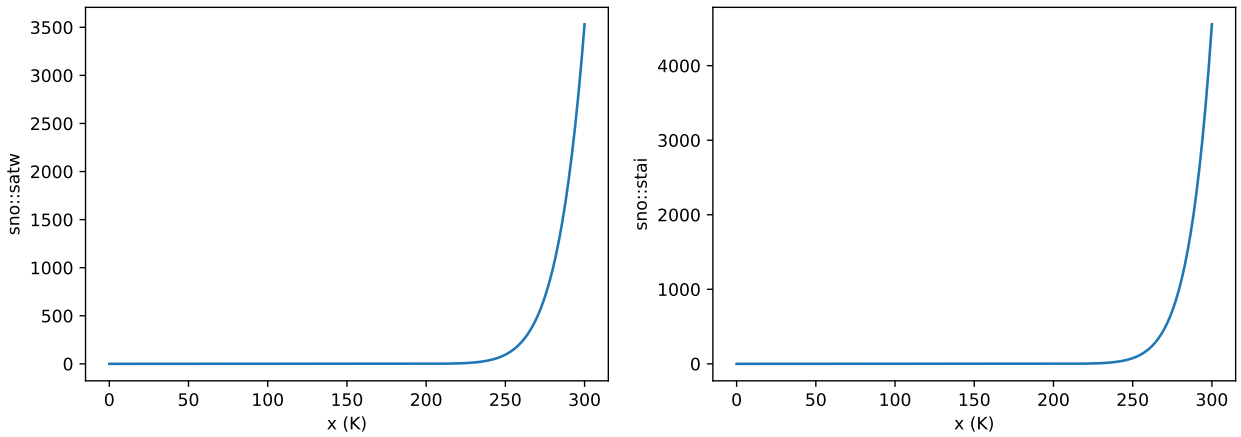

Figure 4.3: The curves of functions `sno::satw` and `sno::sati`.

Because both `sno::satw` and `sno::sati` are continuous and have all orders of derivatives on $x > 0$, we can implement all types of LUTs discussed in Chapter 2 for them. In order to do that, we need to determine the error tolerances and domains for the LUT implementations of `sno::satw` and `sno::sati`, respectively. We somewhat arbitrarily choose the single-precision tolerance 10^{-8} for both `sno::satw` and `sno::sati` LUT implementations. This tolerance is widely used and works perfectly in the sense of the difference of simulation outputs introduced by LUT implementations, as described below. Choosing domains is more complicated. Domains that are too small may not include x values we want to evaluate, and domains that too large may waste cache and memory and degrade the performance of LUTs. At first, we use the domain $[273.16, 323.16]$ for `sno::satw` because the letter "w" in `sno::satw` stands for water. Temperature 273.16 K (0°C) is the freezing point of water, and temperature 323.16 K (50°C) is higher than the highest temperature ever recorded in Canada (*Wikipedia contributors*, 2019). However, we find `CHM` calling `sno::satw` with temperatures much lower than the freezing point during internal iterations. By using trial and error in combination of a binary search strategy on the lower bound of the domain, we find $[223.16, 323.16]$ is a proper domain for `sno::satw`.

A similar problem also happens with `sno::sati`. We find `CHM` calling `sno::sati` with temperatures much higher than the freezing point during internal iterations. In order to solve this problem, we set the maximum of the domain for LUT implementations of `sno::sati` to 323.16 K. We also find `CHM` calling `sno::sati` with extremely low temperatures, like 45.50 K, during internal iterations. In order to solve this problem, we have to set the minimum of the domain for LUT implementations of `sno::sati` to a temperature near 0 K. This creates a problem that we have mentioned in Section 3.2. Error measure (3.1) approximately equals to 2 when $f(x)$ is almost zero. This is the case for `sno::sati` when x is close to 0. We try to create a LUT for `sno::sati` with the domain $[0.01, 323.16]$ and the error tolerance 10^{-8}. It takes `FunC` prohibitively long time to do so. So we decide to use an if-else branch to solve this problem, as follows.

Because the machine epsilon $\varepsilon_{\text{mach}}$ of double-precision arithmetic is 2.22×10^{-16} (*Higham*, 2002) and `sno::sati`$(90) = 1.38 \times 10^{-17} \lesssim \varepsilon_{\text{mach}}$, we use $[90, 323.16]$ as the domain for `sno::sati`. When `sno::satw`

is called with a temperature lower than 90 K, we return 0 directly. Although there is a performance penalty for using if-else branches, this strategy makes LUT implementations for `sno::sati` possible.

Next, we need to decide which type of LUT should be used in `CHM`. We ignore the class `UniformConstantTaylorTable` in the following discussion because it uses too much space and is inefficient in practice. Because the Kananaskis snowpack simulation uses more than 20 GB RAM, which is much larger than the cache size of the `Intel`© `Core`™ `i7-6700`, LUT implementations are highly likely to stay out of cache all the time. There is no other application programs running in the computer I use when we run the simulation. Because the computer has 64 GB RAM and there are more than 30 GB available RAM when we run the simulation, we can safely assume that all LUT implementations are always in the main memory. This case fits the description of the worst case for LUT implementations in (*Green et al.*, 2019). According to the rule of thumb Green et al. propose for the worst case, we should use the LUT implementation with the fewest FLOPs. The class `UniformConstantTaylorTable` uses the fewest FLOPs per evaluation. However, it is almost unusable in practice because it uses much more space than other LUT implementations and takes prohibitively long to generate the desired LUTs. The class `UniformLinearPrecomputedInterpolationTable` uses the second fewest FLOPs per evaluation. Compared to the class `UniformLinearInterpolationTable`, it uses one fewer FLOP but one more IOP per evaluation. Because an IOP is faster than a FLOP in theory, we should expect that the class `UniformLinearPrecomputedInterpolationTable` performs slightly better than the class `UniformLinearInterpolationTable`. However, the class `UniformLinearPrecomputedInterpolationTable` uses twice as much space as the class `UniformLinearInterpolationTable`. Taking all the above factors into consideration, we use the class `UniformLinearInterpolationTable` in our experiments.

For simplicity, we refer to the original `CHM` as `CHM` without LUT and refer to `CHM` with LUT implementations for `sno::satw` and `sno::sati` as `CHM` with LUT.

4.2 Kananaskis Snowpack Simulation

First, we use `vtu` files that store daily values of SWE, snowdepthavg, and snowdepthavg_vert to check whether there is a difference between outputs of `CHM` without LUT and `CHM` with LUT. Figure 4.4 visualizes all daily outputs and *root mean square errors* (RMSEs) of `CHM` without LUT and `CHM` with LUT on the Kananaskis snowpack simulation. Blue lines show the daily maximum values of all variables outputted by `CHM` without LUT across all the triangles. Yellow dash-dot lines show the daily maximum values of all variables outputted by `CHM` with LUT across all the triangles. The difference between outputs of two versions of `CHM` is visually indistinguishable. Daily RMSEs of all variables between outputs of two versions of `CHM` shown in green dotted lines also support this observation. The maximum RMSE of SWE is less than 0.06 mm, the maximum RMSE of snowdepthavg is less than 0.0005 cm, and the maximum RMSE of snowdepthavg_vert is less than 0.0008 cm. All the three maximum RMSEs are much less than the values of their corresponding variables.

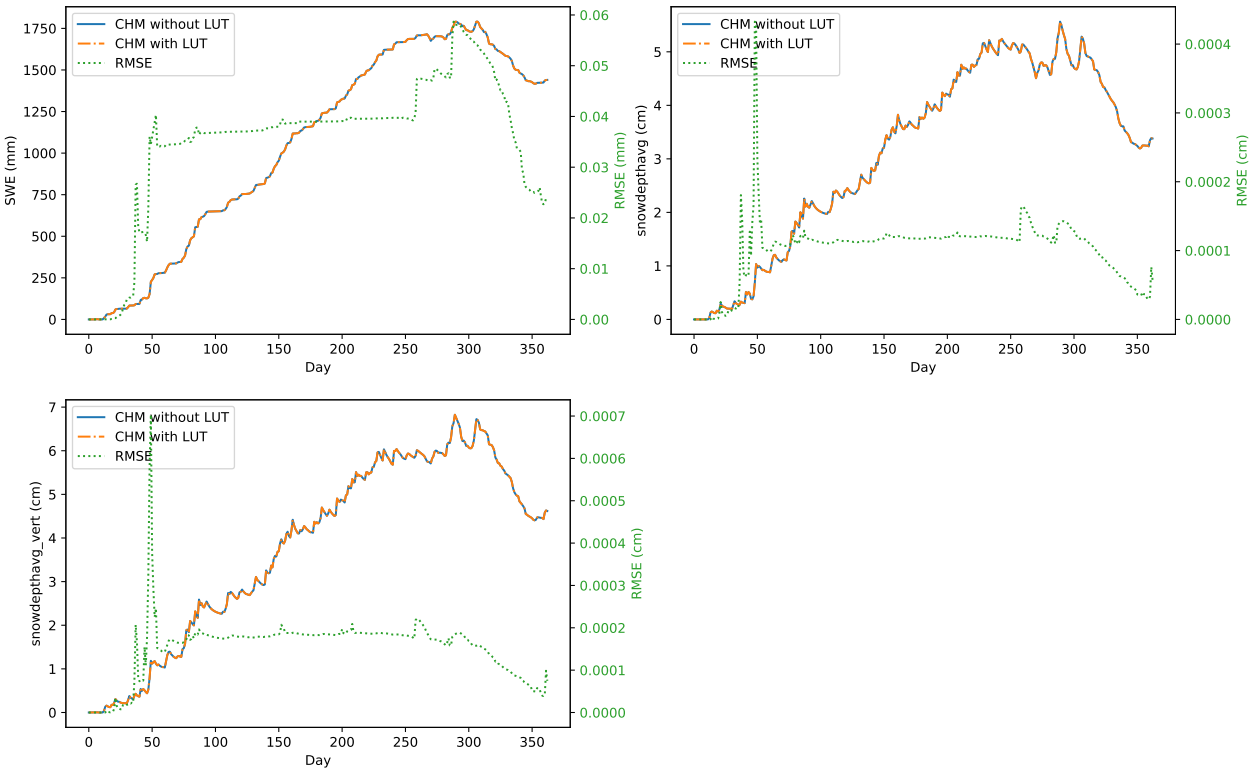

Figure 4.4: Visualizations of daily outputs and RMSEs of both `CHM` without LUT and `CHM` with LUT on the Kananaskis snowpack simulation. Blue lines show the daily maximum values of all variables outputted by `CHM` without LUT across all the triangles. Yellow dash-dot lines show the daily maximum values of all variables outputted by `CHM` with LUT across all the triangles. The difference between outputs of `CHM` without LUT and `CHM` with LUT is visually indistinguishable. Green dotted lines show daily RMSEs of all variables between outputs of two versions of `CHM` without LUT and `CHM` with LUT. All the three maximum RMSEs are much less than the values of their corresponding variables.

We use both `CHM` without LUT and `CHM` with LUT to run the Kananaskis snowpack simulation 90 times, respectively. All the running times are measured in seconds as reported by the debug mode of `CHM`. Table 4.1 provides summary statistics of the Kananaskis snowpack simulation running times for each `CHM` version. The sample mean of the running times of `CHM` without LUT on the Kananaskis snowpack simulation is 8004.34 s (around 2.2 h), and the sample mean of the running times of `CHM` with LUT on the Kananaskis snowpack simulation is 6764.79 s (around 1.9 h). The difference between two sample means is $8004.34 - 6764.79 = 1239.55$ s.

Version	Mean (s)	Standard deviation (s)	Minimum (s)	Maximum (s)
`CHM` without LUT	8004.34	44.24	7965	8157
`CHM` with LUT	6764.79	46.62	6728	6930

Table 4.1: Summary statistics of the Kananaskis snowpack simulation running times for each `CHM` version.

The difference may be due to LUT implementations for `sno::satw` and `sno::sati` or due to the natural variation in running times. Formally, we can set two hypotheses to evaluate the cause of this difference:

> H_0: On average, the running times of `CHM` with LUT and `CHM` without LUT on the Kananaskis snowpack simulation are equal.
>
> H_A: On average, the running times of `CHM` with LUT and `CHM` without LUT on the Kananaskis snowpack simulation are *not* equal.

H_0 is called the *null hypothesis*. It represents the case of no difference between the average running times of two versions of `CHM` on the Kananaskis snowpack simulation. H_A is called the *alternative hypothesis*. It represents the case that there is a difference between the average running times of two versions of `CHM` on the Kananaskis snowpack simulation. According to (*Diez et al.*, 2012), we need to verify two conditions, the *independence* requirement and the *normality* requirement of the data, to use a t-distribution to perform an independent samples t-test. The independence requirement of running times is satisfied because we run one experiment at a time. Figure 4.5 shows the running time histograms of the two versions of `CHM` on the Kananaskis snowpack simulation. Because there are no particularly extreme outliers in either histogram (all the running times are within 3.54 standard deviations of the corresponding mean), the normality requirement of running times is also satisfied (*Diez et al.*, 2012).

By using the Python library `SciPy`, we calculate that p-value is 4.11×10^{-204}, which is far less than the standard significance level 0.05. Therefore, we can safely reject the null hypothesis H_0 in favor of H_A. The data provide strong evidence that the average running time of `CHM` with LUT is less than the average running time of `CHM` without LUT on the Kananaskis snowpack simulation. The corresponding 95% confidence interval of the difference is $(1226.18, 1252.93)$. We note that

$$\frac{1226.18}{6764.79} = 18.12\%, \ \frac{1252.93}{6764.79} = 18.52\%.$$

So in the sense of the average running time, we are 95% confident that implementing linear interpolation LUTs for `sno::satw` and `sno:sati` improves the performance of running `CHM` on the Kananaskis snowpack simulation by 18.12% to 18.52%.

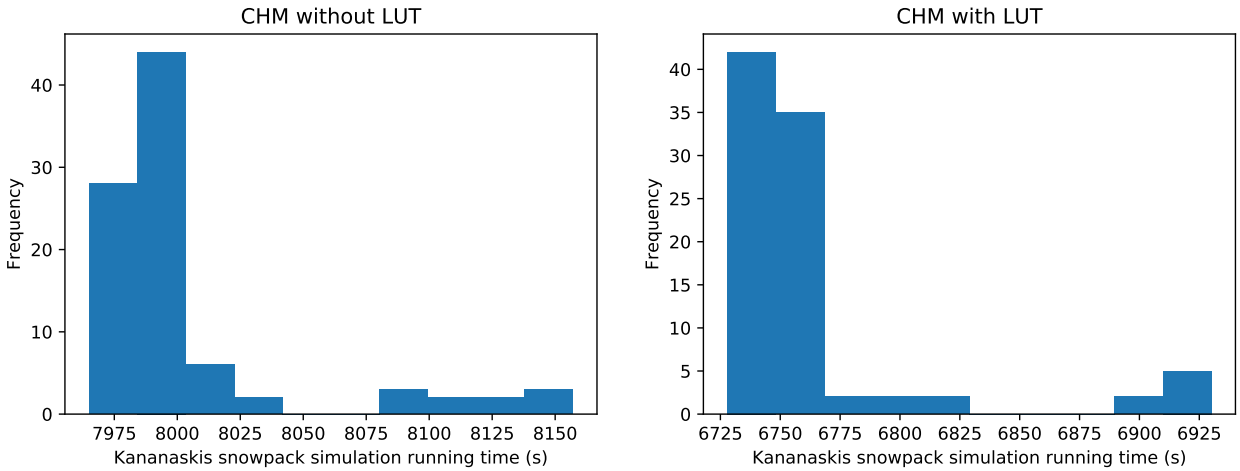

Figure 4.5: The running time histograms of CHM without LUT and CHM with LUT on the Kananaskis snowpack simulation.

Next, we check whether there is a significant difference between the minimum running times of the two versions of CHM on the Kananaskis snowpack simulation. We randomly divide the 90 running times of CHM without LUT on the Kananaskis snowpack simulation into three equal groups. Each group contains 30 running times and is a good estimation of the true running time distribution of CHM without LUT on the Kananaskis snowpack simulation (*Diez et al.*, 2012). The minimum running times of the three groups are 7965 s, 7965 s, and 7968 s. Each group minimum running time is an estimation of the minimum running time of CHM without LUT on the Kananaskis snowpack simulation. According to the Central Limit Theorem; see for example (*Diez et al.*, 2012), the mean of the three group minimum running times,

$$\frac{7965 + 7965 + 7968}{3} = 7966.00 \text{ s},$$

is a better estimate of the minimum running time of CHM without LUT on the Kananaskis snowpack simulation than each group minimum running time.

We apply the same operations to the running times of CHM with LUT on the Kananaskis snowpack simulation. The 90 running times of CHM with LUT on the Kananaskis snowpack simulation are randomly divided into three equal groups. The minimum running times of the three groups are 6728 s, 6729 s, and 6738 s. Each group minimum running time is an estimation of the minimum running time of CHM with LUT on the Kananaskis snowpack simulation. The mean of the three group minimum running times,

$$\frac{6728 + 6729 + 6738}{3} = 6731.67 \text{ s},$$

is a better estimate of the minimum running time of CHM with LUT on the Kananaskis snowpack simulation than each group minimum running time.

The difference between the two means of the minimum running times is $7966.00 - 6731.67 = 1234.33$ s. Similarly, we can create two hypotheses about the difference between the minimum running times:

H_0: The minimum running times of `CHM` with LUT and `CHM` without LUT on the Kananaskis snowpack simulation are equal.

H_A: The minimum running times of `CHM` with LUT and `CHM` without LUT on the Kananaskis snowpack simulation are *not* equal.

Because each group minimum running time is independent of the others (the data independence requirement) and all the group minimum running times of the same `CHM` version are close together (the data normality requirement), we use a t-distribution to perform an independent samples t-test. The p-value here is 9.50×10^{-7}, which is far less than the standard significance level 0.05. Therefore, we can safely reject the null hypothesis H_0 in favor of H_A. The data provide strong evidence that the minimum running time of `CHM` with LUT is less than the minimum running time of `CHM` without LUT on the Kananaskis snowpack simulation. The corresponding 95% confidence interval of the difference is $(1222.02, 1246.64)$. We note that

$$\frac{1222.02}{6731.67} = 18.15\%, \ \frac{1246.64}{6731.67} = 18.52\%.$$

So in the sense of the minimum running time, we are 95% confident that implementing linear interpolation LUTs for `sno::satw` and `sno:sati` improves the performance of running `CHM` on the Kananaskis snowpack simulation by 18.15% to 18.52%.

We often need to run a simulation hundreds of times to develop a simulation model and tune its parameters. If we run the Kananaskis snowpack simulation 100 times, we can save around 1.5 days of computation time just by using `CHM` with LUT. Also, generating LUTs for `sno::satw` and `sno::sati` in `CHM` with LUT takes less than half a minute, which is negligible compared to the reduced running time.

4.3 SnowCast Simulation

Next, we consider a similar simulation called the *SnowCast simulation* on a larger domain. The SnowCast simulation has the same start date and end date as the Kananaskis snowpack simulation but uses slightly different meteorology inputs to drive the snowcover module `snobal`. The modules and their dependencies of the simulation are detailed in Figure 4.6. The domain covered by the SnowCast simulation has an area of around 17,880 km^2 and is shown in black and white (black = low elevation, white = high elevation) in Figure 4.1. The elevation of this domain ranges from 787 m to 3453 m. The SnowCast simulation divides the domain into 238,790 triangles and outputs hourly values of variables provided by all modules for all triangles. As before, we selectively save daily values of SWE, snowdepthavg, and snowdepthavg_vert into `vtu` files.

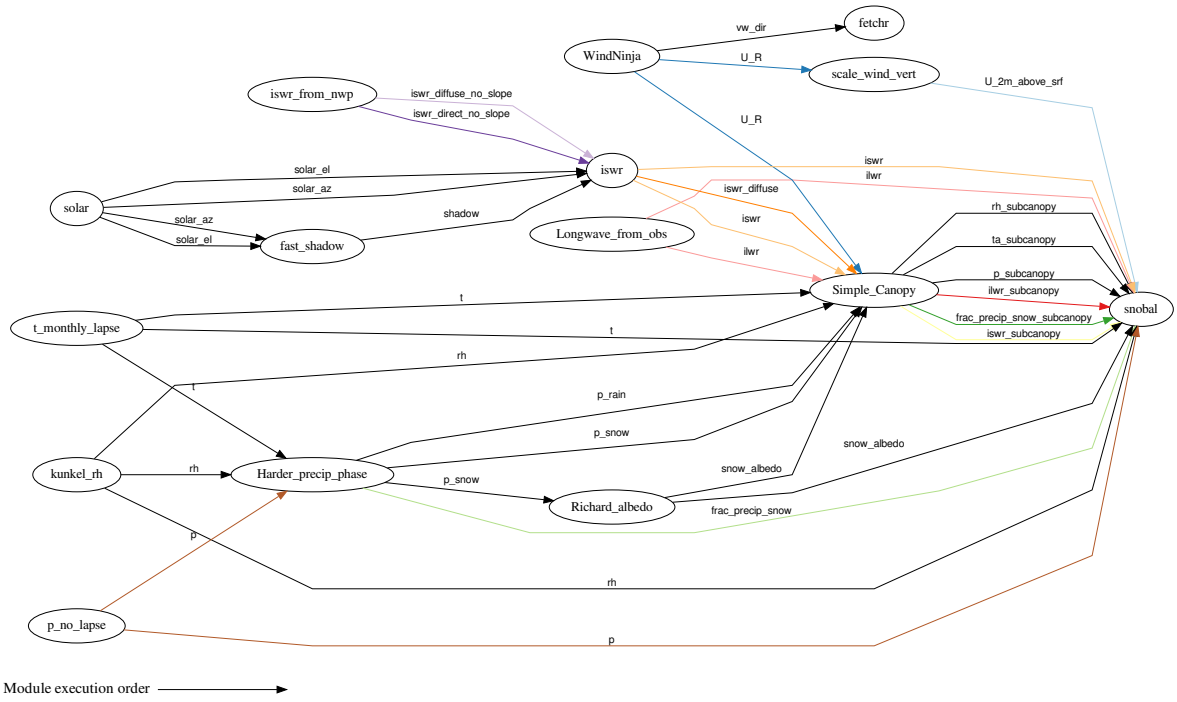

Figure 4.6: The modules and their dependencies of the SnowCast simulation.

Because the Kananaskis snowpack simulation and the SnowCast simulation are very similar, we expect that we can still implement LUTs for `sno::satw` and `sno::sati` to improve the performance of `CHM` on the SnowCast simulation. The error tolerances and domains for LUT implementations of `sno::satw` and `sno::sati` remain the same. Specifically, the error tolerances for both LUT implementations of `sno::satw` and `sno::sati` are both set to 10^{-8}, and the domains are $[223.16, 323.16]$ and $[90, 323.16]$, respectively. When `sno::satw` is called with a temperature lower than 90 K, we return 0 directly. The validity of the error tolerances and domains is proved by the following experiments. Also, the SnowCast simulation uses around 30 GB RAM, which is much larger than the cache size of the `Intel`© `Core`™ `i7-6700`. Accordingly, LUT implementations are likely to stay out of cache all the time. When we run the simulation, there is no other application programs running in my computer. Because my computer has 64 GB RAM and there are around 30 GB available RAM when we run the simulation, we can safely assume that all LUT implementations are always in the main memory. For the same reason of using the class `UniformLinearInterpolationTable` in the Kananaskis snowpack simulation, we continue using the class `UniformLinearInterpolationTable` for `sno::satw` and `sno::sati`.

First, we use `vtu` files that store daily values of SWE, snowdepthavg, and snowdepthavg_vert to check whether there is a difference between outputs of `CHM` without LUT and `CHM` with LUT. Figure 4.7 visualizes all daily outputs and RMSEs of the SnowCast simulation. Blue lines show the daily maximum values of

all variables outputted by CHM without LUT across all the triangles. Yellow dash-dot lines show the daily maximum values of all variables outputted by CHM with LUT across all the triangles. The difference between outputs of two versions of CHM is visually indistinguishable. Daily RMSEs of all variables between outputs of two versions of CHM shown in green dotted lines also support this observation. The maximum RMSE of SWE is less than 0.14 mm, the maximum RMSE of snowdepthavg is less than 0.0014 cm, and the maximum RMSE of snowdepthavg_vert is less than 0.0016 cm. All the three maximum RMSEs are much less than the values of their corresponding variables.

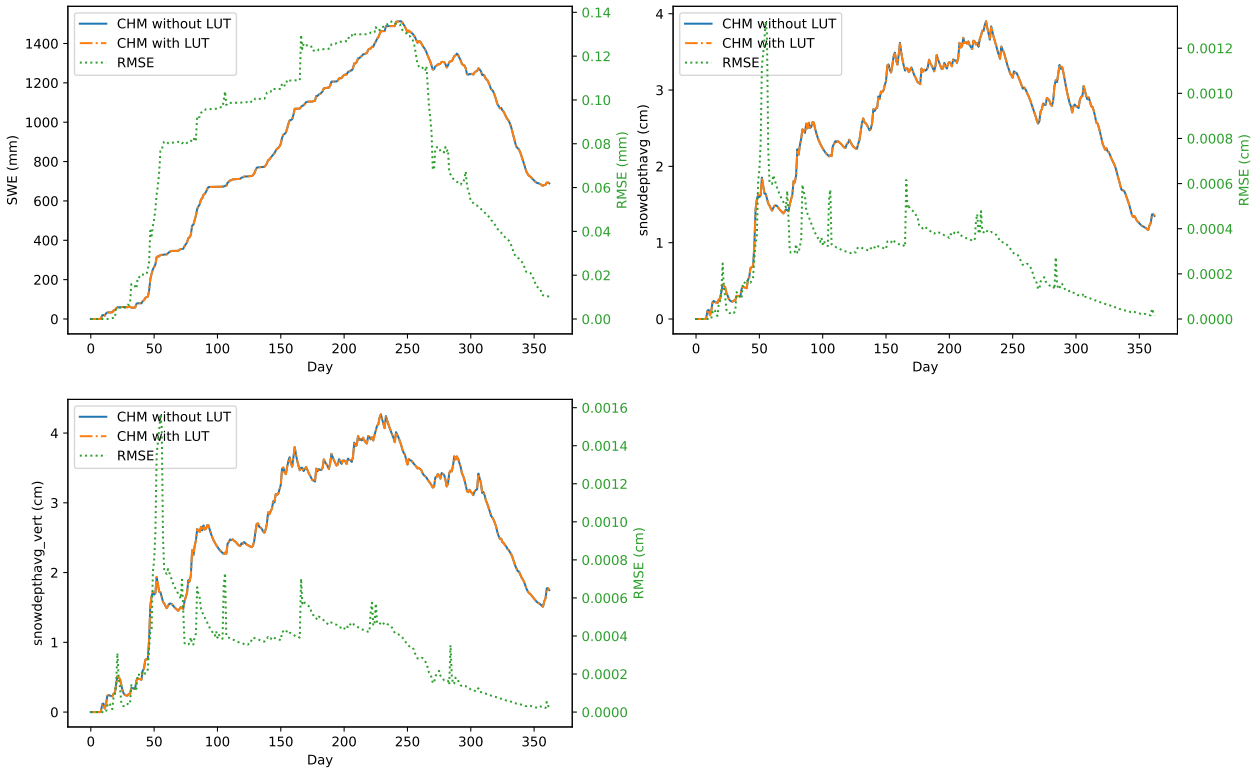

Figure 4.7: Visualizations of daily outputs and RMSEs of both CHM without LUT and CHM with LUT on the SnowCast simulation. Blue lines show the daily maximum values of all variables outputted by CHM without LUT across all the triangles. Yellow dash-dot lines show the daily maximum values of all variables outputted by CHM with LUT across all the triangles. The difference between outputs of CHM without LUT and CHM with LUT is visually indistinguishable. Green dotted lines show daily RMSEs of all variables between outputs of CHM without LUT and CHM with LUT. All the three maximum RMSEs are much less than the values of their corresponding variables.

We use both CHM without LUT and CHM with LUT to run the SnowCast simulation 90 times, respectively. All the running times are measured in seconds by the debug mode of CHM. Table 4.2 provides summary statistics of the SnowCast simulation running times for each CHM version. CHM without LUT takes 22,303.24 s (around 6.2 h) on average to run the SnowCast simulation. CHM with LUT takes 18,417.44 s (around 5.1 h)

on average to run the SnowCast simulation. The difference between two average running times is 22,303.24 − 18,417.44 = 3885.80 s.

Version	Mean (s)	Standard deviation (s)	Minimum (s)	Maximum (s)
`CHM` without LUT	22,303.24	64.87	22,203	22,481
`CHM` with LUT	18,417.44	70.55	18,318	18,665

Table 4.2: Summary statistics of the SnowCast simulation running times for each `CHM` version.

We can set two hypotheses to evaluate the cause of this difference:

H_0: On average, the running times of `CHM` with LUT and `CHM` without LUT on the SnowCast simulation are equal.

H_A: On average, the running times of `CHM` with LUT and `CHM` without LUT on the SnowCast simulation are *not* equal.

The independence requirement between each running time holds because we run one experiment at a time. Figure 4.8 shows the running time histograms of two versions of `CHM` on theSnowCast simulation. Because there are no particularly extreme outliers in both histograms (all the running times are within 3.51 standard deviations of the corresponding mean), the normality requirement of running times is satisfied (*Diez et al.*, 2012). So a t-distribution is still suitable for this inference.

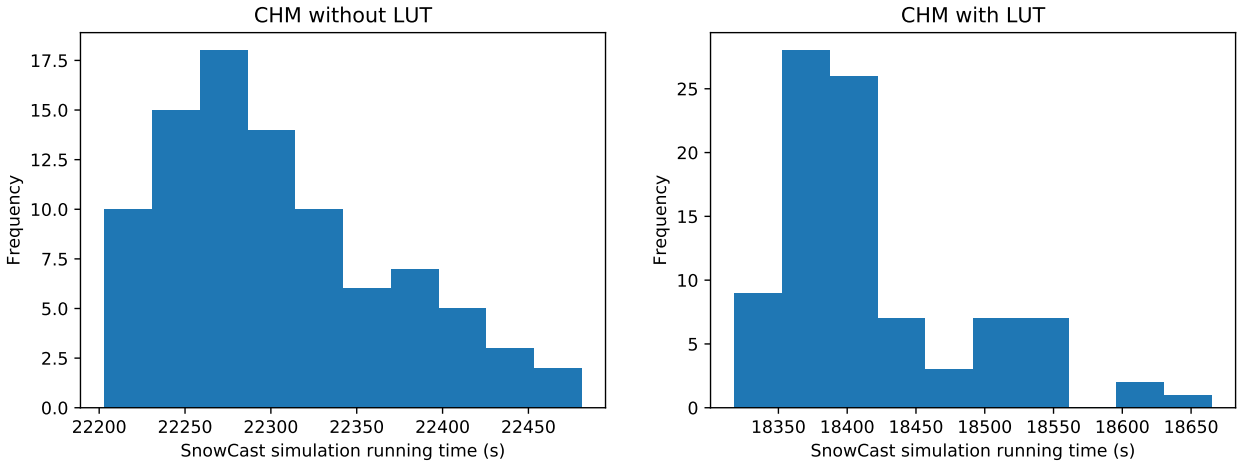

Figure 4.8: The running time histograms of `CHM` without LUT and `CHM` with LUT on the SnowCast simulation.

From this independent samples t-test, the p-value is equal to 2.65×10^{-260}, which is far less than the standard significance level 0.05. Therefore, we can safely reject H_0 in favor of H_A. The data provide strong evidence that the running times of `CHM` with LUT are less than the running times of `CHM` without LUT on the SnowCast simulation. The corresponding 95% confidence interval of the difference is $(3865.86, 3905.74)$.

We note that

$$\frac{3865.86}{18{,}417.44} = 20.99\%,\ \frac{3905.74}{18{,}417.44} = 21.21\%.$$

So in the sense of the average running time, we are 95% confident that implementing linear interpolation LUTs for `sno::satw` and `sno:sati` improves the performance of running `CHM` on the SnowCast simulation by 20.99% to 21.21%.

Next, we check whether there is a difference between the minimum running times of the two versions of `CHM` on the SnowCast simulation. The 90 running times of `CHM` without LUT on the SnowCast simulation are randomly divided into three equal groups. The minimum running times of the three groups are 22,204 s, 22,203 s, and 22,217 s. Each group minimum running time is an estimation of the minimum running time of `CHM` without LUT on the SnowCast simulation. The mean of the three group minimum running times,

$$\frac{22{,}204 + 22{,}203 + 22{,}217}{3} = 22{,}208.00 \text{ s},$$

is a better estimate of the minimum running time of `CHM` without LUT on the SnowCast simulation than each group minimum running time.

We apply the same operations to the running times of `CHM` with LUT on the SnowCast simulation. The 90 running times of `CHM` with LUT on the SnowCast simulation are randomly divided into three equal groups. The minimum running times of the three groups are 18,321 s, 18,318 s, and 18,341 s. Each group minimum running time is an estimation of the minimum running time of `CHM` with LUT on the SnowCast simulation. The mean of the three group minimum running times,

$$\frac{18{,}321 + 18{,}318 + 18{,}341}{3} = 18{,}326.67 \text{ s},$$

is a better estimate of the minimum running time of `CHM` with LUT on the SnowCast simulation than each group minimum running time.

The difference between the two means of the group minimum running times is $22{,}208.00 - 18{,}326.67 = 3881.33$ s. We create two hypotheses about the cause this difference:

> H_0: The minimum running times of `CHM` with LUT and `CHM` without LUT on the SnowCast simulation are equal.
>
> H_A: The minimum running times of `CHM` with LUT and `CHM` without LUT on the SnowCast simulation are *not* equal.

Because each group minimum running time is independent of the others (the data independence requirement) and all the group minimum running times of the same `CHM` version are close together (the data normality requirement), we use a t-distribution to perform an independent samples t-test. The p-value in here is 3.71×10^{-9}, which is far less than the standard significance level 0.05. Therefore, we can safely reject the null hypothesis H_0 in favor of H_A. The data provide strong evidence that the minimum running time of `CHM` with LUT is less than the minimum running time of `CHM` without LUT on the SnowCast simulation. The corresponding 95% confidence interval of the difference is $(3855.80, 3906.87)$. We note that

$$\frac{3855.80}{18{,}326.67} = 21.04\%,\ \frac{3906.87}{18{,}326.67} = 21.32\%.$$

So in the sense of the minimum running time, we are 95% confident that implementing linear interpolation LUTs for `sno::satw` and `sno:sati` improves the performance of running `CHM` on the SnowCast simulation by 21.04% to 21.32%.

If we run the SnowCast simulation 100 times, we can save around 4.5 days of computation time just by using `CHM` with LUT. Also, generating LUTs for `sno::satw` and `sno::sati` in `CHM` with LUT takes less than half a minute, which is negligible compared to the reduced running time.

In this chapter, we first profiled the performance of running `CHM` without LUT on the Kananaskis snowpack simulation and identified two computationally intensive and repeatedly called functions `sno::satw` and `sno::sati`. Then, we generated piecewise linear interpolation LUTs with error tolerances 10^{-8} for both functions. Finally, we compared the outputs and the running times of `CHM` without LUT and `CHM` with LUT on both the Kananaskis snowpack simulation and the SnowCast simulation. The experiments show that both `CHM` versions produced the same outputs and `CHM` with LUT reduced the running times on both simulations by around 20%.

5 Conclusion and Future Work

Cold-region hydrological processes play an important role in the environment of cold regions. Simulations of them help people understand past hydrological events and predict future ones (*Freeze and Harlan*, 1969). Due to the need to use more complex models and simulations over larger domains, it is important to make cold-region hydrological simulations efficient so they can be run within a reasonable period. For this purpose, the open-source software package `CHM` uses unstructured triangular meshes to discretize domain surfaces and it also employs parallelization. We implement LUTs in `CHM` to further reduce running times of cold-region hydrological simulations.

Specifically, this thesis contributes to research on cold-region hydrological simulation in following aspects:

1. We analyze `CHM` and identify two computationally intensive and repeatedly called functions `sno::satw` and `sno::sati` in the `CHM` module `snobal` by using Intel VTune. Their expressions are given in Equation (4.1) and Equation (4.2). Then we scrutinize the curves of the two functions and identify key features, that are critical to implement LUTs for them.

2. We implement `UniformLinearInterpolationTable` for functions `sno::satw` and `sno::sati` by using the C++ library `FunC` and commit my implementations to the Git repository of `CHM`. We run the Kananaskis snowpack simulation and the SnowCast simulation on an `Intel`© `Core`™ `i7-6700 CPU @ 3.40 GHz` computer with `64 GB DDR4 RAM @ 2133 MHz`. Because both simulations use more than 20 GB of memory, which is much larger than the cache size of the `Intel`© `Core`™ `i7-6700`, LUTs stay in cache with a low probability. Accordingly, we decide to implement `UniformLinearInterpolationTable` for both `sno::satw` and `sno::sati`. By using trial and error in combination of a binary search strategy on the lower bounds of domains, we find $[223.16, 323.16]$ and $[0, 323.16]$ large enough to include all inputs of `sno::satw` and `sno::sati` respectively. Because the error measure (3.1) approximately equals to 2 when $f(x)$ is near zero, `FunC` cannot generate the correct LUT by error tolerance for `sno::sati`. We use an if-else branch to divide the domain of `sno::sati` into two sub-domains: $[0, 90]$ and $[90, 323.16]$. `FunC` only generates `UniformLinearInterpolationTable` for `sno::sati` with domain $[90, 323.16]$. `sno::sati` uses its `UniformLinearInterpolationTable` to evaluate function values when input $x \in [90, 323.16]$ and returns zero directly when $x \in [0, 90]$.

3. We improve the performance of `CHM` by around 20% on both the Kananaskis snowpack simulation and the SnowCast simulation by implementing LUTs for `sno::satw` and `sno::sati`. Specifically, we are 95% confident that, in the sense of the average running time, the performance improvement of `CHM`

on the Kananaskis snowpack simulation is between 18.12% and 18.52% and that, in the sense of the minimum running time, the performance improvement of `CHM` on the Kananaskis snowpack simulation is between 18.15% and 18.52%. We are 95% confident that, in the sense of the average running time, the performance improvement of `CHM` on the SnowCast simulation is between 20.99% and 21.21% and that, in the sense of the minimum running time, the performance improvement of `CHM` on the SnowCast simulation is between 21.04% and 21.32%. Also, all the maximum RMSEs of SWE, snowdepthavg, and snowdepthavg_vert are much less than the values of their corresponding variables, and the difference between outputs of the original `CHM` and the `CHM` with LUT implementations is visually negligible.

4. We provide a systematic procedure of implementing LUTs that can be easily applied to numerical computing software packages. Nowadays, many software packages use iterative methods for optimization or to find a solution of equations and may call some computationally intensive functions repeatedly. Other researchers can follow the procedure and use `FunC` to implement LUTs in their programs as an optimization.

Here are some possible directions that we can follow to extend the studies in this thesis further in future:

1. We wish to implement a different error measure in `FunC`. `FunC` has a problem with generating LUTs by error tolerance when function values are near zero. This is demonstrated in Section 3.2. A better error measure is

$$E = \max_x \frac{|f(x) - \tilde{f}(x)|}{1 + |f(x)|}.$$

   We get an approximation of the relative error of $\tilde{f}(x)$ when $|f(x)|$ is large enough, and we get an approximation of the absolute error of $\tilde{f}(x)$ when $|f(x)|$ is near zero. By using this error measure, `FunC` may generate LUTs by error tolerance for `sno:sati` with domain $[0, 323.16]$ successfully.

2. We wish to compare the performance of LUTs when they stay in cache with different probabilities. In this thesis, our LUT implementations are all likely to stay out of cache all the time. In such a case, the LUT with the fewest FLOPs per evaluation performs the best. In the future work, we want to compare the performance of different LUT types in the case where smaller LUTs with more FLOPs per evaluation stay in cache with a higher probability and larger LUTs with fewer FLOPs per evaluation stay in cache with a lower probability.

3. We wish to identify more functions in `CHM` that can be improved by LUT implementations. In this thesis, we only implement LUTs for two functions, `sno::satw` and `sno::sati` in `snobal`. They are computationally intensive and repeatedly called and take up many computational resources. Green et al. find that LUTs are faster than direct evaluation even for simple functions like the exponential function (*Green et al.*, 2019). We may find some less computationally intensive functions that are considered ideal by Intel VTune but can be improved by LUT implementations.

4. We wish to use `FunC` to implement LUTs in other software packages. The procedure of implementing LUTs in this thesis is systematic, and we can use the same procedure to implement LUTs for computationally intensive and repeatedly called functions in other software packages. We may be able to provide a significant improvement in the true minimum running time.

Bibliography

Amdahl, G. M. (2013), Computer architecture and amdahl's law, *Computer*, *46*(12), 38–46, doi:10.1109/MC.2013.418.

Battiato, S., and R. Lukac (2008), *Color-Mapped Imaging*, pp. 83–88, Springer US, Boston, MA, doi:10.1007/978-0-387-78414-4_11.

Buehler, S., P. Eriksson, T. Kuhn, A. Von Engeln, and C. Verdes (2005), ARTS, the atmospheric radiative transfer simulator, *Journal of Quantitative Spectroscopy and Radiative Transfer*, *91*(1), 65–93.

Buehler, S. A., P. Eriksson, and O. Lemke (2011), Absorption lookup tables in the radiative transfer model ARTS, *Journal of Quantitative Spectroscopy and Radiative Transfer*, *112*(10), 1559–1567.

Cooper, J., S. McKeever, and A. Garny (2006), On the Application of Partial Evaluation to the Optimisation of Cardiac Electrophysiological Simulations, in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pp. 12–20, ACM, New York, NY, USA, doi:10.1145/1111542.1111546.

Dias, M. A., D. O. Sales, and F. S. Osorio (2014), Automatic generation of luts for hardware neural networks, in *2014 Brazilian Conference on Intelligent Systems*, pp. 115–120, doi:10.1109/BRACIS.2014.31.

Diez, D. M., C. D. Barr, and M. Cetinkaya-Rundel (2012), *OpenIntro statistics*, OpenIntro.

Dozier, J., and J. Frew (1990), Rapid calculation of terrain parameters for radiation modeling from digital elevation data, *IEEE Transactions on Geoscience and Remote Sensing*, *28*(5), 963–969.

Duarte, C. M., T. M. Lenton, P. Wadhams, and P. Wassmann (2012), Abrupt climate change in the arctic, *Nature Climate Change*, *2*(2), 60.

Freeze, R., and R. Harlan (1969), Blueprint for a physically-based, digitally-simulated hydrologic response model, *Journal of Hydrology*, *9*(3), 237 – 258, doi:https://doi.org/10.1016/0022-1694(69)90020-1.

Frigo, M., and S. G. Johnson (2005), The design and implementation of FFTW3, *Proceedings of the IEEE*, *93*(2), 216–231.

Gonzales, R. C., and R. E. Woods (2002), *Digital image processing*, vol. 2, Prentice Hall New Jersey.

Green, K. R., T. A. Bohn, and R. J. Spiteri (2018), FunC source code, `https://github.com/uofs-simlab/func`.

Green, K. R., T. A. Bohn, and R. J. Spiteri (2019), Direct function evaluation versus lookup tables: When to use which?, *SIAM Journal on Scientific Computing*, *41*(3), C194–C218.

Higham, N. J. (2002), *Accuracy and Stability of Numerical Algorithms*, vol. 80, SIAM.

Intel (2019), VTune Amplifier, `https://software.intel.com/en-us/vtune`.

Kumar Meher, P. (2010), An optimized lookup-table for the evaluation of sigmoid function for artificial neural networks, in *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pp. 91–95, doi:10.1109/VLSISOC.2010.5642617.

Kuon, I., R. Tessier, J. Rose, et al. (2008), FPGA architecture: Survey and challenges, *Foundations and Trends® in Electronic Design Automation*, *2*(2), 135–253.

Loh, E., M. L. Van De Vanter, and L. G. Votta (2005), Can software engineering solve the HPCS problem?, in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pp. 27–31, ACM.

Marks, D., J. Domingo, D. Susong, T. Link, and D. Garen (1999), A spatially distributed energy balance snowmelt model for application in mountain basins, *Hydrological Processes*, *13*(12-13), 1935–1959.

Marsh, C. B., R. J. Spiteri, J. W. Pomeroy, and H. S. Wheater (2018), Multi-objective unstructured triangular mesh generation for use in hydrological and land surface models, *Computers & Geosciences*, *119*, 49–67.

Marsh, C. B., J. W. Pomeroy, and H. S. Wheater (2019a), The Canadian Hydrological Model (CHM): A multi-scale, multi-extent, variable-complexity hydrological model – Design and overview, *Geoscientific Model Development Discussions*, *2019*, 1–44, doi:10.5194/gmd-2019-109.

Marsh, C. B., R. J. Spiteri, J. W. Pomeroy, and H. S. Wheater (2019b), CHM source code, `https://github.com/Chrismarsh/CHM`.

Mirams, G. R., C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S.-J. Dunn, A. G. Fletcher, D. G. Harvey, M. E. Marsh, J. M. Osborne, P. Pathmanathan, J. Pitt-Francis, J. Southern, N. Zemzemi, and D. J. Gavaghan (2013), Chaste: An open source c++ library for computational physiology and biology, *PLOS Computational Biology*, *9*(3), 1–8, doi:10.1371/journal.pcbi.1002970.

Pharr, M., and R. Fernando (2005), *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*, first ed., Addison-Wesley Professional.

Quarteroni, A., R. Sacco, and F. Saleri (2010), *Numerical mathematics*, vol. 37, Springer Science & Business Media.

Reis, L., L. Aguiar, D. Baptista, and F. Morgado-Dias (2014), A software tool for automatic generation of neural hardware, *Neuron*, *1*(1), 229–235.

Rudin, W., et al. (1964), *Principles of mathematical analysis*, vol. 3, McGraw-Hill New York.

Thomas, G. B., M. D. Weir, J. Hass, and F. R. Giordano (2005), *Thomas' calculus*, Addison-Wesley.

Viviroli, D., H. H. Dürr, B. Messerli, M. Meybeck, and R. Weingartner (2007), Mountains of the world, water towers for humanity: Typology, mapping, and global significance, *Water Resources Research*, *43*(7), 1–13.

Wikipedia contributors (2019), List of extreme temperatures in Canada — Wikipedia, the free encyclopedia, `https://en.wikipedia.org/wiki/List_of_extreme_temperatures_in_Canada`.

Wilcox, C., M. M. Strout, and J. M. Bieman (2011), Mesa: automatic generation of lookup table optimizations, in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, pp. 1–8, ACM.