UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

BSc thesis

Nikolaj Hey Hinnerskov

Massively Parallel Selection of Stable History Period

in Change Detection for Time Series Data with Missing Values

Supervisors: Cosmin E. Oancea and Dmitry Serykh

Handed in: June, 2021

Abstract

Vast amounts of publicly available satellite time series data enable large-scale monitoring of environmental disturbances such as deforestation or drought. Such monitoring schemes are, however, computationally expensive, warranting execution on massively parallel hardware. This project presents the theoretical background for stable history detection, an essential component of monitoring, along with its implementation in a hardware-agnostic data-parallel language. The implementation is benchmarked on synthetic and real-world data to reveal a two to three orders of magnitude speedup compared to existing publicly available code. Though the implementations are given as high-level specifications, no non-trivial implementation details are left out; including the need for numerically stable linear model fitting. While this project will focus on satellite data with missing values, the implementations can be applied to any time series data.

Contents

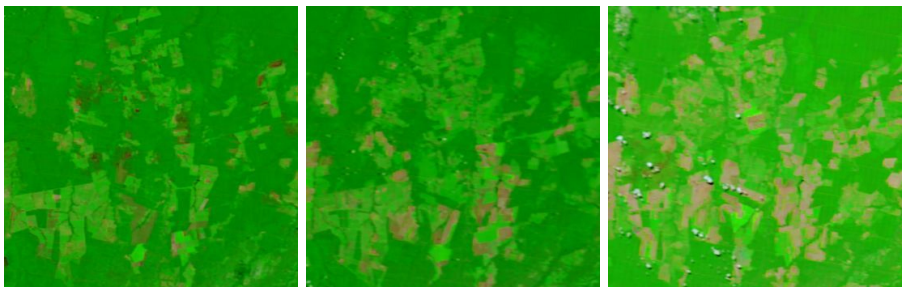

Figure 1: Deforestation in Mato Grosso, Brazil, over the years 2003, 2010 and 2021. Sequencing images of the same area in this manner yields a time series of values for each pixel. The images are NASA Worldview Snapshots (`https://wvs.earthdata.nasa.gov`).

1 Introduction

Accurate detection of landscape changes is a crucial tool in recognizing—and by extension, reducing—negative effects on natural resources and humans from human-induced climate change [Gie+20; VZH12]. Frequent and spatially accurate satellite image data of the earth are publicly available through initiatives such as NASA's Earth Science Data Systems program.[1] Sequencing images of the same area over time yields a time series of values for each pixel (see figure 1), that when monitored, enables detection of landscape changes at a large scale. The satellite image data dealt with in this report are accurate—in some cases—down to 30 square meters per pixel.[2]

For most places on earth, the landscape changes seasonally. Hence, the goal is to detect disturbances in the environment (abnormalities) rather than mere change. To this end, time series based *structural change* detection may be combined with *season trend modelling*. One such method is *break detection for additive season and trend* (BFAST), which is applied for each pixel time series in the image data [VZH12]. The version considered here is specifically BFAST-Monitor (henceforth BFAST).

In essence, monitoring disturbances consist of (1) fitting a linear regression model to the data in a history period that is known to be stable (without disturbances), and (2) detecting the first *break* from this fitted model, if any, during a monitoring period [Gie+20; Zei+02]. This is done for each pixel in the satellite image data. An image covering just 10 km^2 of land may consist of hundreds of thousands of pixels, each containing data for hundreds of time steps. Fitting such change detection schemes, therefore, requires intense amounts of computation. Gieseke et al. [Gie+20] show how to implement this monitoring scheme on commodity hardware (GPUs) and at orders-of-magnitude faster runtimes than existing publicly available code. Using this implementation, landscape changes can be detected at a large scale (in [Gie+20] over the entire continental tropical Africa).

However, this relies on the assumption that a stable history period without disturbances is known. Such a period may be selected using expert knowledge, but it can vary across pixels, and with hundreds of thousands of these to monitor (for just a small image), human supervision quickly becomes intolerable. To alleviate this, a data-driven method for selecting the stable history period is employed. As we progress in this report, we will find that the data-driven selection is much more computation-

[1] https://earthdata.nasa.gov/esds
[2] Data sets are from [Gie+20].

ally demanding than the monitoring itself; it requires fitting not just one model for each time series but fitting a number of models linear in the length of the history period under consideration. This makes it a crucial bottleneck in speeding up the program and, as such, an excellent candidate for parallelisation.

BFAST [VZH12], including stable history detection, is available as a package in the R programming language [R C20]. This package makes use of a structural change detection package [Zei+02], also in the R language, and we shall refer to these two packages collectively as "the R reference implementation". The data-parallel implementation of BFAST by [Gie+20] does not include stable history detection and is, therefore, the subject of this project.

1.1 Contributions

The main contributions of this bachelor project are

1. An explanation of the theory needed for implementing stable history detection in practice. This includes algorithmic descriptions of the reversely ordered CUSUM test (ROC) [PT02] and the recursive residuals [GH84], and discretisations of continuous time equations (11) to (13) from [Zei+02]. While this project will focus on stable history detection, the algorithmic description may be relevant to any application seeking to detect structural change in time series data.

2. A data-parallel implementation of stable history detection targeting *Graphical Processing Units* (GPUs), integration of this with the BFAST implementation by [Gie+20], and validation against the R reference implementation by [VZH12] and [Zei+02].

3. An implementation of a linear model fitting library in Futhark that will handle ill-conditioned problems, including translation of a LINPACK[3] rank-revealing QR decomposition from FORTRAN to Futhark[4].

1.2 Related work

The theory conveyed in this report is compiled from the recursive residuals [GH84], the CUSUM test and variations on this [BDE75; PT02; OB20], structural change detection [Zei+02] and stable history detection [VZH12]. Relevant literature such as Pesaran and Timmermann [PT02] (ROC test), Zeileis et al. [Zei+02] (an R package with the CUSUM test) or Otto and Breitung [OB20] (a related "Backward" CUSUM test) detail the mathematics of the ROC and related tests, but no literature that we are aware of include algorithmic descriptions of either the recursive residuals or the ROC test. The algorithms given here are based in part on the theory referenced above and in part on the source code of R language implementations by [Zei+02] and [VZH12].

This project builds on work by Gieseke et al. [Gie+20], which parallelises the monitoring part of BFAST. Another data-parallel implementation exists, but it does not handle missing values (see section 2.2.1) [Meh+18]. Missing values are both an important aspect of BFAST and a hugely limiting factor in the efficient execution on GPUs. A third implementation exists for the Google Earth Engine (GEE) [Ham+20]. In evaluating its correctness, Hamunyela et al. observe several differences between their implementation and the R reference implementation. The differences are attributed to the use of different numerical solvers for fitting linear regression models

[3] https://www.netlib.org/linpack/
[4] https://www.futhark-lang.org

in R and GEE, though they are not explained further. The problem of choosing the right numerical solver shall prove to be a major issue also in implementing stable history detection. One that is tackled in section 4, where two different approaches are discussed and eventually, an implementation of ordinary least squares regression that will match that of R is given.

1.3 Report structure

The report is organised as follows. Section 2.1 introduces data-parallel programming on GPUs. Section 2.2 presents the necessary background on change detection in time series data. Section 3 details the theoretical aspects of implementing stable history detection in practice. In reading this section, it may be beneficial to skip ahead to the algorithmic descriptions and then return to the theory afterwards. Section 4 is an intermezzo between the theoretical part and its data-parallel implementation and is not strictly necessary to understand the rest of the report; however, the implementation would not validate in its absence. Section 5 discusses various parallelisation strategies and presents a data-parallel implementation. Sections 6 and 7 then benchmark and validate this. Finally, section 8 provides concluding remarks, including future work.

1.4 Mathematical preliminaries

The reader is assumed to be familiar with basic linear algebra and probability theory. This includes matrices, vectors, norms, linear systems of equations and orthogonal factorization; and random variables and probability distributions. The latter of which is not as important as the first. We highlight the concepts most crucial to the readability of this report through a set of brief and informal introductions:

Linear regression models. Given a data set $\{y, x_1, \ldots, x_k\}$, a linear model assumes a linear relationship between the target value y and the column vector of k regressors $\mathbf{x}$:

$$y = \sum_{i=1}^{k} x_i \beta_i = \mathbf{x}^T \boldsymbol{\beta}. \tag{1}$$

where $\boldsymbol{\beta}$ is a column vector of k parameters. Often we impose that $x_1 = 1$ so that β_1 is the intercept—a type of offset for the target value. The linear model assumption is practical in nature and will rarely hold in reality, meaning a linear combination that describes the target values perfectly does not exist. In this case, we *estimate* the best solution to the above equation by choosing $\hat{\boldsymbol{\beta}}$ so that the squared difference between the true target value y and the model prediction $\hat{y} = \mathbf{x}^T \hat{\boldsymbol{\beta}}$ is minimised. We call $\hat{\boldsymbol{\beta}}$ the *ordinary least squares* (OLS) estimate.

Non-linear response from linear models. Linear models are not limited to linear responses. We can predict target values, y, that are non-linear. For example, if we fit the model

$$y = w_2 x^2 + w_1 x + w_0$$

we need only estimate a linear relationship between regressors x^2 and x and the target value y using parameters w_2, w_1 and w_0. Yet the response is clearly quadratic. In practice we do this by first augmenting the regressors; say we have a value $x_1 = x$, then we pre-compute $x_2 = x^2$ and treat this as a fixed value:

$$y = w_2 x_2 + w_1 x_1 + w_0.$$

Statistical hypothesis testing. A *statistical hypothesis* is a statement whose truth is testable based on observed data. Here, we outline the steps required to perform a statistical hypothesis test. The point is not to understand each individual step, but to introduce relevant terms and to gain familiarity with the process from an algorithmic point of view:

1. Formulate a hypothesis whose truth is not known by stating a *null hypothesis* and its *alternative*. For example, a null hypothesis could be that the average annual rainfall has remained constant since last year. The alternative is that the average annual rainfall has changed.
2. Collect data that will be used to test the hypothesis. For example, we observe the rainfall on only some days in each month of the year, and average this.
3. Choose a *confidence level* α. For example, $\alpha = 5\%$.
4. Assume that the collected data are samples from from a random variable X with some known distribution.
5. Then, using this assumption, calculate a *test statistic* resulting in a p-value (the details of which are not needed here). The p-value is the probability of obtaining results that are at least as unlikely as the observed result given that the null hypothesis is true.
6. If the p-value is less than α, then we conclude that our sample is sufficiently unlikely under the null hypothesis to *reject* it, meaning we consider the alternative to be true. If the p-value is greater than α, we *accept* the null hypothesis, meaning we consider it true.

See [Kre06, chapter 25] for more details.

1.5 Notation in pseudocode

An array-programming style of pseudocode is used in the algorithms in this report. Regarding indexing, take $\mathbf{v}[a : b]$ to mean values of $\mathbf{v}$ at indices a through $b - 1$ and take the absence of boundaries to mean all values, e.g. $\mathbf{v}[1 :]$ is all values except the first and $\mathbf{v}[:]$ is simply all values. All of this extends to 2D-indexing for which we use syntax similar to $\mathbf{v}[i, j]$. Further, operations are vectorised (applied element-wise) wherever appropriate.

A consequence of the indexing is that $v[i]$ is 0-indexed, while subscripts such as v_i are 1-indexed. That is, if $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ then v_1 is the same as $v[0]$ and so on. While not the most fortunate notation, this allows a closer correspondence between equations and algorithms.

2 Background

The background section is split into two parts. One is about programming massively parallel hardware (section 2.1). The other is about change detection in satellite time series data (section 2.2).

2.1 Programming massively parallel hardware

The data-parallel implementation given in this report targets *Graphical Processing Units* (GPUs). A GPU is a hardware acceleration device that attaches to a *host*. The host is a computer with one or more CPUs and some random access memory. From a programmer's perspective,[5] a GPU consists of a number of threads and several types of memory. The threads are organised into *blocks*, each containing a number of *warps* [Hen17, p. 53]. A warp is a hardware-dependent fixed number of threads—typically 16 or 32—that execute instructions in lock-step [Oan18; Gie+20]. It follows that a warp is the most fine-grained control of threads on a GPU available to the programmer. The *block size* is often limited to a maximum of 32 warps of 32 threads each (1024 threads in total).[6] As for the memory, two types are important to the ensuing discussion: (1) *shared memory*, which is fast on-chip memory shared across all threads in a block (typically limited to less than 100 KiB per block), (2) *global memory*, which is slow off-chip memory shared across all threads on the GPU (typically several gibibytes in size) [Hen17, p. 56].

Conventional GPU programs consist of device code and host code. The device code consists of *kernels* which are small, sequential programs to be run on the GPU. Kernels are written[7] with a *single instruction, multiple threads* (SIMT) model in mind, where each instruction in the kernel is executed by multiple threads, each with their own slice or copy of the data. Upon launch of a kernel, the host code orchestrates memory allocation and sets the block size. This creates a barrier between the compute part and its execution; e.g. we have to decide, in advance, how threads are to be grouped into blocks which in turn will have an effect on the usage of shared memory. Efficient utilisation of GPU resources thus has to be facilitated by the host code. We will return to this point later.

The data-parallel implementation given in this report is not a conventional GPU program with host and device code, but it will be compiled into one.[8] The program source code is instead written in the purely functional data-parallel array language, Futhark. Futhark programs are written using "bulk" operations on arrays. To facilitate this, we are given a set of operators called *second-order array combinators* (SOACs), whose nested composition, like pieces of a puzzle, build programs that are parallel in infinitely many cores (or threads) [Oan18; Hen17]. In reality, infinitely many cores are, of course, not available to us, but programming as if they were, allows us to essentially abstract away the host part of the GPU program and instead let it be handled by the compiler.[9] Still, it is beneficial to keep the SIMT model in mind when programming in Futhark. Ultimately, the program will adhere to this model when compiled to target the GPU, and so we may have to consider the hardware constraints presented in the paragraphs above. The SOACs imitate familiar higher-order functions such as `map`, `filter` and `reduce`[10] and, the reader can largely assume that

[5] at least in CUDA/OpenCL-like models of programming GPUs.
[6] This is for NVIDIA GPUs, AMD may differ.
[7] Again, in CUDA/OpenCL-like models.
[8] It will be compiled to OpenCL.
[9] The technical term for this is implicit parallelism.
[10] A parallel fold that assumes it is given an associative operator.

a familiar name will have familiar semantics.

2.1.1 A Futhark primer Futhark is a purely functional array language. A consequence of being an array language is that operations are performed on arrays, not lists—and somewhat related to this, that the precision of primitive types must be given explicitly. For example, `i8` is a signed 8-bit integer and `f64` is a double precision float [EHO18]. Extending this to array types, `[n]f64` is an array of n floats.

The central element of Futhark is its SOACs, which mirror higher-order functions from conventional functional languages only they may be executed in parallel. The type and semantics of each SOAC used in this report follow:[11]

- `val map 'a [n] 'x: (f: a -> x)-> (as: [n]a)-> *[n]x`
  Apply the function `f` to each element in the array `as`. Here `[n]` is a *size-parameter* meaning the value of `n` is inferred from `as`.
- `val reduce [n] 'a: (op: a -> a -> a)-> (ne: a)-> (as: [n]a)-> a`
  Combine array elements using associative operator `op` whose neutral element is `ne`. In other words, a parallel fold that assumes it is given an associative operator. If the operator is not associative, the result is indeterministic when run in parallel.
- `val scan [n] 'a: (op: a -> a -> a)-> (ne: a)-> (as: [n]a)-> *[n]a`
  Combine array elements from left to right, storing intermediate results. This is also called an inclusive prefix sum or cumulative sum. While the semantics are sequential (left to right), it is parallel in practice, so once again, the operator must be associative.

Futhark also supports sequential loops and in-place updates using *uniqueness types* that will—at compile-time—determine whether an in-place update is safe. Unique types are decorated with an asterisk: `*[n]f64`. Futhark does not support irregular (jagged) arrays.

2.1.2 Constraints on parallelism The Futhark compiler does not support nested irregular parallelism, which is to compose parallel operations with parallel operations of different sizes. An example is a map inside a map, where the size of the innermost operation varies:

```
map (\i -> map f (1...i)) [1,2,...,n]
```

If all parallelism in the above code is to be exploited and this is to be executed on a GPU, we must allocate n blocks of n threads, because the inner map may require up to n threads (recall that the block size is common to all blocks). Or, we can rewrite (transform) the code by noting that a map composed with a map equals a map on the flattened data:

```
map f [1, 1,2, ..., 1,2,...,n]
```

The difficulty of such transformations increase with with the complexity of the operations, but there exists rewrite rules for all the SOACs used in this report. Particularly, the rewrite rules given in [Oan18, p. 47] preserve asymptotically the number of operations performed by the nested irregular program (this type of flattening was first introduced in [Ble96]). Using these rewrite rules it is possible to systematically derive a flat implementation from any nested irregular program consisting of the SOACs presented here.

[11] These descriptions are based on the Futhark prelude documentation found at `https://futhark-lang.org/docs/prelude/`.

2.1.3 Incremental flattening Flattening techniques are also used to map program-level parallelism to hardware-level parallelism. While our program may be parallel in infinitely many cores, the hardware will not be. Parallelism in excess of what the hardware has to offer is at best wasted and it may be beneficial to instead sequentialise operations.

In [Hen+19] a technique called *incremental flattening* is proposed where all possible mappings from program-level parallelism to hardware-level parallelism are generated. This results in multiple code versions that are then autotuned on a set of representative data sets, generating *static* thresholds that determine the level of nested parallelism to exploit. At each point in the code where a `map` operator is encountered, the technique will (from [Hen+19]):

1. map the program parallelism onto the current hardware level and sequentialise the inner parallelism,
2. map the discovered program parallelism onto the current hardware level and recursively map the remaining inner parallelism at the next lower hardware level and
3. recursively continue flattening the current hardware level.

This generates semantically equivalent code version that are then guarded by thresholds. The thresholds are checked against the dimensions of the data at runtime and the appropriate code version is run. This way we retain all parallelism in the program when needed, but use efficient sequential code when it is in excess.

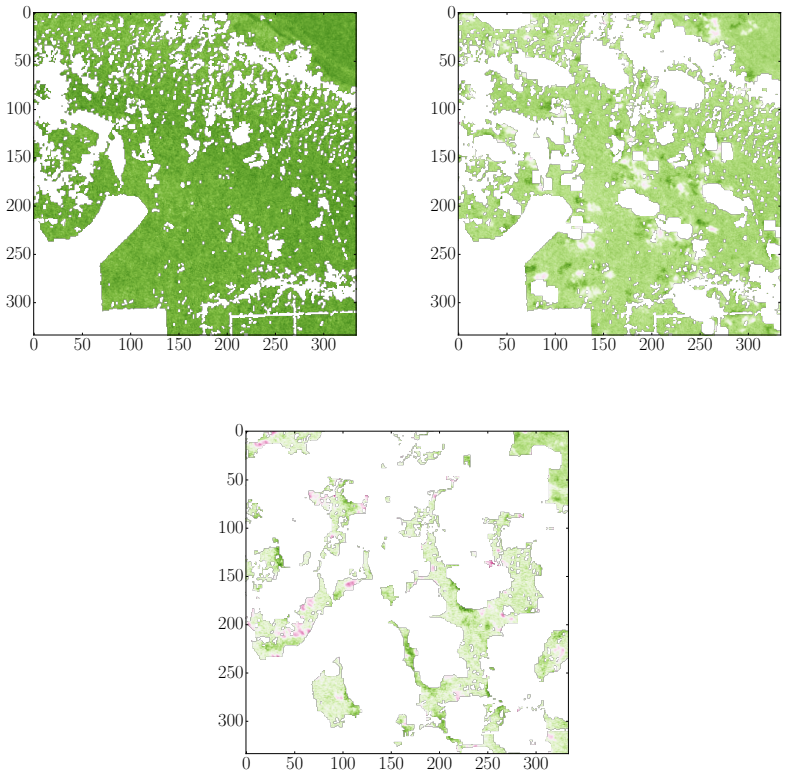

Figure 2: Satellite image data of a 10 km^2 area in Peru (about 100-thousand pixels) [Gie+20]. We show data for three time steps out of 235 to illustrate how clouds obscure different pixels at different points in time (sharp edges are imaging effects). Note that these are not true colors, but visualized vegetation indices.

2.2 Change detection in satellite time series

In this section, we introduce satellite time series data (section 2.2.1), the season-trend model (section 2.2.2) and change detection (section 2.2.3). These are the three central components of BFAST (section 2.2.4).

2.2.1 Satellite time series data Artificial satellites orbit the earth, passing over the same area at regular intervals (for example, every five days). Satellite image data are thus collected over time, yielding for each pixel a time series of values [Gie+20]. In figure 2, we show example satellite image data of Peru that will be used throughout this report. The data results from preprocessing of satellite imagery and consists of *vegetation indices*, which are a quantitative indicator of the amount of vegetation for each pixel [Gie+20]. For our purposes, we merely treat each pixel as a time series of reals. Figure 2 also illustrates how clouds may obscure the view of the ground, causing the corresponding pixels to contain little or no information. These pixels are typically masked in the resulting data, meaning each pixel time series may have many missing values.

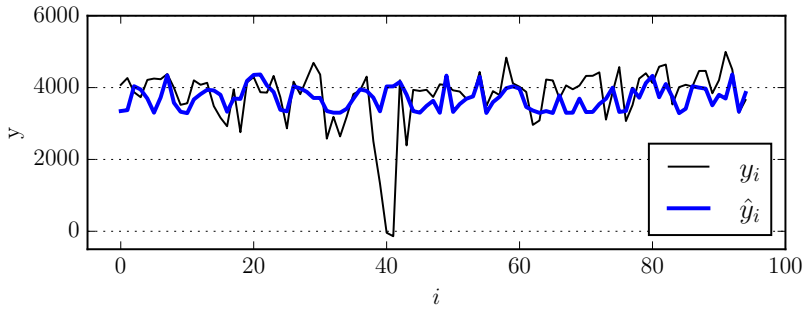

Figure 3: The season-trend model from equation (2) is fit to time series data from a single pixel in Peru (see figure 2). Here, y_i is the observed value and $\hat{y}_i$ is the predicted value using an OLS estimate for the parameters of equation (3).

2.2.2 The season-trend model The satellite time series data are modelled using a method that accounts for seasonal and trend changes. Given a time series of reals, $y_1, \ldots, y_N$, we assume a *season-trend model* [VZH12]

$$y_i = \alpha_1 + \alpha_2 i + \sum_{j=1}^{p} \gamma_j \sin\left(\frac{2\pi j i}{f} + \delta_j\right) + \epsilon_i, \qquad i = 1, 2, \ldots, N, \tag{2}$$

where α_1 is the intercept, α_2 is the trend and the third term captures the seasonal pattern with *amplitudes* $\gamma_1, \ldots, \gamma_p$ and *phases* $\delta_1, \ldots, \delta_p$. These are the unknown parameters to be estimated. A known parameter is f which specifies the frequency of observations over a 365-day period. For example, $f = 365$ for observations that are each one day apart and $f = 52$ for observations that are 7 days apart. Another known parameter is p which determines the number of *harmonic terms* used to account for seasonality.[12] The error term ϵ_i captures the remaining error [VZH12; Gie+20]. On figure 3 the season-trend model is fit to time series data with $p = 3$.

The model from equation (2) can be written as a standard linear regression model [VZH12]

$$y_i = \mathbf{x}_i^T \boldsymbol{\beta}_i + \epsilon_i, \tag{3}$$

$$\mathbf{x}_i = (1, i, \sin(2\pi 1 i/f), \cos(2\pi 1 i/f), \ldots, \sin(2\pi p i/f), \cos(2\pi p i/f))^T, \tag{4}$$

$$\boldsymbol{\beta}_i = (\alpha_1, \alpha_2, \gamma_1 \cos(\delta_1), \gamma_1 \sin(\delta_1), \ldots, \gamma_p \cos(\delta_p), \gamma_p \sin(\delta_p))^T, \tag{5}$$

where at time i, y_i and ϵ_i are as before, $\mathbf{x}_i$ is a column vector of $k = 2p + 2$ regressors and $\boldsymbol{\beta}_i$ is a column vector of k regression parameters. Using this form, we can estimate the parameters $\boldsymbol{\beta}_i$ using ordinary least squares (OLS) methods.

2.2.3 Structural change We write $\boldsymbol{\beta}_i$ with a subscript i to indicate that the parameters of the model may vary over time (but note that the elements of $\boldsymbol{\beta}_i$ are independent of i). If they do vary over time, meaning at least one of the parameters at some time i is not equal to the others, we say that there is *structural change*. For example, if $\boldsymbol{\beta}_1 = \boldsymbol{\beta}_2 \neq \boldsymbol{\beta}_3$ then there is structural change with a *structural break*

[12] p is typically small, e.g. $p = 3$.

occurring at time 3 [Zei+02; Han01]. Alternatively, the regression parameters are time-invariant,

$$\boldsymbol{\beta}_1 = \boldsymbol{\beta}_2 = \cdots = \boldsymbol{\beta}_N = \boldsymbol{\beta}, \tag{6}$$

in which case there is no structural change and the same model applies to observations $y_1, \ldots, y_N$ [Zei+02]. When the parameters are time-invariant we also say that the model is *stable* [VZH12].

To relate this to satellite time series data, we can expect to see that the regression parameters are constant for one period, $i = 1, 2, \ldots, n$, but change over another, $i = n+1, \ldots, N$ [VZH12]. Figure 3 gives an example of this where the model is stable until a structural break occurs at time $i = 41$.[13] Though, shortly after, the model is once again stable. This divides the time series into two *stable periods* in which the regression parameters are time-invariant: one before the break and one after the break. For most of the remainder of this text, we are concerned with detecting a stable period in a given time series.

2.2.4 BFAST-monitor BFAST combines season-trend modelling with structural change detection and is applied independently for each pixel in the satellite image data. Given a pixel time series, the main steps are to:

1. divide the time series into a history period, $i = 1, 2, \ldots, n$, and a monitoring period, $i = n + 1, \ldots, N$. The end of the history period n is a user-specified parameter.
2. Select the start, l, of a period in which a stable season-trend model can be fitted. We call this the *stable history period*. The selection can be done manually or by using data-driven methods.
3. Fit the season-trend model on the stable history period, $i = l, l + 1, \ldots, n$.
4. Detect the first structural break from this model during the monitoring period, if it exists.

In this project we are concerned only with selecting the start of a stable history period using data-driven methods.

[13] This structural break was determined using a test introduced later.

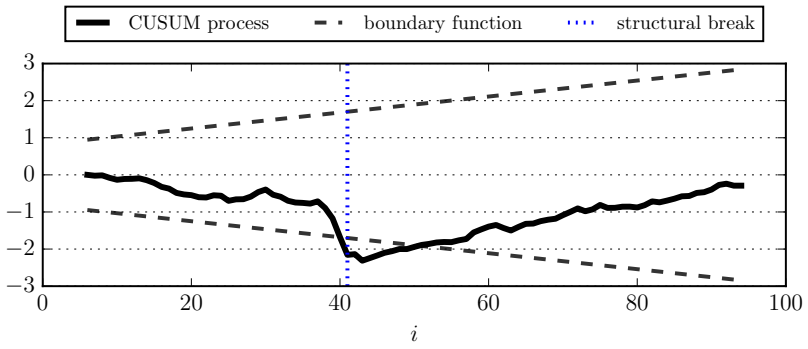

Figure 4: A CUSUM test on the time series in figure 3. A structural break is detected at time step 41, where the CUSUM process crosses the boundary function. Note how the boundary can be crossed with either negative or positive magnitude.

3 Stable history detection

The history period of BFAST ends at a user-specified point in time, n. What remains is to select a starting point that will make use of as much data as possible, but ensure a stable period. To this end, we test for structural change in the history period $i = 1, \ldots, n$. Specifically, we test the null hypothesis of "no structural change", which is exactly equation (6), against the alternative that the regression parameters vary over time. If the alternative is true, a structural break will have occurred at some point in time, and we use this break point to set the start of the stable history period [Zei+02]. In this section, it is detailed how to carry out such a test. First, the theoretical aspects are presented, then algorithms are given. From now on, when referring to "the time series" we mean the history part, $i = 1, 2, \ldots, n$, only.

Overview In testing the null hypothesis of no structural change, a standard statistical tool is to consider cumulative sums of prediction errors [OB20]. Among these methods is the CUSUM test, which evaluates a type of cumulative prediction error against a *boundary function*. The boundary function is a linear function that the cumulative prediction error will cross only with low probability under the null hypothesis (this probability is a parameter that is typically set to 5%). More specifically, the cumulative prediction error is a *CUSUM process* whose mean is zero under the null hypothesis. If the alternative is true and there is but a single structural break point, the CUSUM process will deviate systematically from its mean after this point in time. We detect the structural break only when the CUSUM process crosses the boundary function [VZH12; Zei+02; OB20; BDE75]. An example CUSUM test is shown in figure 4. A lot of statistical theory is omitted here; see [Zei+02] and [OB20] for details.

The test will only detect the *first* structural break, however, there may be multiple breaks in the history period. Since the end of the stable history period is fixed and we are to detect its start, we must know when the break closest to the end occurs. This is achieved by instead starting at the end of the proposed history period and then, moving backwards in time, detecting the first break. This is referred to as a reverse ordered CUSUM (ROC) test [PT02]. In practice, it simply amounts to performing a

"forward" CUSUM test on the reversed input data. To bring figure 4 into this setting, we can imagine that the data has already been reversed. Then the break point is the start of a stable history period and the period immediately before this (the values to the left of the break) is stable history.

Before we introduce the ROC test formally, we look at the prediction error, whose definition was eluded earlier.

3.1 Recursive residuals

In this section, we define the type of prediction error used for the ROC test, namely *recursive residuals*, which are standardized one-step-ahead prediction errors. In the two following paragraphs, we dissect this statement before giving a formal definition.

A *one-step-ahead prediction error* is the residual obtained when using a model, fit on the first i data points in a time series, to predict data point $i+1$. For a time series of length n and a model with k parameters, we may compute $n-k$ such one-step-ahead prediction errors: fitting a model with k parameters requires at least k data points, and so we can at most repeat the initial definition for $i = k, \ldots, n-1$. For the recursive residuals we compute and store all $n-k$ one-step-ahead predictions. See figure 5 for an example on the time series from figure 3.

We then *standardize* the one-step-ahead prediction errors. Here, standardization refers to a linear transformation of the one-step-ahead prediction errors so that they have zero mean under the null hypothesis of no structural change (recall equation (6)) [GH84]. *Under the null hypothesis* is key; if structural change occurs at some point in time, the recursive residuals will only have zero mean up until this point. Since the CUSUM process cumulates recursive residuals, it will systematically deviate from its mean of zero after a structural break point.

To formally define the recursive residuals, let $\hat{\boldsymbol{\beta}}_i$ be the ordinary least squares estimate of the regression coefficients $\boldsymbol{\beta}$ based on the first i observations in a time series. Similarly, let $\mathbf{X}_i = (\mathbf{x}_1^T, \ldots, \mathbf{x}_i^T)$ and $\mathbf{y}_i$ be the $i \times k$ regressor matrix and $i \times 1$ vector of target values based on the first i observations of this time series, respectively. Then

$$\hat{\boldsymbol{\beta}}_i = (\mathbf{X}_i^T \mathbf{X}_i)^{-1} \mathbf{X}_i^T \mathbf{y}_i, \quad i = k, \ldots, n, \tag{7}$$

and the recursive residuals are defined as

$$r_i = \frac{y_i - \mathbf{x}_i^T \hat{\boldsymbol{\beta}}_{i-1}}{\sqrt{1 + \mathbf{x}_i^T (\mathbf{X}_{i-1}^T \mathbf{X}_{i-1})^{-1} \mathbf{x}_i}}, \quad i = k+1, \ldots, n. \tag{8}$$

Here, the numerator is the ith one-step-ahead prediction error while the denominator ensures standardization (the details of which are left out for brevity). The recursive residuals for the time series in figure 3 are depicted on figure 6.

Update formulas From equation (8) we realize that calculation of the recursive residuals is a costly operation: we have to fit $n-k$ models for a single time series. Note that the OLS estimate, $\hat{\boldsymbol{\beta}}_i$, from equation (7) requires computing the *covariance parameters* $(\mathbf{X}_i^T \mathbf{X}_i)^{-1}$. That is, we must perform $n-k$ inversions for a single time series. Fortunately, there exists update formulas that, in theory, allows us to compute the inversion only once. If we compute the initial fit on k observations to obtain covariance parameters $\mathbf{M}_k = (\mathbf{X}_k^T \mathbf{X}_k)^{-1}$ and regression parameters $\hat{\boldsymbol{\beta}}_k$. Then the subsequent covariance parameters will be given by [GH84]:

$$\mathbf{M}_i = \mathbf{M}_{i-1} - \mathbf{d}\mathbf{d}^T \frac{1}{1 + \mathbf{x}_i^T \mathbf{d}}, \quad i = k+1, \ldots, n, \tag{9}$$

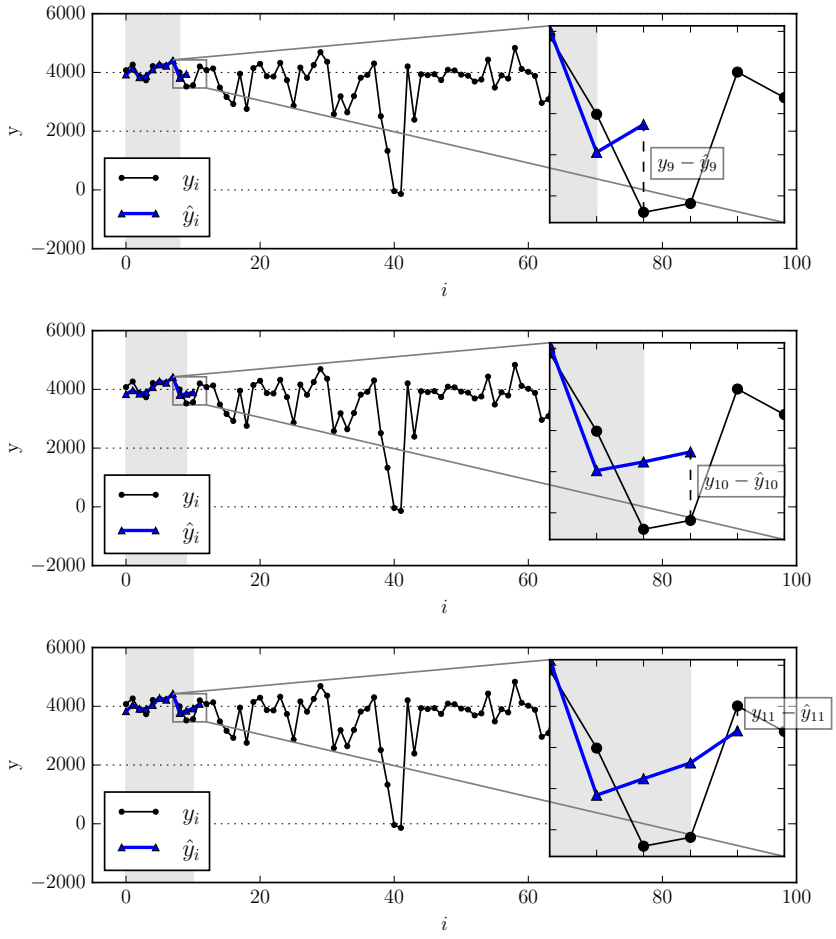

Figure 5: This figure illustrates the computation of one-step-ahead prediction errors on the time series from figure 3. Denote by $\hat{y}$ a season trend model with $k = 8$ parameters to be estimated using OLS methods. *Top row*: $\hat{y}$ is fit on the first eight observations $y_1, \ldots, y_8$ (top row, gray area) and the residual $y_9 - \hat{y}_9$ is stored for later use. This is the first one-step-ahead prediction. *Mid row*: $\hat{y}$ is fit on the first nine observations $y_1, \ldots, y_9$ and the residual $y_{10} - \hat{y}_{10}$ is stored for later use. This is the second one-step-ahead prediction. *Bottom row*: $\hat{y}$ is fit on the first ten observations $y_1, \ldots, y_{10}$ and the residual $y_{11} - \hat{y}_{11}$ is stored for later use. This is the third one-step-ahead prediction. We repeat this procedure for the rest of the time series.

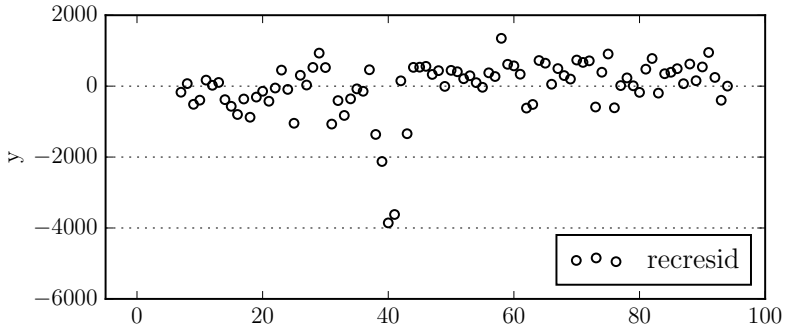

Figure 6: The recursive residuals for the time series in figure 3. They were obtained by standardizing the one-step-ahead prediciton errors from figure 5.

where $\mathbf{d} = \mathbf{M}_{i-1}\mathbf{x}_i$, and the corresponding regression parameters by

$$\hat{\boldsymbol{\beta}}_i = \hat{\boldsymbol{\beta}}_{i-1} + \mathbf{M}_i\mathbf{x}_i(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}}_{i-1}). \tag{10}$$

To show that this is a more efficient way to compute the fits needed for the recursive residuals, we provide a short time complexity analysis: with or without the update formulas, we have to compute the initial OLS estimate $\hat{\boldsymbol{\beta}}_k = (\mathbf{X}_k^T\mathbf{X}_k)^{-1}\mathbf{X}_k^T\mathbf{y}_k$, whose cost is dominated by the number of operations performed when squaring the $k \times k$ regressor matrix $\mathbf{X}_k$. Namely, there are k^2 elements in $\mathbf{X}_k^T\mathbf{X}_k$, each of which require $O(k)$ operations to compute. Hence the total cost for this initial step is $O(k^3)$ operations.

Without update formulas, we then have to perform another $n-k-1$ fits, with each fit including one more observation than the previous. That is, we have to compute equation (7) for $i = k+1, \ldots, n$. Again, this is dominated by squaring the regressor matrix $\mathbf{X}_i$, whose result contains k^2 elements. This time each of these elements require $O(i)$ operations to compute. Hence the cost for each subsequent fit is $O(ik^3)$ operations. For sake of argument, assume that each of these have the lower cost of $O(k^3)$ and let $m = n - k$. Then the total cost without update formulas is upper bounded by $O\left((n-k)k^3\right) = O(mk^3)$.

With update formulas there are no further matrix-matrix multiplications of this size to perform (nor are there any further matrix inversions to perform, whose complexity otherwise matches that of matrix-matrix multiplication). Though we do have to perform the matrix multiplication $\mathbf{d}\mathbf{d}^T$ where $\mathbf{d}$ is a $k \times 1$ vector (equivalent to an outer product). There are k^2 elements in the resulting matrix, but each element warrants only a single multiplication operation. Matrix-vector multiplications have exactly the same asymptotic cost and so the total cost with update formulas is upper bounded by $O\left((n-k-1)k^2 + k^3\right) = O\left((m-1)k^2 + k^3\right)$. Since $m > 1$, we see that the update formulas have better or equal asymptotic complexity. For $m > k$, the update formulas clearly have better asymptotic complexity.

This analysis assumes usage of the matrix multiplication algorithm that results from the definition of matrix-matrix multiplication, which has cubic asymptotic cost. Algorithms with better asymptotic guarantees exist, e.g. a recent result achieves a subcubic upper bound [AW20]. This does however not change the analysis; for matrix multiplication to not dominate the asymptotic cost we would need a subquadratic upper bound. Alternatively, we could change the argument to rely on similar costs for matrix inversion.

Algorithm It is possible to formulate an algorithm directly from equations (7) to (10), but empirical tests reveal that this approach is prone to numerical instability once implemented (see section 4.3). In an attempt to alleviate this, algorithm 1 differs from the equations in two ways:

1. Lines 2 and 11 check the update formula against an OLS fit for the first few iterations. This is done to ensure that the update formulas produce a model that is approximately equal to the OLS fit.

2. Lines 1 and 12 do not solve equation (7) directly when computing this OLS fit. In fact it is sometimes required to drop parameters from the model in order to ensure a meaningful fit (see section 4.1). This in turn means that "approximately equal" in item 1 has to check both the number of parameters used in the model and the parameters themselves.[14]

Algorithm 1 makes short of mentioning how to do this OLS fitting—including how to obtain the rank (number of parameters in the model) of submatrices of $\mathbf{X}$. See section 4 for this and a discussion about numerical instability in linear model fitting and its ties to the recursive residuals.

Algorithm 1 Recursive residuals

Require: A vector $\mathbf{y} = (y_1, \ldots, y_n)^T \in \mathbb{R}^n$ containing a time series of targets, and the regressor matrix $\mathbf{X} = (\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T) \in \mathbb{R}^{n \times k}$ with each $\mathbf{x}_i$ for $i = 1, \ldots, n$ as in equation (4). Also $n > k$.
Ensure: Recursive residuals $\mathbf{r} \in \mathbb{R}^{(n-k)\times 1}$

1: $\mathbf{M}_k$, $\hat{\boldsymbol{\beta}}_k = \text{fit}(\mathbf{X}[: k, :], \mathbf{y}[: k])$ ▷ initial OLS regression, see section 4
2: check = true
3: **for** $i = k + 1, \ldots, n$ **do**
4: $\quad \mathbf{x}_i = i$th row of $\mathbf{X}$ ▷ $(k \times 1)$
5: $\quad \mathbf{d} = \mathbf{M}_{i-1}\mathbf{x}_i$ ▷ $(k \times 1)$
6: $\quad f = 1 + \mathbf{x}_i^T \mathbf{d}$ ▷ for standardization
7: $\quad \hat{y} = \mathbf{x}_i^T \hat{\boldsymbol{\beta}}_{i-1}$ ▷ one-step-ahead prediction
8: $\quad r_{i-k} = (y_i - \hat{y})/\sqrt{f}$ ▷ recursive residual
9: $\quad \mathbf{M}_i = \mathbf{M}_{i-1} - \mathbf{d}\mathbf{d}^T/f$ ▷ update covariance parameters $(k \times k)$
10: $\quad \hat{\boldsymbol{\beta}}_i = \hat{\boldsymbol{\beta}}_{i-1} - \mathbf{M}_i \mathbf{x}_i (y_i - \hat{y})$ ▷ update parameters $(k \times 1)$
11: $\quad$ **if** check **then**
12: $\quad\quad \mathbf{M}_i$, $\mathbf{b} = \text{fit}(\mathbf{X}[: i, :], \mathbf{y}[: i])$ ▷ OLS regression
13: $\quad\quad$ If $\mathbf{X}[: i-1, :]$ and $\mathbf{X}[: i, :]$ both have full rank
14: $\quad\quad\quad$ and parameters from update formula, $\hat{\boldsymbol{\beta}}_i$, and full
15: $\quad\quad\quad$ fit, $\mathbf{b}$, are approximately equal, set check = false.
16: $\quad\quad \hat{\boldsymbol{\beta}}_i = \mathbf{b}$
17: $\quad$ **end if**
18: **end for**
19: return $\mathbf{r} = (r_1, r_2, \ldots, r_{n-k})$

[14] When referring to the values of the parameters, approximately equal is comparison subject to some tolerance value. We leave the actual implementation up to interpretation. A C++ version of strucchange, used by newer versions of the R reference implementation of BFAST, makes use of an absolute check while the R version of strucchange uses both relative and absolute checks. The absolute check looks like so: `mean(abs(a - b))<= tol` where `tol` is some reasonable floating point value. In the reference implementations, the square root of machine epsilon divided by k is used. With 64-bit floating point numbers and $k = 8$ this is approximately 1.8×10^{-9} on an Intel i5-4460 CPU.

3.2 The ROC test

We have seen what a reverse ordered CUSUM test is in the context of stable history detection. In this subsection it is introduced formally. We start by defining two key components from [Zei+02]: the *CUSUM process* and the *boundary function*. These are given in a continuous normalized time ($0 \leq t \leq 1$), later we derive formulas for working with discrete time input ($i = 1, 2, \ldots, n$).

The CUSUM process $W(t)$ contains cumulative sums of recursive residuals and is defined as

$$W(t) = \frac{1}{\sigma\sqrt{n-k}} \sum_{i=k+1}^{k+\lfloor t\cdot(n-k)\rfloor} r_i \qquad (0 \leq t \leq 1). \tag{11}$$

where r_i is the ith recursive residual, $n-k$ is the total number of recursive residuals (they are computed only after the first k data points) and $k+\lfloor t\cdot(n-k)\rfloor$ is the last recursive residual considered at normalized time t [Zei+02].

The boundary is a linear function, $b(t)$, whose crossing detects structural change in the model used to compute the recursive residuals:

$$b(t) = \lambda \cdot (1 + 2t) \qquad (0 \leq t \leq 1) \tag{12}$$

where λ is a constant that depends on a user specified confidence level α. This confidence level is the probability that the CUSUM process crosses the boundary when there is no structural change [Zei+02].

To determine whether a crossing of the boundary function is significant we further perform a "structural change test" (`sctest`) which results in a p-value. If this p-value is smaller than the confidence level α, the null hypothesis is rejected and a structural break point is declared where the boundary was crossed. Otherwise, the null hypothesis is accepted and the start of the stable history is set to the earliest possible point, $t = 0$.

The significance test requires computing the test statistic S_r [Zei+02]:

$$S_r = \max_t \frac{W(t)}{1+2t} \tag{13}$$

and the p-value, p_{S_r}:

$$p_{S_r} = f(S_r)$$

where

$$f(x) = 2 \cdot [Q(3x) + \exp(-4x^2)(1 - Q(x))] \tag{14}$$

and $Q(x)$ is the complementary cumulative distribution function to the standard normal distribution $\mathcal{N}(0,1)$ [BDE75]:[15,16]

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp\left(-u^2/2\right) du.$$

Equations (11) to (13), given in [Zei+02], are functions of a continuous, normalized time: $0 \leq t \leq 1$. In the next section we adapt them to better suit discrete time input data in the form of $i = 1, 2, \ldots, n$.

[15] While the formula for the p-value is derived from [BDE75, p. 154], the connection with the structural change test was obtained by studying the source code of R package `strucchange`.

[16] Most programming languages contain approximations to this function.

From continuous to discrete time In practice, we wish to find a break in our discrete time input data, so we only need to evaluate the CUSUM test at a series of discrete time steps.

First, it follows that the CUSUM process in equation (11) not just *contains* cumulative sums of recursive residuals, but that the process *is* a cumulative sum of recursive residuals. To see this, first consider a change of variable by defining a linear function j of t, so that $j(t)$ traverses $[0, n-k]$ as t traverses $[0, 1]$. The function j is given explicitly by multiplying t with $(n-k)$:

$$j(t) = t \cdot (n-k).$$

Substituting $t = j(t)/(n-k)$ into equation (11), we then have

$$W'(j(t)) = W\big(j(t)/(n-k)\big) = \frac{1}{\sigma\sqrt{n-k}} \sum_{i=k+1}^{k+\lfloor j(t)\rfloor} r_i \qquad (0 \le t \le 1).$$

Now let i be the integer that takes on all non-zero discrete values in $[0, n-k]$; that is $i = 1, 2, \ldots, n-k$. Solving $j(t) = i$ for t, we get $t = i/(n-k)$, and so

$$W_i = W'\big(j\big(i/(n-k)\big)\big) = W'(i) = \frac{1}{\sigma\sqrt{n-k}} \sum_{i=k+1}^{k+i} r_i \qquad (i = 1, 2, \ldots, n-k),$$

where the third equality holds because i is an integer. W_i is the recursive residual at the ith discrete time step; evaluating W_i for all i, we obtain a cumulative sum of recursive residuals. In other words, we have shown that when only discrete time steps are considered, each recursive residual contributes to the sum exactly once.

Second, it follows that the boundary function in equation (12) is given by

$$b_i = \lambda \cdot \left(1 + 2\frac{i}{n-k}\right)$$

and that the test statistic in equation (13) is given by

$$S_r = \max_i \frac{W_i}{1 + 2\big(i/(n-k)\big)}.$$

The proofs of these two formulas are similar to the one given for the CUSUM process.

Algorithm We conclude this section with an algorithm for carrying out a reverse ordered CUSUM test to detect the start of a stable history period. See algorithm 2. First, lines 2 to 7 compute the CUSUM process on the reversed input data. Here, reversal of $\mathbf{X}$ amounts to reversing the order of the rows only. Then, in lines 9 to 14, the p-value and λ are calculated for the significance test and boundary, respectively. Both make use of f from equation (14). Because λ only depends on the parameter α, it can be precomputed and e.g. stored in a lookup table. In line 16 the significance test is carried out. If the null hypothesis is rejected, the boundary values are produced and t is set to the point at which the CUSUM process crosses this boundary, if it exists. Otherwise $t = 0$. One final note is that the process may cross with either positive or negative magnitude (recall figure 4); checking the absolute value against the positive boundary covers both cases (line 20).

Algorithm 2 Stable history detection (ROC test)

Require: A confidence level $\alpha \in \mathbb{R}$ with $0 < \alpha < 1$, a vector $\mathbf{y} = (y_1, \ldots, y_n)^T \in \mathbb{R}^n$ containing a time series of targets, and the regressor matrix $\mathbf{X} = (\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T) \in \mathbb{R}^{n \times k}$ with each $\mathbf{x}_i$ for $i = 1, \ldots, n$ as in equation (4). Also $n > k$.

Ensure: Time $t \in \mathbb{N}$ at which the stable history period starts.

— *Compute CUSUM process on reversed data*
$\mathbf{X}_{rev},\ \mathbf{y}_{rev} = \text{reverse}(\mathbf{X}), \text{reverse}(\mathbf{y})$
$\eta = n - k$ ▷ number of recursive residuals
$\mathbf{r} = \text{recresid}(\mathbf{X}_{rev}, \mathbf{y}_{rev})$ ▷ recursive residuals ($\eta \times 1$), see algorithm 1
$\bar{r} = \text{mean}(\mathbf{r})$
$\sigma = \sqrt{\frac{1}{\eta - 1} \sum_{i=1}^{\eta} (r_i - \bar{r})^2}$ ▷ sample SD of rec. residuals
$\mathbf{w} = \dfrac{1}{\sigma\sqrt{\eta}} \cdot \text{prefixSum}(\mathbf{r})$ ▷ CUSUM process ($\eta \times 1$)
— *Perform structural change test*
$\mathbf{x} = \text{copy } \mathbf{w}$
$\mathbf{j} = [\frac{1}{\eta}, \frac{2}{\eta}, ..., \frac{\eta}{\eta}]$
$\mathbf{x} = \mathbf{x} \cdot 1/(1 + 2 \cdot \mathbf{j})$
$p = f(\max(\text{abs}(x)))$ ▷ f from equation (14)
— *Perform significance test and find break*
$\lambda = \text{f}(\alpha)$ ▷ independent of y, so can be precomputed
$t = 0$
if $p < \alpha$ **then**
  for $i = 0, \ldots, \eta - 1$ **do** ▷ boundary of limiting process ($\eta \times 1$)
    $\mathbf{b}[i] = \lambda \cdot (1 + \frac{2i}{\eta})$
  end for
  $\text{inds} = \text{where}(|\mathbf{w}| > \mathbf{b})$ ▷ get indices where predicate is true
  if $\text{length}(\text{inds}) > 0$ **then**
    $t = \eta - \min(\text{inds})$ ▷ get index of first break in reversed data and convert this to non-reversed
  end if
end if
return t

4 Linear models and ill-conditioned problems

This section details a set of problems related to numerical stability and *ill-conditioning* encountered when fitting linear models as part of this project. First, we highlight a key mathematical assumption on ordinary least squares estimation that will serve as background for the ensuing discussion. Second, we illustrate how methods for computing the OLS estimate may differ in terms of numerical stability. Third, we discuss the need for these methods and put this in relation to previous work.

The outcome of this section is a library for fitting linear least squares models in Futhark that will obviate numerical issues often subtle in practice to programmers. We defer the implementation to appendix A.

4.1 Violating mathematical assumptions

The linear regression model and the ordinary least squares (OLS) method for estimating its parameters is subject to a number of mathematical assumptions. One assumption—quite subtle in practice—is that of no *linear dependence* between the regressors in $\mathbf{X}$ [Hay00, p. 10]. Regressors correspond to columns of $\mathbf{X}$, and so two regressors are linearly dependent if one column is a linear combination of the other. If this is the case, the execution of a program designed to compute OLS estimates will produce results that cannot be interpreted as such. This and the following sections will lead to an OLS estimating program that will produce meaningful results, even when assumptions are violated initially.

Another way to address the assumption of no linear dependence between regressors is to consider the *rank* of $\mathbf{X}$. The rank of a matrix is the maximal number of linearly *independent* columns [Mac95]. So if $\mathbf{X}$ has full rank, the regressors will be linearly independent, as desired. The R reference implementation makes use of a special *rank-revealing* QR decomposition (see [Cha87]) that when combined with dropping columns until $\mathbf{X}$ has full rank will produce a meaningful fit even when model assumptions are initially violated. Note that dropping a column is equivalent to dropping a regressor, the meaningful fit is therefore a linear model with fewer parameters.

This covers the case when columns of $\mathbf{X}$ are truly linearly dependent. In practice, they just have to be close enough for numerical work [Mol04, chapter 5]. If this is the case, the problem is said to be *ill-conditioned*. A more general notion (not necessarily pertaining to regressors) is that of "near singular" matrices. These matrices are invertible, but ill-conditioned so that slight changes make them singular [Lay16, p. 116]. If what we wish to invert depends on previous computations—especially when floating point numbers are involved—we may be inducing such singularity in our program. The next section will attempt to illustrate this.

4.2 Conditioning and sensitivity to numerics

We use the word *conditioning* to indicate how sensitive the solution of a given problem is to changes in the input data. If a problem has a high *condition number*, small changes in input will mean a large change in its solution. Conversely, a small condition number indicates a robustness to slight changes in input [KC09]. Since we are doing numerical work using floating point numbers, it is reasonable to expect that we introduce some degree of slight changes with every operation.

For our purposes it will be sufficient to quantify conditioning for matrices. Formally, the condition number of matrix $\mathbf{A}$, $\kappa(\mathbf{A})$, is defined as the magnitude of $\mathbf{A}$

multiplied by the magnitude of its inverse:

$$\kappa(\mathbf{A}) = ||\mathbf{A}||\ ||\mathbf{A}^{-1}||,$$

where $||\cdot||$ is a matrix norm [KC09].

Other than having a language for this type of problem, we are now able to illustrate why Gauss-Jordan elimination is generally not used to solve linear least squares (LLS) equations in statistical libraries such as statsmodels, numpy or those built in to the R language. The OLS estimate for the parameters, $\boldsymbol{\beta}$, of a linear regression model is given by

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}.$$

A direct way to obtain $\boldsymbol{\beta}$ is thus to compute $(\mathbf{X}^T\mathbf{X})^{-1}$ using Gauss-Jordan elimination (assuming it exists) followed by a single matrix multiplication. However, it can be shown that $\kappa(\mathbf{X}^T\mathbf{X}) = \kappa(\mathbf{X})^2$, meaning any ill-conditioning in $\mathbf{X}$ will be squared when computing $(\mathbf{X}^T\mathbf{X})^{-1}$ [Saa03, p. 260]. The higher the condition number, the closer the matrix is to being singular [Lay16, p. 116]. Thinking back to the previous section (section 4.1), if $\mathbf{X}$ is close to ill-conditioned, it may be that $\mathbf{X}^T\mathbf{X}$ is outright singular. We conclude that solving the system in this manner has the potential to greatly enhance numerical issues. Changing the method for solving the LLS equations will not change the conditioning of $\mathbf{X}$, but we can avoid operating on $\mathbf{X}^T\mathbf{X}$ altogether. Consider decomposing $\mathbf{X}$ into its QR factorization,

$$\mathbf{X} = \mathbf{Q}\mathbf{R}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular and square [KC09, p. 280]. Then note,

$$\mathbf{X}^T\mathbf{X} = \mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{R} = \mathbf{R}^T\mathbf{R} \tag{15}$$

where the last equality holds by orthogonality; $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. Plugging this into the LLS equations, we get

$$\mathbf{R}^T\mathbf{R}\boldsymbol{\beta} = \mathbf{R}^T\mathbf{Q}^T\mathbf{y}.$$

If $\mathbf{X}$ has full rank, then $\mathbf{R}$ is non-singular [Cha87]. It follows that $\mathbf{R}^T$ is non-singular [Lay16, p. 107] and so we can premultiply the equation by $(\mathbf{R}^T)^{-1}$ to obtain

$$\mathbf{R}\boldsymbol{\beta} = \mathbf{Q}^T\mathbf{y}. \tag{16}$$

This final equation is what we solve in practice to obtain the OLS estimate given a QR decomposition. The point being of course that we avoid operating on the product that squares the condition number of the regressor matrix.

This should not be taken as an attempt at a formal argument that QR decomposition is more numerically stable than directly solving the LLS equations. Empirical validation showed this to be the case for stable history detection. Though, in general, it should serve to illustrate that choosing a suitable method is an important task. One that we will discuss next.

4.3 Discussion on related work

As we saw in section 4.2, the LLS equations may be solved directly via the inverse of the squared regressor matrix, $\mathbf{X}^T\mathbf{X}$, using e.g. Gauss-Jordan elimination. This is the approach taken by Gieseke et al. for the massively parallel monitoring part of BFAST because it is very amenable to parallelisation and as such makes for a very fast fit in Futhark [Gie+20]. Experimentation revealed that this strategy is however not viable when computing the stable history period.

For the monitoring part of BFAST, a linear model is fit once over the entire stable history period. This means sufficient data are available for the fit. In contrast, when recursive residuals are employed to detect a stable history period, each subset of the proposed history period is tested. This means linear models are fit with as little data as mathematically possible (k data points for a k parameter model) even when the proposed period is much larger. Typically, $\mathbf{X}$ has hundreds of rows, while the number of parameters, k, is small (less than 10). Selecting k data points amounts to dropping all but k rows from $\mathbf{X}$. Even though $\mathbf{X}$ is well-conditioned it may be that this smaller submatrix is not.

Using randomly generated data with missing values, designed to mimic real world data sets, and $k = 8$, linear dependencies appear when fitting models only with k or $k + 1$ data points. We will not try to make sense of this, but simply note that linear dependencies seem to be inherent to stable history detection only in the most extreme cases.

A Futhark library for fitting linear models using rank-revealing QR decomposition is detailed in appendix A along with two translations of FORTRAN routines related to rank-revealing QR-decomposition. We use this library in the implementation of stable history detection.

```
map (λy0 →
      let (n̄, X, y) = filtermissing X0 y0
      -- CUSUM process on reversed data
      let n = n̄ - k
      let r = recresid (reverse X) (reverse y)
      let mean = (reduce (+) 0 r) / n
      let sumd =  reduce (+) 0 (map (λa → (a - mean)**2) r)
      let sd   = sqrt (sumd / (n-1))
      let fr   = sd * (sqrt n)
      let sdized = map (λri → ri/fr) r
      let rcusum = scan (+) 0 sdized
      -- Structural change test
      let xs = map2 (λx i → abs (x / (1+2*i/n)) rcusum (1...n)
      let x  = reduce (max) (−∞) xs
      let pval  = f x -- f from equation (14)
      let level = f α
      let inds  = map2 (λi ri → if abs ri > α + (2*α*(i+1)/n)
                                then i
                                else ∞
                       ) (1...n) rcusum
      let ind = reduce (min) ∞ inds
      in if pval < level then n - ind else 0
    ) (ys: [m][N]f64)
```

Figure 7: Nested irregular parallel pseudocode for stable history detection using the ROC test. See algorithm 2 and section 3.2 for the sequential algorithm.

5 Data-parallel implementation

F
G H

This section details a data-parallel implementation of stable history detection when applied to satellite image data. Algorithm 2 applies to a single pixel (time series). Here we extend this to work on an image consisting of many pixels. Three approaches are weighed against each other, one is a simple approach that exploits only parallelism in the number of pixels (outer parallelism), while the others exploit also parallelism inherent to the computations for a single pixel (inner parallelism).

Satellite data can contain missing values; for example, clouds may obscure the ground at any given time and so data for this point in the pixel time series will be missing [Gie+20]. Missing values are represented by dummy values, e.g. floating point NANs. Missing data are ignored (filtered) in BFAST [VZH12, p. 5], which is easily handled in a sequential setting, but presents difficulties in a parallel setting when a program is to be statically mapped[17] for efficient execution on GPUs [Oan18]. Specifically, filtering missing values cause the length of time series to differ across pixels, which in turn is cause for nested irregular parallelism (not supported by the Futhark compiler). The implementation is greatly influenced by the handling of missing values for this reason.

Refer to section 2.1 for a primer on GPUs, Futhark and related terminology.

A data-parallel specification In figure 7 we show pseudocode for the data-parallel specification of algorithm 2 in a functional style. It takes as input an array `ys` contain-

[17] Read: the host code is to be written—or generated in the case of Futhark.

ing m time series of length N. It is assumed that the regressor matrix `X0` containing N rows of length k is part of the closure (precomputed or given elsewhere). Each row in `X0` is a pattern of k regressors as defined in equation (4). Inside the map, missing values in the timeseries are filtered in line 2 (the corresponding rows in `X0` are dropped). This step causes nested irregular parallelism in the operations that follow. There is very close correspondence with algorithm 2. Refer to section 3.2 for an explanation of each line.

5.1 Parallelisation strategy

The pseudocode in figure 7 exhibits nested irregular parallelism. As we saw in section 2.1 nested irregular parallelism is not supported by the Futhark compiler and we must either waste resources on idle threads or perform flattening transformations or forgo the inner (nested) parallelism by sequentialising the variable-length computations. In the following we discuss the strengths and weaknesses of each of these approaches.

(1) *Sequentialisation:* If we sequentialise all inner parallelism, we can map each pixel-specific computation to its own GPU thread. An advantage of this is that operations (SOACs) can be fused into sequential loops in a way that reduces the number of acceses to global memory [Gie+20]. Accessing global memory is up to two orders of magnitude slower than scalar operations [Gie+20, p. 5]. A second advantage is that computations inside the outer loop can operate on the actual length of the pixel time series (real world data sets may have a high frequency of missing values). Though the gains in this are limited as we cannot eschew the regularity imposed by the hardware; recall that GPU threads are divided into warps of some hardware-dependent size,[18] which execute their instructions in lock-step. If threads in the same warp have different-sized operations, for example loops with different counts, then all other threads will idle while the thread that executes the largest size (most iterations) finishes [Gie+20]. Only once all threads in the warp are done executing their instruction, will the warp of threads move on to the next.[19] For the same reasons, we cannot hide expensive computations behind branches (e.g. in figure 7 `ind` is only needed when `pval < level`); if one thread in the warp enters the branch, then all threads will have to execute it (discarding their result as needed) [Hen17, p. 55]. Lastly, hardware utilisation is highly dependent on the batch size. Small batches will not saturate GPU resources. This leads us to the next strategy which exploits inner parallelism.

(2) *Flattening:* The Futhark compiler does not support nested irregular parallelism, so we have to transform the program in order to exploit inner parallelism. One solution is to filter and flatten the data and then apply the flattening transformations that were briefly presented in section 2.1. While such solution can be systematically derived, it would be quite difficult to implement efficiently. In particular, the number of operations are *asymptotically* the same as before applying the transformations, but the space is not (an asymptotic increase in memory usage can be very limiting). Further, recall that the GPU programming interface dictates that threads must be grouped into blocks of some common size. If we filter missing values and subsequently flatten the data, each pixel time series may need to span one or more blocks. This limits the use of fast

[18] On NVIDIA GPUs this is 16 or 32.
[19] We call this *thread divergence*.

shared memory (which is private to each block), meaning global memory must be accessed more often.

(3) *Padding:* A final approach is to *pad* the filtered time series with dummy values until an upper bound on the inner size of (non-missing values of) all time series is reached. This regularizes the problem, eliminating the nested irregular parallelism arising from missing values. To operate on the smallest possible inner sizes, we then distribute the outer map around statements that operate on the same size. This is the approach taken in [Gie+20].[20] A downside is that we leave GPU threads idle on padded values. A great advantage is that we can now fit each time series within a single block; the maximum block size is 1024 for most NVIDIA GPUs and the length of the time series in BFAST are in the hundreds. This enables the use of fast shared memory for most, if not all intermediate arrays.

To summarize, we may (1) sequentialise all inner operations, (2) parallelise the inner operations through flattening transformations or (3) parallelise the inner operations, but work with regularized (padded) data. We will not consider approach (2) any further for the reasons outlined above. Next we show how approach (3) can be implemented.

5.2 Implementation in Futhark

To implement approach (3), we first filter each time series and then pad them to some upper bound on the length, $\overline{N}$. All remaining operations in figure 7, except the recursive residuals, will then have the same inner-parallel size of $n = \overline{N} - k$. The first step is thus to distribute the outer map around the filtering, recursive residuals and ROC test, respectively:

```
let (Ns, Xs, ys) = map (filtermissing X) ys0
let rs = map (λX y → recresid (reverse X) (reverse y)) Xs ys
in map (λy r N →
          let n = N - k
          let mean = (reduce (+) 0 r) / n
          ... --- rest is unchanged
        ) ys rs Ns
```

The implementation of `filtermissing` is given in appendix B. The map in the last line does not have to be distributed further, however the map over the recursive residuals does.

In figure 8 we give the data-parallel specification for the recusive residuals (algorithm 1). There are two things to note compared to algorithm 1. First, the loop is split into two; one where the numerical stability check is made unconditional and one where there is no check. Second, once the map has been distributed, it is interchanged inwards into the loop[21] (not shown here) and checking is made to stop only once *all* threads are finished lest it will be cause for nested irregular paralleism. This transformation means that the resulting program differs slightly from algorithm 1. For some time series, more checks than needed will be performed and since we do not discard the results of the model fit computed during this check, but we use these values in place of the update formula values, we have slightly altered the result. The

[20] A possible dummy value for floating point time series is the floating point NAN-value.

[21] It can be shown that in a perfect nest of loops, a parallel outer loop (a `map`) can always be moved inwards without violating data-dependencies [Oan18].

```
map (λy X →
      let n = N
      let (rank, M, β̂) = fit X[:k,:] y[:k]
      let (i, check)   = (k, true)
      let r = replicate 0 (n-k)
      -- Sequential loop with check
      while check && i < (n-1) do
        let (r[i-k], M, β̂) = loopbody i M β̂
        let (rank',  M, b) = fit X[:i+1,:] y[:i+1]
        let absdiff = map2 (λa b → abs (a-b)) b β̂
        let approx  = ((reduce (+) 0 absdiff) / k) <= tol
        let check   = !(rank == rank' == k && approx)
        let (i, rank, β̂) = (i+1, rank', b)
      -- Sequential loop without check
      while i < n do
        let (r[i-k], M, β̂) = loopbody i M β̂
        let i = i+1
      in r
    ) (ys: [m][N]f64) (Xs: [m][N][k])
```

(a) Recursive residuals in an outer map over the filtered array of pixels `ys` and the corresponding regressor matrices `Xs`.

```
let loopbody (i: i64) (M: [k][k]f64) (β̂: [k]f64) =
  -- Recursive residual
  let x = X[i,:]
  let d = matvecmul M x
  let f = 1 + (dotprod x d)
  let resid    = y[i] - (dotprod x β̂)
  let recresid = resid / (sqrt f)
  -- Update formulas
  let D = outerprod d d
  let M = map2 (map2 (λa b → a - b/fr)) M D
  let β̂ = map2 (+) β̂ (map (λa → (dotprod x a) * resid) M)
  in (recresid, M, β̂)
```

(b) Common loop body for recursive residuals. Assume that it is inlined in figure 8a and that `X` and `y` are part of the closure.

Figure 8: Nested irregular parallel pseudocode with in-place updates.

model fit is more precise so this will have no ill effect on the results. Other than this, we distribute the map over statements of the same inner size.

Operations `dotprod` and `matvecmul` used in figure 8 are straightforward to implement in Futhark. The dot product is a map-reduce:

```
let dotprod [n] (xs: [n]f64) (ys: [n]f64): f64 =
  reduce (+) 0 (map2 (*) xs ys)
```

And matrix-vector multiplication a map of this:

```
let matvecmul [n][m] (xss: [n][m]f64) (ys: [m]f64) =
  map (dotprod ys) xss
```

The `fit` function is OLS regression using a rank-revealing QR-decomposition, as detailed in section 4. The implementation is given in appendix A.

5.2.1 Incremental flattening Until now we have focused on how to parallelise stable history detection. However, with enough pixels, the data-parallelism will be in excess of what the hardware has to offer and it may be beneficial to instead efficiently sequentialise operations. Incremental flattening, as presented in section 2.1.3, is part of the Futhark compiler and we use it to both autotune the data-parallel map-distributed program and to generate an inner-sequential version, from the map-distributed version. Henceforth we call the program derived using approach (3) for *the map-distributed version* and the generated program adhering to approach (1) for the *inner-sequential version*.

	D1	D2	D3	D4	D5	D6	Peru	Sahara
m	16384	16384	32768	32768	65536	16384	111556	67858
n	512	256	256	128	128	256	113	114
f_{NAN}	50%	50%	50%	50%	50%	75%	69%	25%

Table 1: Data sets from [Gie+20]. The dimensions m and n denote the number of pixels and the length of the time series (proposed history period), respectively. While f_{NAN} denotes the frequency of missing (NAN) values.

6 Benchmarking

This section first evaluates the performance impact of the parallelisation strategies discussed in section 5, and identifies the parts of the implementation where optimisation efforts are best focused. After this, the performance impact that the addition of stable history detection has had on the BFAST implementation from [Gie+20] is briefly analysed.

Setup All benchmarks in this section are run on a machine with a 16-core Intel Xeon E5-2650 CPU running at 2.6 GHz, 126 GB of RAM and an NVIDIA RTX 2080 Ti GPU with 11GB of DRAM and 4352 cores running at 1.55 GHz. Table 1 shows the data sets used for benchmarking, which includes synthetic data sets D1–D6 and two real data sets of satellite imagery from Peru and the Saharan desert, respectively. The synthetic data sets are used so that we may control performance-critical features of the data. D1–D5 vary the number of pixels and length of each pixel time series, while keeping the number of missing values constant. D6 has a higher frequency of missing values.

6.1 The ROC test

The performance of stable history detection is shown on figure 9 where `map-distr` denotes the map-distributed version and `inner-seq` denotes the inner sequential version (see section 5.2.1). We report memory reads and writes performed by the program per second (in GB/s) in order to have a performance-measure that is normalised across data sets. The memory reads and writes are calculated as the actual memory accessed by the program. That is, we count every word accessed, multiply this by the size of a word and divide by the total runtime.

The map-distributed version is 1.3–2.5 times faster than the inner-sequential version, depending on the data set in question. But do note that the inner-sequential version is generated by Futhark from the map-distributed program. It should be possible to write a more efficient version in OpenCL by hand. Still, the performance gap illustrates shows that it is worth exploiting not only outer parallelism but also inner parallelism. The results corroborate the discussion of the two code versions from section 5.1, which serves to highlight why one version is faster than the other.

The performance is quite stable across all data sets. With D1 being the poorest performing; it has a low number of pixels and long time series compared to the others (so does D6, but it has many more missing values). The map-distributed version enjoys the greatest speedup on this data set, however. We also see that the performance on the synthetic data sets is comparable to the performance of the real world data sets.

We see that about half of the bandwidth provided by the hardware is utilised (percentages under each group of bars). For reference, a naive copy of memory from

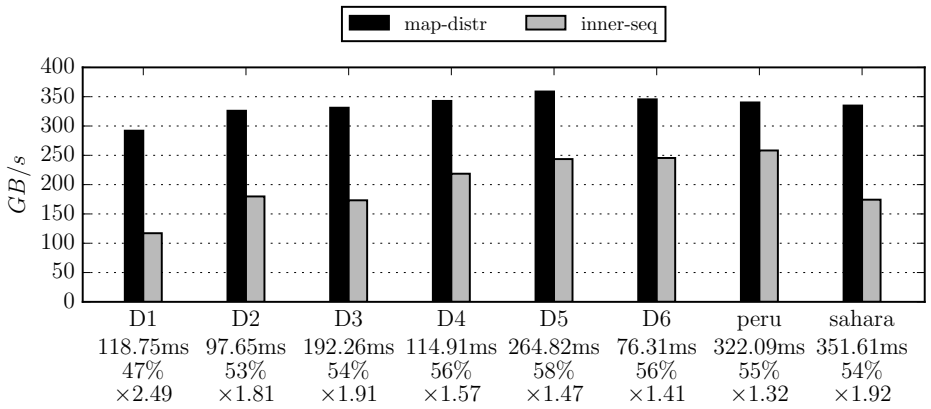

Figure 9: Memory reads and writes measured in GB per second. The map-distributed version is `map-distr` and the inner-sequential version is `inner-seq`. Under each data set, the runtime for `map-distr` is shown along with utilised percentage of hardware bandwidth, which is 616 GB/s for the GPU used here, and the speedup in runtime for `map-distr` relative to `inner-seq`.

		D1	D2	D3	D4	D5	D6
ROC test	(%)	81.8	87.3	85.7	87.3	88.8	89.6
recursive residuals	(%)	18.2	12.7	14.3	12.7	11.2	10.4

Table 2: Runtime distribution between the two main components of the map-distributed version of the stable history detection. The first row shows the total runtime of both components, while the two bottom rows show the percentage of total runtime. *ROC test* corresponds to figure 7 (minus call to `recresid`) while *recursive residuals* correspond to figure 8.

one location in global memory to another should be close to hardware bandwidth. The performance of the program is memory-bound; we perform only few floating point operations for each memory access. This suggests that the program has much room for optimisation.

The map-distributed version is two to three orders of magnitude faster than the R reference implementation when run on a single CPU core. This is hardly a fair comparison, but it should give an idea of the scale at which stable history detection has been accelerated through the work of this project.

On table 2 we decompose the runtime of the stable history detection into its two main components: the ROC test and the recursive residuals. In all cases, the recursive residuals are responsible for more than 80% of the total runtime. This suggests that optimisation efforts should be focused on the recursive residuals. Next we decompose the runtime of the recursive residuals to further pinpoint where the program spends most of its time.

6.2 Recursive residuals

The recursive residuals can be divided into two performance-critical parts. The loop, which carries out the update formulas and subsequently check these against a linear model fit using QR decomposition (`check-loop`), and the loop, which too carries the

		D1	D2	D3	D4	D5	D6
filter	(%)	4.0	2.7	2.8	2.7	2.2	2.4
no-check-loop	(%)	64.3	37.6	35.7	28.0	23.2	23.2
check-loop	(%)	35.7	62.4	64.3	72.0	76.8	76.8
number of checks		8/304	12/164	12/164	9/86	11/87	12/92

Table 3: Runtime distribution between components of the recursive residuals computation. Labels `no-check-loop`, `check-loop` and `filter` denote the sequential loop of update formulas, sequential loop of update formulas with checking against QR decomposition-based linear model fit and the filtering that puts valid values before missing values before missing values in the time series and each corresponding regressor matrix, respectively. The number of `check-loop` iterations out of the total number of iterations is shown on the bottom row.

update formulas, but without such a check (`no-check-loop`). From table 3 we see that for all data sets except D1, the runtime is dominated by `check-loop`. This is the first few iterations of the recursive residuals computation and is no more than 12 iterations across all data sets. Thus a very small number of iterations dominate the runtime. With the only difference between `no-check-loop` and `check-loop` being the model fitting, any optimisation efforts should be focused on the linear model fitting presented in section 4.

Referring back to figure 9 this may also serve to explain why the performance is so stable across data sets. Since the runtime is dominated only by a few number of iterations, the actual length of each time series has less of an impact.

6.3 Integration with BFAST-Monitor

Here, we briefly analyse the impact that the stable history detection has on the performance of BFAST when integrated with the implementation from [Gie+20]. We start by presenting three different code versions:

1. `bfast-fix` is the monitoring part of BFAST, where the start of the stable history period is fixed to time 0 for all pixels (time series). This corresponds with the implementation from [Gie+20] except that it has been converted from 32-bit floating point numbers to 64-bit floating point numbers.

2. `bfast-var` is identical to `bfast-fix` except that it takes an array of *precomputed* stable history start indices. This is to show whether a variable start of history has any effect on the performance prior to adding the actual stable history computation.

3. `bfast-ROC` is identical to `bfast-var` except that it does not take a precomputed array of stable history indices, but computes them using the map-distributed ROC test presented in this report.

Recall that detecting the start of the stable history period requires fitting up to $n-k$ models where n is the length of the pixel time series and k is the number of parameters, while the monitoring part has to fit only a single model per time series. As we saw in the previous section, the stable history detection will only perform up to 12 QR decomposition-based model fits and use update formulas for the rest on data sets D1–D6 (corresponding to the `check-loop` and `no-check-loop`, respectively). This is much less than the worst case $n-k$, but we still expect to see a considerable slowdown compared to the implementation given in [Gie+20].

	D1	D2	D3	D4	D5	D6	Peru	Sahara
`bfast-fix` (ms)	17.8	10.0	18.1	10.9	22.1	14.0	38.7	35.0
`bfast-var` (ms)	18.1	10.1	18.5	11.1	21.9	14.7	33.8	35.1
`bfast-ROC` (ms)	135.1	105.6	205.3	123.5	282.6	88.9	348.3	377.8
slowdown $\left(\frac{\text{roc-f64}}{\text{fix-f64}}\right)$	×7.6	×10.5	×11.3	×11.4	×12.8	×6.3	×9.0	×10.8

Table 4: This table shows the runtime in milliseconds for the three versions of BFAST presented in section 6.3. The array of indices fed to `bfast-variable` was precomputed using `bfast-ROC`.

On table 4, we see runtimes for each of the three versions. There are two things to note, (1) there is no significant slowdown in adding the precomputed variable start of history (`bfast-fix` and `bfast-var` are similar), but (2) including stable history detection (ROC test) slows it down by up to 13 times.

	D0	Peru	Sahara	Africa
m	5000	111556	67968	500000
N	100	235	228	327

Table 5: Data sets used in testing. Multiple synthetic data sets are used, but we only show one. The dimensions m and N denote the number of pixels and the length of the time series, respectively.

7 Validation

In this section, we validate the data-parallel implementations of recursive residuals and stable history detection against the R reference implementations.

Methodology and data We write a test suite in Python that calls both R and Futhark code.[22] We run both synthetically generated (random) data and three real world data sets through the test suite. The three real world data sets are satellite imagery from Peru, the Saharan desert and a small subset of a data set that contains the entire continental tropical Africa [Gie+20]. Their characteristics are shown on table 5.

7.1 The ROC test

The stable history detection outputs the index of the start of a stable history period. It validates with no differences when run a number of synthetic data sets. Nor are there any differences when run on the more than 500-thousand pixels in the Africa data set. There is however one difference in the Peru data set, and one difference in the Sahara data set. Both differences are caused by the CUSUM process crossing its boundary earlier than the R reference implementation (line 20 of algorithm 2). This is shown in table 6 where (part of) the CUSUM processes and boundaries for the two pixels in question are tabulated. The CUSUM processes for R and Futhark differ around the 12th decimal place (not apparent from the table data), while the boundaries differ around the 6th decimal place. In both cases, the Futhark boundary is too large, but more generally we can conclude that precision in the 6th decimal place is not sufficient to match the results of the R reference implementation. This causes the start of the stable history period to be respectively one and two time steps too early in the Futhark version. This should give an idea of how sensitive the ROC test is to numerics.

7.2 Recursive residuals

The recursive residuals are arrays of floating point numbers, so to validate these we compute a relative error measure. More precisely, for two arrays to be considered equal the following has to hold for each element:

$$abs(a - b) \leq tol_a + tol_r \cdot abs(b)$$

where a is the value subject to test, b is the reference value, $tol_a = 10^{-5}$ and $tol_r = 10^{-8}$. The tolerance tol_a is what we consider close to zero.

All real world data sets pass this test. But there are three differences in the synthetic data set D0. The differences are minor; the maximum relative error for the

[22] The Futhark code is called from Python using PyOpenCL (`https://documen.tician.de/pyopencl/`).

i	54	53 (R crosses)	52	51 (F. crosses)
R abs(cusum)	2.21110450	2.36514**512**	2.34888991	2.46709502
Futhark abs(cusum)	2.21110450	2.36514512	2.34888991	2.**46709502**
R boundary	2.35594091	2.36514**381**	2.37434670	2.38354960
Futhark boundary	2.35594294	2.36514584	2.37434874	2.**38355164**

(a) Part of a CUSUM test on a pixel in the Sahara data set. The reference implementation crosses (exceeds) the boundary at index 53 while our implementations cross the boundary at index 51.

i	28	27 (R crosses)	26 (F. crosses)	25
R abs(cusum)	2.21958123	2.33554**313**	2.38459770	2.41008640
Futhark abs(cusum)	2.21958123	2.33554313	2.38459**770**	2.41008640
R boundary	2.31599845	2.33554**275**	2.35508704	2.37463133
Futhark boundary	2.31600045	2.33554475	2.35508**906**	2.37463337

(b) Part of a CUSUM test on a pixel in the Peru data set. The reference implementation crosses (exceeds) the boundary at index 27 while our implementations cross the boundary at index 26.

Table 6: Tabular representation of two CUSUM tests (refer to figure 4 for a visual example on different data). Note that the CUSUM test is computed on the reversed input data hence the indices are in reversed order. The table should be read from left to right and numbers should be compared within columns. The points of interest are highlighted in bold.

recursive residuals of all three pixels is about 0.0725%. In all three cases the difference occurs in the first recursive residual computed, which is when the computation is at its most unstable and a k-parameter model is fit using only k data points. The difference does not have to do with the fit, however. Instead it is introduced in what amounts to line 5 of algorithm 1. No further investigation was done.

7.3 OLS methods

In section 4, it was claimed that a rank-revealing QR decomposition is necessary when fitting linear models in the computation of the recursive residuals. To quantify this statement, *if we solve the linear least squares equations directly using Gauss-Jordan elimination*: 67 out of 5000 pixels in D0 differ from the reference implementation. In the Sahara data set 13573 out of 67968 pixels differ. However, there are no differences for data sets Peru and Africa. The same does not hold for the ROC test, where Gauss-Jordan elimination will cause differences across all data sets.

8 Conclusion

To conclude, stable history detection has been implemented to run on massively parallel hardware (GPUs), thus further enabling large scale detection of environmental disturbances in satellite image data. The implementation is, however, not restricted to such scenario and can be applied to detect structural change in any time series data.

The theory needed to implement stable history detection has been explained, a proof has been given for the discretisation of key formulas, and two algorithms have been formulated; one for the recursive residuals and one for the ROC test that together constitute stable history detection.

Benchmarking of parallelisation strategies suggest that (1) exploiting parallelism not only in the number of pixels but also the parallelism inherent to each pixel time series computation yields a significant speedup in practice (up to 2.5 times for the tested data sets and GPU), and (2) that the solution given here only achieves about half of theoretical bandwidth, meaning there is room for optimisation. Still, the implementation is two to three orders of magnitude faster than existing publicly available code (the R reference implementation) when processing a batch of pixels.

Validation revealed that stable history detection is sensitive to numerics and that advanced numerical methods for linear regression are necessary to achieve a sufficiently accurate implementation.

The data-parallel implementation of stable history detection in Futhark, an equivalent sequential Python implementation and the test suite for validation are all available at `https://github.com/nhey/recresid-roc`. The Futhark library for fitting linear models using a rank-revealing QR decomposition is available at `https://github.com/nhey/lm`. The integration of stable history detection with BFAST is available at `https://github.com/nhey/bfast`.

Future work The R reference implementation offers many features not considered in this project. These features can most likely be parallelised through similar techniques as those presented in [Gie+20] and sustained here.

This project revealed a need for Futhark implementations of advanced factorizations from linear algebra, where the focus is as much on numerical stability as it is on high-performance. The implementation of rank-revealing QR decomposition, given in this report, is written in a sequential style and the benchmarking results should benefit greatly from this being parallelised.

Finally, there are likely performance optimisations to be had in the data-parallel implementations given in the repositories linked to above; the implementation part of this project has been focused on strict validation against the R reference implementation while showing different parallelisation strategies, and not much effort has gone into fine-tuning the resulting implementations.

References

[AW20] Josh Alman and Virginia Vassilevska Williams. *A Refined Laser Method and Faster Matrix Multiplication*. 2020. arXiv: 2010.05846 [cs.DS].

[BDE75] R. L. Brown, J. Durbin and J. M. Evans. "Techniques for Testing the Constancy of Regression Relationships over Time". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 37.2 (1975), pp. 149–192. ISSN: 00359246. URL: http://www.jstor.org/stable/2984889.

[Ble96] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (Mar. 1996), pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: https://doi.org/10.1145/227234.227246.

[Cha87] Tony F. Chan. "Rank revealing QR factorizations". In: *Linear Algebra and its Applications* 88-89 (1987), pp. 67–82. ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(87)90103-0. URL: https://www.sciencedirect.com/science/article/pii/0024379587901030.

[EHO18] Martin Elsman, Troels Henriksen and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2018. URL: https://futhark-book.readthedocs.io.

[GH84] Jacqueline S. Galpin and Douglas M. Hawkins. "The Use of Recursive Residuals in Checking Model Fit in Linear Regression". In: *The American Statistician* 38.2 (1984), pp. 94–105. ISSN: 00031305. URL: http://www.jstor.org/stable/2683242.

[Gie+20] F. Gieseke et al. "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020, pp. 385–396. DOI: 10.1109/ICDE48307.2020.00040.

[Ham+20] Eliakim Hamunyela et al. "Implementation of BFASTmonitor Algorithm on Google Earth Engine to Support Large-Area and Sub-Annual Change Monitoring Using Earth Observation Data". In: *Remote Sensing* 12 (Sept. 2020), p. 2953. DOI: 10.3390/rs12182953.

[Han01] Bruce E. Hansen. "The New Econometrics of Structural Change: Dating Breaks in U.S. Labour Productivity". In: *Journal of Economic Perspectives* 15.4 (Dec. 2001), pp. 117–128. DOI: 10.1257/jep.15.4.117. URL: https://www.aeaweb.org/articles?id=10.1257/jep.15.4.117.

[Hay00] Fumio Hayashi. *Econometrics*. Princeton Univ. Press, 2000. ISBN: 0691010188.

[Hen+19] Troels Henriksen et al. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 53–67. ISBN: 9781450362252. DOI: 10.1145/3293883.3295707. URL: https://doi.org/10.1145/3293883.3295707.

[Hen17] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[KC09] David R. Kincaid and E. Ward Cheney. *Numerical analysis: mathematics of scientific computing*. 3rd ed. Pacific Grove, Calif.: Brooks/Cole, 2009. ISBN: 9780821847886.

[Kre06] Erwin Kreyszig. *Advanced engineering mathematics*. eng. 9th. Hoboken, NJ: Wiley, 2006. ISBN: 0471728977.

[Lay16] D.C. Lay. *Linear Algebra and Its Applications*. Pearson Education, 2016. ISBN: 9780321982384.

[Mac95] George Mackiw. "A Note on the Equality of the Column and Row Rank of a Matrix". In: *Mathematics Magazine* 68.4 (1995), pp. 285–286. DOI: 10.1080/0025570X.1995.11996337.

[Meh+18] Malte von Mehren et al. *Massively-Parallel Break Detection for Satellite Data*. 2018. arXiv: 1807.01751 [cs.DC].

[Mol04] C.B. Moler. *Numerical Computing with MATLAB*. Other titles in applied mathematics. Society for Industrial and Applied Mathematics, 2004. ISBN: 9780898715606. URL: https://se.mathworks.com/content/dam/mathworks/mathworks-dot-com/moler/leastsquares.pdf.

[Oan18] Cosmin E. Oancea. *Lecture Notes for the Software Track of the PMPH Course*. 2018.

[OB20] Sven Otto and Jörg Breitung. *Backward CUSUM for Testing and Monitoring Structural Change*. 2020. arXiv: 2003.02682 [econ.EM].

[PT02] Hashem Pesaran and Allan Timmermann. "Market Timing and Return Prediction under Model Instability". In: *Journal of Empirical Finance* 9 (Apr. 2002), pp. 495–510. DOI: 10.1016/S0927-5398(02)00007-5.

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020. URL: https://www.R-project.org/.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. USA: Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342.

[VZH12] Jan Verbesselt, Achim Zeileis and Martin Herold. "Near real-time disturbance detection using satellite image time series". In: *Remote Sensing of Environment* 123 (2012), pp. 98–108. ISSN: 0034-4257. DOI: https://doi.org/10.1016/j.rse.2012.02.022. URL: https://www.sciencedirect.com/science/article/pii/S0034425712001150.

[Zei+02] Achim Zeileis et al. "Strucchange: An R Package for Testing for Structural Change in Linear Regression Models". In: *Journal of Statistical Software* 7 (July 2002). DOI: 10.18637/jss.v007.i02.

A From FORTRAN to Futhark: linear models

Advanced implementations of QR decomposition exists in both R and numpy. In both cases these implementations are merely wrapped calls to underlying FORTRAN routines (famously LAPACK and LINPACK). To implement rank-revealing QR decomposition with limited time available, it seemed most reasonable to replicate one of these FORTRAN routines.

The R language uses a modified version of LINPACKs pivoting QR decomposition, `dqrdc`, for fitting models with the `lm` package. It is aptly named `dqrdc2`.[23] As part of

[23] https://github.com/wch/r-source/blob/79298c499218846d14500255efd622b5021c10ec/src/appl/dqrdc2.f

this project `dqrdc2` was translated from FORTRAN to both Futhark and Python.[24] The Futhark version, being the most interesting, is detailed in section A.2.

Another LINPACK routine called `dqrqty` was translated from FORTRAN. Given the output from `dqrdc2` and a vector $\mathbf{y}$, this will compute $\mathbf{Q}^T\mathbf{y}$ where $\mathbf{Q}$ is from the QR decomposition of a matrix $\mathbf{X}$. It is necessary to translate this, because `dqrdc2` does not directly output the QR decomposition, but rather outputs something from which $\mathbf{Q}$ and $\mathbf{R}$ can be constructed. See section A.2 for the Futhark source code.

The two translations are organised in a Futhark module that we call `linpack_d`—we shall see this in action in the next section.

A.1 Implementing robust linear model fitting in Futhark

This section highlights details necessary to understand the implementation of linear model fitting using rank-revealing QR decomposition in Futhark. Refer to section 2.1 for an introduction to Futhark.

We structure the implementation in a Futhark module on figure 10 to keep it decoupled from the stable history detection code. The module is set to work with 64-bit floating point numbers only in lines 1 to 9. This is limitation is due to the translation of FORTRAN routine `dqrdc2` from the `linpack_d` module.[25] We could alter `dqrdc2` to work with 32-bit floating point numbers, but this is not straightforward as two key tolerance values have to be derived for the 32-bit version.[26]

For convenience, the module then defines what a regression result is in line 11 along with auxiliary functions `dotprod_nan`, `back_substitution` and `identity` in lines 13 to 25, which are a dot product that ignores nan values (treats them as zeros), back substitution for solving upper triangular systems of equations (see [KC09, p. 150]) and a function that generates an identity matrix, respectively.

The interesting part is the `fit` function in line 40, which given an $p \times n$ transposed regressor matrix $\mathbf{X}^T$ and an $n \times 1$ vector of target values, $\mathbf{y}$, will produce the OLS estimate of the regression parameters, $\hat{\boldsymbol{\beta}}$, the covariance parameters, $(\mathbf{X}^T\mathbf{X})^{-1}$, and the rank of $\mathbf{X}$. A rank-revealing QR decomposition, `dqrdc2`, is available to us in the `linpack_d` module from appendix A. From this we obtain the upper triangular matrix $\mathbf{R}$ from the QR decomposition of $\mathbf{X}$ along with the rank of $\mathbf{X}$ (lines 41 and 42):

```
let (A', pivot, qraux, rank) = linpack.dqrdc2 (copy X') 1e-7
let R = transpose (lower_triangular_nan A'[:p,:p])
```

Here, `A'` contains $\mathbf{R}^T$ below the main diagonal.[27] To extract $\mathbf{R}$ we first need to know its dimension, k. Since $\mathbf{X} = \mathbf{QR}$, we know that the shared dimension of $\mathbf{Q}$ and $\mathbf{R}$ is $k = \min(n, p, r)$ where r is the rank of $\mathbf{X}$. Fitting a linear regression with p parameters requires at least p data points, so we always have $n \geq p$. Moreover, the rank of $\mathbf{X}$ is at most p. We conclude that $k = r \leq p$. However, if we want `fit` to be mappable in Futhark it must be regular (see section 5). Since the rank of $\mathbf{X}$ may vary, we cannot use this value to directly slice `A'` lest the result will be cause for irregular parallelism. The first step in determining $\mathbf{R}$ from the output is therefore to slice `A'[:p,:p]` and then set the values above the diagonal *determined by the rank* to floating point NAN. This is done in the `lower_triangular_nan` function in line 32.

[24] The Python programming language, `https://www.python.org/`

[25] in LINPACK a d-prefix denotes double—or 64-bit—precision

[26] Attempts at this were made, but a good balance between accuracy and performance, which is ultimately what these tolerance values adjust, was not immediately found.

[27] As to why this is, consult the original FORTRAN source code. This style of output makes more sense in low-level imperative languages that expose memory pointers to the programmer, such as FORTRAN or C.

```
module lm_f64 = {
  import "../../diku-dk/linalg/linalg"
  import "linpack"

  module T = f64
  type real = T.t

  module linalg = mk_linalg T
  module linpack = linpack_d

  type results [p] = { params: [p]real, cov_params: [p][p]real, rank: i64 }

  let dotprod_nan [n] (xs: [n]real) (ys: [n]real): real =
    reduce (+) 0 (map2 (\x y -> if T.isnan y then 0 else x*y) xs ys)

  let back_substitution [n] (U: [n][n]real) (y: [n]real): [n]real =
    let x = replicate n (T.i64 0)
    in loop x for j in 0..<n do
      let i = n - j - 1
      let sumx = dotprod_nan x[i+1:n] U[i,i+1:n]
      let x[i] = (y[i] T.- sumx) T./ U[i,i]
      in x

  let identity (n: i64): [n][n]real =
    tabulate_2d n n (\i j ->if j == i then T.i64 1 else T.i64 0)

  let chol2inv [n] (U: [n][n]real): ([n][n]real, [n][n]real) =
    let UinvT = map (back_substitution U) (identity n)
    let Uinv = transpose UinvT
    in (linalg.matmul Uinv UinvT, Uinv)

  let lower_triangular_nan [m][n] (rank: i64) (L: [m][n]real): [m][n]real =
    map2 (\j ->
           map2 (\i ele -> if i+1 > rank || j+1 > rank
                           then T.nan
                           else ele
                ) (iota n)
         ) (iota m) L

  let fit [n][p] (X': [p][n]real) (y: [n]real): results [p] =
    let (A', pivot, qraux, rank) = linpack.dqrdc2 (copy X') 1e-7
    let R = transpose (lower_triangular_nan rank A'[:p,:p])
    let (cov_params, Rinv) = chol2inv R
    let Q'y = linpack.dqrqty A' qraux rank y
    let Q'y = Q'y[:p]
    let beta = linalg.matvecmul_row Rinv Q'y
    let beta = scatter (replicate p (T.i64 0)) pivot beta
    let pp = p*p
    let pivot_2d = map (\i -> map (\j -> (i,j)) pivot) pivot
                   |> flatten :> [pp](i64,i64)
    let cov_params = scatter_2d (replicate pp (T.i64 0) |> unflatten p p)
                                pivot_2d (cov_params |> flatten :> [pp]real)
    let cov_params = map (map (\x -> if T.isnan x then 0 else x)) cov_params
    let beta = map (\x -> if T.isnan x then 0 else x) beta
    in { params = beta, cov_params = cov_params, rank = rank }
}
```

Figure 10: A futhark module for fitting linear models that pose ill-conditioned problems.

Next, we compute the covariance parameters. Equation (15) reveals that $\mathbf{X}^T\mathbf{X} = \mathbf{R}^T\mathbf{R}$ and we use this result in line 43:

```
let (cov_params, Rinv) = chol2inv R
```

The function `chol2inv` computes $(\mathbf{U}^T\mathbf{U})^{-1}$ given an upper triangular matrix $\mathbf{U}$:

```
let chol2inv [n] (U: [n][n]real): ([n][n]real, [n][n]real) =
  let UinvT = map (back_substitution U) (identity n)
  let Uinv = transpose UinvT
  in (linalg.matmul Uinv UinvT, Uinv)
```

In the two first lines of the body, $\mathbf{U}$ is inverted by solving the linear system $\mathbf{UH} = \mathbf{I}$ for $\mathbf{H} = \mathbf{U}^{-1}$. This is done using back substitution because $\mathbf{U}$ is upper triangular. In the third and final line $(\mathbf{U}^T\mathbf{U})^{-1}$ is computed. To see this, note that

$$(\mathbf{U}^T\mathbf{U})^{-1} = \mathbf{U}^{-1}(\mathbf{U}^T)^{-1} = \mathbf{U}^{-1}(\mathbf{U}^{-1})^T.$$

In our case, the intermediate result $\mathbf{U}^{-1} = \mathbf{R}^{-1}$ is usable later. The reader may recognize `chol2inv` as the standard procedure used to invert a matrix given its Cholesky decomposition, albeit transposed.[28]

We then compute the regression parameters by solving equation (16):

$$\mathbf{R}\boldsymbol{\beta} = \mathbf{Q}^T\mathbf{y}$$

for $\boldsymbol{\beta}$ to obtain the OLS estimate. Traditionally this would, once again, be done using back-substitution because $\mathbf{R}$ is upper triangular [KC09, p. 150]. But since we need to compute $\mathbf{R}^{-1}$ regardless and back-substituion involves a number of sequential steps, it is preferable to premultiply both sides with $\mathbf{R}^{-1}$ in a data-parallel setting (lines 44 to 46):

```
let Q'y = linpack.dqrqty A' qraux rank y
let Q'y = Q'y[:p]
let beta = linalg.matvecmul_row Rinv Q'y
```

Here, we use another translation of a LINPACK FORTRAN routine, `dqrqty`, to compute the right hand side. To reiterate, given `y` along with the previously computed `A'` and `qraux`, which contain the information needed to construct $\mathbf{Q}$ from the QR decomposition of $\mathbf{X}$, it will output $\mathbf{Q}^T\mathbf{y}$. See section A.2 for the Futhark translation of the FORTRAN source code.

A final implementation detail is that the translated LINPACK routines make use of pivoting to improve numerical stability. Therefore, we pivot each result back to its original configuration in lines 47 to 52. Lastly, we rid the results of internal padding using NAN values in lines 53 and 54.

A.2 Translation of LINPACK QR-decomposition

The algorithm used in `dqrdc2` is based on Householder reflections, but modified to also reveal the rank of the input matrix. Luckily the source code is endowed with a few explanatory comments.[29]

[28] The name `chol2inv` comes from similar functionality in the R language. In particular we note that if fed transpose of $\mathbf{L}$ from Cholesky decomposition $\mathbf{A} = \mathbf{LL}^T$, the result is $\mathbf{A}^{-1}$.

[29] The FORTRAN source code also includes a few historic comments. A particularly enjoyable one is given as part of the program preface: *i am very nervous about modifying linpack code in this way. if you are a computational linear algebra guru and you really understand how to solve this problem please feel free to suggest improvements to this code.* The comment seems to have been written in 1995. Though the routine was last modified in 2019.

FORTRAN ordering makes use of row-major indexing, but has a column-major layout in memory. Therefore the Futhark version of `dqrdc2` operates on arrays, transposed relative to the original FORTRAN source program.

The routine takes as input a matrix `x` whose decomposition is to be computed along with a tolerance value, `tol`, used when determining the rank of `x`. We begin by computing the norms of the columns of `x`:

```
module T = f64
type t = f64

local let dotprod xs ys : t =
  T.(reduce (+) (i64 0) (map2 (*) xs ys))

let dnrm2 xs =
  T.sqrt (dotprod xs xs)

let dqrdc2 [n][p] (x: *[p][n]t) (tol: t): ([p][n]t, [p]i64, [p]t, i64) =
  let qraux = replicate p (T.i64 0)
  let work = replicate (2*p) (T.i64 0) |> unflatten 2 p
  let jpvt = iota p
  let (qraux, work) =
    loop (qraux, work) for j < p do
      let nrm = dnrm2 x[j,:]
      let qraux[j] = nrm
      let work[0,j] = nrm
      let work[1,j] = if T.(nrm == i64 0) then T.i64 1 else nrm
      in (qraux, work)
```

We then enter the loop that will perform Householder reflections. Each iteration is initiated by pivoting the columns of `x` so that those with non-negligible norm come first. This is the rank-revealing operation. In code:

```
let lup = i64.min n p
let k = p + 1
let (x, jpvt, qraux, _, k) =
  loop (x, jpvt, qraux, work, k) for l < lup do
    let (x, jpvt, qraux, work, k) =
      loop (x, jpvt, qraux, work, k) = (x, jpvt, qraux, work, k)
        while (l+1 < k) && T.(qraux[l] < work[1,l] * tol) do
          let lp1 = l + 1
          let x =
            loop (x) for i < n do
              let t = x[l,i]
              let x =
                loop (x) for j0 < (p - lp1) do
                  let j = j0 + lp1
                  let x[j-1,i] = x[j,i]
                  in  x
              let x[p-1, i] = t
              in  x
          let i = jpvt[l]
          let t  = qraux[l]
          let tt = work[0,l]
          let ttt = work[1,l]

          let (jpvt,qraux,work) =
            loop (jpvt,qraux,work)
```

```
        for j0 < (p - lp1) do
          let j = j0 + lp1
          let jpvt[j-1] = jpvt[j]
          let qraux[j-1] = qraux[j]
          let work[0,j-1] = work[0,j]
          let work[1,j-1] = work[1,j]
          in  (jpvt,qraux,work)

      let jpvt[p-1] = i
      let qraux[p-1] = t
      let work[0,p-1] = tt
      let work[1,p-1] = ttt
      let k = k - 1
      in  (x, jpvt, qraux, work, k)
```

The actual Householder reflection on the current column follows suit.

```
let (qraux, work, x) =
  if l+1 == n
  then (qraux, work, x)
  else let nrmxl = dnrm2 x[l,l:]
       in if T.(nrmxl == i64 0)
          then (qraux, work, x)
          else let nrmxl = if T.(x[l,l] != i64 0)
                           then dsign nrmxl x[l,l]
                           else nrmxl
               let x =
                 loop (x) for i0 < n - l do
                   let i = i0 + l
                   in x with [l,i] = x[l,i] / nrmxl
               let x[l,l] = T.(i64 1 + x[l,l])
               let (qraux, work, x) =
                 loop (qraux, work, x) for j in (l+1..<p) do
                   let t =
                     loop (t) = (T.i64 0) for i0 < n - l do
                       let i = i0 + l
                       in t - x[l,i] * x[j,i]
                   let t = t / x[l,l]
                   let x =
                     loop (x) for i0 < n - l do
                       let i = i0 + l
                       in x with [j,i] = x[j,i] + t * x[l,i]
                   in if T.(qraux[j] == i64 0)
                      then (qraux, work, x)
                      else let tt = T.(((abs x[j,l])/qraux[j]) ** i64 2)
                           let tt = T.(i64 1 - tt)
                           let tt = T.(max tt (i64 0))
                           let t = tt
                           in if T.(abs t >= f64 1e-6)
                              then let qraux[j] = T.(qraux[j] * sqrt t)
                                   in (qraux, work, x)
                              else let qraux[j] = dnrm2 x[j,l+1:]
                                   let work[0,j] = qraux[j]
                                   in (qraux, work, x)
               let qraux[l] = x[l,l]
               let x[l,l] = T.(i64 (-1) * nrmxl)
               in (qraux, work, x)
```

```
   in (x, jpvt, qraux, work, k)
let k = i64.min (k-1) n
in (x, jpvt, qraux, k)
```

where `dsign` sets the sign of its first argument `a` to match that of its second argument `b`:

```
local let dsign a b = T.((sgn b) * (abs a))
```

This operation is undefined when `b` is zero.

Another LINPACK routine called `dqrqty` was translated from FORTRAN. Given the output from `dqrdc2` and a vector $\mathbf{y}$, the function `dqrqty` computes $\mathbf{Q}^T\mathbf{y}$ where $\mathbf{Q}$ is from the QR decomposition of a matrix $\mathbf{X}$. The code follows without further explanation:

```
module T = f64
type t = f64

-- Compute `Q'y` given output from `dqrdc2` (`Q` is reconstructed
-- from `x` and `qraux`).
let dqrqty [n][p] (x: [p][n]t) (qraux: [p]t) (k: i64) (y: [n]t) =
  let ju = i64.min k (n-1)
  in loop (qty) = (copy y) for j < ju do
    if qraux[j] T.== (T.i64 0)
    then qty
    else let t = - qraux[j] * qty[j]
         let t =
           loop (t) for i0 < (n - j - 1) do
             let i = i0 + j + 1
             in t - x[j,i] * qty[i]
         let t = t / qraux[j]
         let qty[j] = qty[j] + t * qraux[j]
         let qty =
           loop (qty) for i0 < (n - j - 1) do
             let i = i0 + j + 1
             in qty with [i] = qty[i] + t * x[j,i]
         in qty
```

B Filtering missing values

Generally, data sets represent missing values using floating point NAN values. This is a simple way to store time series with missing values; if we did not use NAN values, we would have to keep auxiliary arrays that map each value to their respective time step or use some similar data structure. We therefore "filter" an array containing missing values by moving all non-NAN values to the start of the array (in order) and padding the rest of the array with NANs:

```
let filterPadWithKeys [n] 't
           (p : (t -> bool))
           (dummy : t)
           (arr : [n]t) : (i64, [n]t, [n]i64) =
  let tfs = map (\a -> if p a then 1i64 else 0i64) arr
  let isT = scan (+) 0i64 tfs
  let i   = last isT
  let inds= map2 (\a iT -> if p a then iT - 1 else -1i64) arr isT
  let rs  = scatter (replicate n dummy) inds arr
```

```
  let ks  = scatter (replicate n (-1i64)) inds (iota n)
  in (i, rs, ks)

let filter_nan_pad = filterPadWithKeys ((!) <-< f64.isnan) f64.nan
```

The function `(<-<)` is function composition. The implementation is from [Gie+20]. The function `scatter` facilitates in-place updates. A short description is provided:

```
val scatter 't [m] [n]: (dest: *[m]t) -> (is: [n]i64) -> (vs: [n]t) -> *[m]t
```

Using indices found in `is` and values found in `vs`, update `dest` *in-place* like so: `for i in 0...n-1: dest[is[i]] = vs[i]`. Note that the size of `dest` does not have to equal `n`. Further, any index value less than zero in `is` is ignored.