Functional Numerical Methods in Clojure

An Exploration of Overkill

Sam Ritchie

2020-05-26

Contents

1 Introduction

This document is an exploration of the `definite-integral` implementation in SICMUtils, a port of the wonderful `scmutils` system.

1.1 The Interface

This interface is the target. The goal of the document is to make this `definite-integral` interface come to life.

```
(ns quadrature
  (:require [sicmutils.numerical.compile :as c]
            [quadrature.adaptive :as qa]
            [quadrature.boole :as boole]
            [quadrature.common :as qc]
            [quadrature.bulirsch-stoer :as bs]
```

```
            [quadrature.infinite :as qi]
            [quadrature.midpoint :as mid]
            [quadrature.milne :as milne]
            [quadrature.riemann :as riemann]
            [quadrature.romberg :as romberg]
            [quadrature.simpson :as simp]
            [quadrature.simpson38 :as simp38]
            [quadrature.trapezoid :as trap]
            [sicmutils.util :as u]))
```

This namespace unites all of the work inside `quadrature` behind a single interface, fronted by the all-powerful `definite-integral` function.

The interface takes `f`, an integrand, along with bounds `a` and `b`:

```
(definite-integral f a b)
```

Optionally, you can provide a dictionary of customizing options. These are passed down to whatever method you supply via the `:method` key.

```
(definite-integral f a b opts)
```

1.1.1 Implementation

The keys in `quad-methods` below define the full range of integration methods available in the package. Each entry in this dictionary is either:

- An 'integrator' function that matches the interface above for `definite-integral` (possibly created with `qc/defintegrator`)
- a dictionary of extra options. This must contain a `:method` key.

This latter style is used when the method itself is a specialization of a more general method.

```
(def ^:private quadrature-methods
  {:open                    {:method :adaptive-bulirsch-stoer
                             :interval qc/open}
   :closed                  {:method :adaptive-bulirsch-stoer
                             :interval qc/closed}
   :closed-open             {:method :adaptive-bulirsch-stoer
                             :interval qc/closed-open}
   :open-closed             {:method :adaptive-bulirsch-stoer
```

```
                            :interval qc/open-closed}
   :bulirsch-stoer-open     bs/open-integral
   :bulirsch-stoer-closed   bs/closed-integral
   :adaptive-bulirsch-stoer (qa/adaptive bs/open-integral bs/closed-integral)
   :left-riemann            riemann/left-integral
   :right-riemann           riemann/right-integral
   :lower-riemann           riemann/lower-integral
   :upper-riemann           riemann/upper-integral
   :midpoint                mid/integral
   :trapezoid               trap/integral
   :boole                   boole/integral
   :milne                   milne/integral
   :simpson                 simp/integral
   :simpson38               simp38/integral
   :romberg                 romberg/closed-integral
   :romberg-open            romberg/open-integral})

(def available-methods
  (into #{} (keys quadrature-methods)))
```

The user can specify a method by providing the `:method` key in their options with:

- a key in the above dictionary
- another dictionary
- a custom integration function

The latter two are the allowed value types in `quadrature-methods`.

```
(defn- extract-method
  "Attempts to turn the supplied argument into an integration method; returns nil
  if method doesn't exist."
  [method]
  (cond (fn? method)
        [method {}]

        (keyword? method)
        (extract-method
         (quadrature-methods method))
```

```
        (map? method)
        (let [[f m] (extract-method
                     (:method method))]
          [f (merge (dissoc method :method) m)])))

(defn get-integrator
  "Takes:

  - An integration method, specified as either:
    - a keyword naming one of the available methods in `available-methods`
    - a function with the proper integrator signature
    - a dictionary of integrator options with a `:method` key

  - `a` and `b` integration endpoints
  - an optional dictionary of options `m`

  And returns a pair of an integrator function and a possibly-enhanced options
  dictionary.

  (Some integration functions require extra options, so the returned dictionary
  may have more entries than the `m` you pass in.)

  If either endpoint is infinite, the returned integrator is wrapped in
  `qi/improper` and able to handle infinite endpoints (as well as non-infinite
  endpoints by passing through directly to the underlying integrator)."
  ([method a b] (get-integrator method a b {}))
  ([method a b m]
   (when-let [[integrate opts] (extract-method method)]
     (let [integrate (if (or (qc/infinite? a)
                             (qc/infinite? b))
                       (qi/improper integrate)
                       integrate)]
       [integrate (dissoc (merge opts m) :method)]))))
```

1.1.2 Final API

Here we are! The one function you need care about if you're interested in definite integrals. Learn to use this, and then dig in to the details of individual methods if you run inton trouble or want to learn more. Enjoy!

```
(defn definite-integral
  "Evaluates the definite integral of integrand ‘f‘ across the interval $a, b$.
  Optionally accepts a dictionary ‘opts‘ of customizing options; All ‘opts‘ will
  be passed through to the supplied ‘integrate‘ functions.

  If you’d like more control, or to retrieve the integration function directly
  without looking it up via ‘:method‘ each time, see ‘get-integrator‘.

  All supplied options are passed through to the underlying integrator; see the
  specific integrator for information on what options are available.

  ## Keyword arguments:

  ‘:method‘: Specifies the integration method used. Must be

  - a keyword naming one of the available methods in ‘available-methods‘
  - a function with the proper integrator signature
  - a dictionary of integrator options with a ‘:method‘ key

  Defaults to ‘:open‘, which specifies an adaptive bulirsch-stoer quadrature method.

  ‘:compile?‘ If true, the generic function will be simplified and compiled
  before execution. (Clojure only for now.) Defaults to false.

  ‘:info?‘ If true, ‘definite-integral‘ will return a map of integration
  information returned by the underlying integrator. Else, returns an estimate
  of the definite integral."
  ([f a b] (definite-integral f a b {}))
  ([f a b {:keys [method compile? info?]
           :or {method :open
                compile? false
                info? false}
           :as opts}]
   (if-let [[integrate m] (get-integrator method a b opts)]
     (let [f      #?(:clj (if compile? (c/compile-univariate-fn f) f)
                     :cljs f)
           result (integrate f a b m)]
       (if info? result (:result result)))
     (u/illegal (str "Unknown method: " method
                     ". Try one of: "
```

```
                    available-methods)))))
```

2 Basics

2.1 Riemann Sums

```
(ns quadrature.riemann
  (:require [quadrature.interpolate.richardson :as ir]
            [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [sicmutils.generic :as g]
            [sicmutils.util :as u]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]
            [sicmutils.numsymb]))
```

This namespace includes functions for calculating the Riemann integral of a single-variable function. These are probably *not* methods that you'll want to use; see the documentation and defaults in `quadrature` for good recommendations. But they're clear and understandable. The goal of this namespace is to lay the groundwork for visualizable routines that you can use to step toward understanding of the tougher methods.

"Quadrature", in this context, means "numerical integration". The word is a historical term for calculating the area inside of some geometry shape. Riemann sums are a group of methods for numerical integration that use this strategy:

- partition the area under the curve of some function f into n "slices"
- generate some area estimate for each slice
- add up all of the slices to form an estimate of the integral
- increase the number of slices, and stop when the estimate stops changing.

The Riemann integral of a function f is the limit of this process as $n \to \infty$.

How do you estimate the area of a slice? All of these methods estimate the area by forming a rectangle. For the base, use $x_r - x_l$. For the height, you might use:

- the function value at the left point, $f(x_l)$ (Left Riemann sum)
- the right point, $f(x_r)$ (Right Riemann sum)
- the max of either $max(f(x_l), f(x_r))$ ("upper" Riemann sum)
- the minimum, $min(f(x_l), f(x_r))$, called the "lower" Riemann sums
- the function value at the midpoint: $f(\frac{x_l+x_r}{2})$

This namespace builds up to implementations for `left-integral`, `right-integral`, `upper-integral` and `lower-integral`. `midpoint.cljc` holds an implementation of the Midpoint method.

A closely related method involves forming a trapezoid for each slice. This is equivalent to averaging the left and right Riemann sums. The trapezoid method lives in `trapezoid.cljc`.

2.1.1 Riemann Sum Implementation

We'll start with an inefficient-but-easily-understandable version of these methods. To form a Riemann sum we need to:

- partition some range $[a, b]$ into `n` slices
- call some area-generating function on each slice
- add all of the resulting area estimates together

`windowed-sum` implements this pattern:

```
(defn windowed-sum
  "Takes:

  - ‘area-fn‘, a function of the left and right endpoints of some integration
  slice
  - definite integration bounds ‘a‘ and ‘b‘

  and returns a function of ‘n‘, the number of slices to use for an integration
  estimate.

  ‘area-fn‘ should return an estimate of the area under some curve between the
  ‘l‘ and ‘r‘ bounds it receives."
  [area-fn a b]
```

```
  (fn [n]
    (let [width       (/ (- b a) n)
          grid-points (concat (range a b width) [b])]
      (ua/sum
       (map area-fn grid-points (rest grid-points))))))
```

Test this out with a function that returns 2 for every slice, and we get back an estimate (from the function returned by `windowed-sum`) of 2x the number of slices:

```
(let [area-fn   (fn [l r] 2)
      estimator (windowed-sum area-fn 0 10)]
  [(estimator 10)
   (estimator 20)])

[20.0 40.0]
```

Now, let's implement the four classic "Riemann Integral" methods.

Let's say we want to integrate a function f. The left and right Riemann sums estimate a slice's area as a rectangle with:

- width $==$ $x_r - x_l$, and
- height $==$ $f(x_l)$ or $f(x_r)$, respectively.

`left-sum` is simple to implement, given `windowed-sum`:

```
(defn- left-sum* [f a b]
  (-> (fn [l r] (* (f l) (- r l)))
      (windowed-sum a b)))
```

Every internal slice has the same width, so we can make the sum slightly more efficient by pulling out the constant and multiplying by it a single time.

Internally, we also generate all of the internal "left" points directly from the slice index, instead of pre-partitioning the range. This is fine since we don't need x_r.

```
(defn- left-sum
  "Returns a function of `n`, some number of slices of the total integration
  range, that returns an estimate for the definite integral of $f$ over the
  range $[a, b)$ using a left Riemann sum."
  [f a b]
```

```
  (let [width (- b a)]
    (fn [n]
      (let [h  (/ width n)
            fx (fn [i] (f (+ a (* i h))))]
        (* h (ua/sum fx 0 n))))))
```

`right-sum` is almost identical, except that it uses $f(x_r)$ as the estimate of each rectangle's height:

```
(defn- right-sum* [f a b]
  (-> (fn [l r] (* (f r) (- r l)))
      (windowed-sum a b)))
```

Same trick here to get a more efficient version. This implementation also generates an internal function `fx` of the window index. The only difference from the `left-sum` implementation is an initial offset of `h`, pushing every point to the right side of the window.

```
(defn- right-sum
  "Returns a function of `n`, some number of slices of the total integration
  range, that returns an estimate for the definite integral of $f$ over the
  range $(a, b]$ using a right Riemann sum."
  [f a b]
  (let [width (- b a)]
    (fn [n]
      (let [h     (/ width n)
            start (+ a h)
            fx    (fn [i] (f (+ start (* i h))))]
        (* h (ua/sum fx 0 n))))))
```

The upper Riemann sum generates a slice estimate by taking the maximum of $f(x_l)$ and $f(x_r)$:

```
(defn- upper-sum
  "Returns an estimate for the definite integral of $f$ over the range $[a, b]$
  using an upper Riemann sum.

  This function may or may not make an evaluation at the endpoints $a$ or $b$,
  depending on whether or not the function is increasing or decreasing at the
  endpoints."
  [f a b]
```

```
  (-> (fn [l r] (* (- r l)
                   (max (f l) (f r))))
      (windowed-sum a b)))
```

Similarly, the lower Riemann sum uses the *minimum* of $f(x_l)$ and $f(x_r)$:

```
(defn- lower-sum
  "Returns an estimate for the definite integral of $f$ over the range $[a, b]$
  using a lower Riemann sum.

  This function may or may not make an evaluation at the endpoints $a$ or $b$,
  depending on whether or not the function is increasing or decreasing at the
  endpoints."
  [f a b]
  (-> (fn [l r] (* (- r l)
                   (min (f l) (f r))))
      (windowed-sum a b)))
```

2.1.2 Estimating Integrals with Riemann Sums

Given the tools above, let's attempt to estimate the integral of $f(x) = x^2$ using the left and right Riemann sum methods. (The actual equation for the integral is $\frac{x^3}{3}$).

The functions above return functions of `n`, the number of slices. We can use `(us/powers 2)` to return a sequence of `(1, 2, 4, 8, ...)` and map the function of `n` across this sequence to obtain successively better estimates for $\int_0^{10} x^2$. The true value is $\$10^3\ \frac{}{3=333.333...\$.}$

Here's the `left-sum` estimate:

```
(take 5 (map (left-sum g/square 0 10)
             (us/powers 2)))

(0.0 125.0 218.75 273.4375 302.734375)
```

And the `left-sum` estimate:

```
(take 5 (map (right-sum g/square 0 10)
             (us/powers 2)))

(1000.0 625.0 468.75 398.4375 365.234375)
```

Both estimates are bad at 32 slices and don't seem to be getting better. Even up to $2^16 = 65,536$ slices we haven't converged, and are still far from the true estimate:

```
(-> (map (left-sum g/square 0 10)
         (us/powers 2))
    (us/seq-limit {:maxterms 16}))

{:converged? false, :terms-checked 16, :result 333.31807469949126}
```

This bad convergence behavior is why common wisdom states that you should never use left and right Riemann sums for real work.

But maybe we can do better.

2.1.3 Sequence Acceleration

One answer to this problem is to use "sequence acceleration" via Richardson extrapolation, as described in `richardson.cljc`.

`ir/richardson-sequence` takes a sequence of estimates of some function and "accelerates" the sequence by combining successive estimates.

The estimates have to be functions of some parameter n that decreases by a factor of t for each new element. In the example above, n doubles each time; this is equivalent to thinking about the window width h halving each time, so $t = 2$.

This library's functional style lets us accelerate a sequence of estimates `xs` by simply wrapping it in a call to `(ir/richardson-sequence xs 2)`. Amazing!

Does Richardson extrapolation help?

```
(let [f (fn [x] (* x x))]
  (-> (map (left-sum f 0 10)
           (us/powers 2))
      (ir/richardson-sequence 2)
      (us/seq-limit)))

{:converged? true, :terms-checked 4, :result 333.3333333333333}
```

We now converge to the actual, true value of the integral in 4 terms!

This is going to be useful for each of our Riemann sums, so let's make a function that can accelerate a generic sequence of estimates. The following function takes:

- the sequence of estimates, `estimate-seq`
- a dictionary of "options"

This library is going to adopt an interface that allows the user to configure a potentially very complex integration function by sending a single dictionary of options down to each of its layers. Adopting that style now is going to allow this function to grow to accomodate other methods of sequence acceleration, like polynomial or rational function extrapolation.

For now, `{:accelerate? true}` configures Richardson extrapolation iff the user hasn't specified a custom sequence of integration slices using the `:n` option.

```
(defn- accelerate
  "NOTE - this is only appropriate for Richardson-accelerating sequences with t=2,
  p=q=1.

  This only applies to the Riemann sequences in this namespace!"
  [estimate-seq {:keys [n accelerate?] :or {n 1}}]
  (if (and accelerate? (number? n))
    (ir/richardson-sequence estimate-seq 2 1 1)
    estimate-seq))
```

Check that this works:

```
(let [f (fn [x] (* x x))]
  (-> (map (left-sum f 0 10)
           (us/powers 2))
      (accelerate {:accelerate? true})
      (us/seq-limit)))

{:converged? true, :terms-checked 4, :result 333.3333333333333}
```

Excellent!

2.1.4 Incremental Computation

The results look quite nice; but notice how much redundant computation we're doing.

Consider the evaluation points of a left Riemann sum with 4 slices, next to a left sum with 8 slices:

```
x---x---x---x----
x-x-x-x-x-x-x-x--
```

Every time we double our number of number of evaluations, half of the windows share a left endpoint. The same is true for a right sum:

```
----x---x---x---x
--x-x-x-x-x-x-x-x
```

In both cases, the new points are simply the *midpoints* of the existing slices.

This suggests a strategy for incrementally updating a left or right Riemann sum when doubling the number of points:

- Generate a new midpoint estimate of each `n` slices
- Add this estimate to the previous estimate
- Divide the sum by `2` to scale each NEW slice width down by 2 (since we're doubling the number of slices)

First, implement `midpoint-sum`. This is very close to the implementation for `left-sum`; internally the function adds an offset of $\frac{h}{2}$ to each slice before sampling its function value.

```
(defn midpoint-sum
  "Returns a function of ‘n‘, some number of slices of the total integration
  range, that returns an estimate for the definite integral of $f$ over the
  range $(a, b)$ using midpoint estimates."
  [f a b]
  (let [width (- b a)]
    (fn [n]
      (let [h      (/ width n)
            offset (+ a (/ h 2.0))
            fx     (fn [i] (f (+ offset (* i h))))]
        (* h (ua/sum fx 0 n))))))
```

The next function returns a function that can perform the incremental update to a left or right Riemann sum (and to a midpoint method estimate, as we'll see in `midpoint.cljc`):

```
(defn Sn->S2n
  "Returns a function of:

  - ‘Sn‘: a sum estimate for ‘n‘ partitions, and
  - ‘n‘: the number of partitions

  And returns a new estimate for $S_{2n}$ by sampling the midpoints of each
  slice. This incremental update rule is valid for left and right Riemann sums,
  as well as the midpoint method."
  [f a b]
  (let [midpoints (midpoint-sum f a b)]
    (fn [Sn n]
      (-> (+ Sn (midpoints n))
          (/ 2.0)))))
```

After using `left-sum` to generate an initial estimate, we can use `Sn->S2n` to generate all successive estimates, as long as we always double our slices. This suggests a function that takes an initial number of slices, `n0`, and then uses `reductions` to scan across `(us/powers 2 n0)` with the function returned by `Sn->S2n`:

```
(defn- left-sequence* [f a b n0]
  (let [first-S ((left-sum f a b) n0)
        steps   (us/powers 2 n0)]
    (reductions (Sn->S2n f a b) first-S steps)))
```

Verify that this function returns an equivalent sequence of estimates to the non-incremental `left-sum`, when mapped across powers of 2:

```
(let [f (fn [x] (* x x))]
  (= (take 10 (left-sequence* f 0 10 1))
     (take 10 (map (left-sum f 0 10)
                   (us/powers 2 1)))))
```

```
true
```

2.1.5 Generalizing the Incremental Approach

We need to use the same style for `right-sum`, so let's try and extract the pattern above, of:

- generating an initial estimate of `n0` slices using some function `S-fn`

- refining an estimate of `n0` slices => `n0 / 2` slices using some incremental updater, `next-S-fn`

In fact, because methods like the Midpoint method from `midpoint.cljc` can only incrementally update from `n` => `n/3`, let's make the factor general too.

`geometric-estimate-seq` captures the pattern above:

```
(defn geometric-estimate-seq
  "Accepts:

  - ‘S-fn‘: a function of ‘n‘ that generates a numerical integral estimate from
  ‘n‘ slices of some region, and
  - ‘next-S-fn‘: a function of (previous estimate, previous ‘n‘) => new estimate
  - ‘factor‘: the factor by which ‘n‘ increases for successive estimates
  - ‘n0‘: the initial ‘n‘ to pass to ‘S-fn‘

  The new estimate returned b ‘next-S-fn‘ should be of ‘factor * n‘ slices."
  [S-fn next-S-fn factor n0]
  (let [first-S (S-fn n0)
        steps   (us/powers factor n0)]
    (reductions next-S-fn first-S steps)))
```

And another version of `left-sequence`, implemented using the new function:

```
(defn left-sequence**
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed-open interval $a, b$ by taking left-Riemann sums with

  n0, 2n0, 4n0, ...

  slices."
  ([f a b] (left-sequence** f a b 1))
  ([f a b n0]
   (geometric-estimate-seq (left-sum f a b)
                           (Sn->S2n f a b)
                           2
                           n0)))
```

2.1.6 Incremental Updates with Any Sequence

What if we want to combine the ability to reuse old results with the ability to take successively refined estimates that *don't* look like geometric series? The series 1, 2, 3... of natural numbers is an obvious choice of windows... but only the even powers are able to reuse estimates.

Integration methods like the Bulirsch-Stoer approach depend on sequences like 2, 3, 4, 6...

We absolutely want to be able to save potentially-expensive function evaluations.

One way to do this is to memoize the function `f` that you pass in to any of the methods above.

Alternatively, we could implement a version of `geometric-estimate-seq` that takes *any* sequence of estimate,s and maintains a sort of internal memoization cache.

For every `n`, check the cache for `prev == n/factor`. If it exists in the cache, use `next-S-fn`; else, use `S-fn`, just like we did in `geometric-estimate-seq` for the initial value.

`general-estimate-seq` does this:

```
(defn- general-estimate-seq
  "Accepts:

  - ‘S-fn‘: a function of ‘n‘ that generates a numerical integral estimate from
  ‘n‘ slices of some region, and
  - ‘next-S-fn‘: a function of (previous estimate, previous ‘n‘) => new estimate
  - ‘factor‘: the factor by which ‘next-S-fn‘ increases ‘n‘ in its returned estimate
  - ‘n-seq‘: a monotonically increasing sequence of ‘n‘ slices to use.

  Returns a sequence of estimates of returned by either function for each ‘n‘ in
  ‘n-seq‘. Internally decides whether or not to use ‘S-fn‘ or ‘next-S-fn‘ to
  generate successive estimates."
  [S-fn next-S-fn factor n-seq]
  (let [f (fn [[cache _] n]
            (let [Sn (if (zero? (rem n factor))
                       (let [prev (quot n factor)]
                         (if-let [S-prev (get cache prev)]
                           (next-S-fn S-prev prev)
                           (S-fn n)))
                       (S-fn n))]
```

```
            [(assoc cache n Sn) Sn]))]
    (->> (reductions f [{} nil] n-seq)
         (map second)
         (rest))))
```

We can combine `general-estimate-seq` and `geometric-estimate-seq` into a final method that decides which implementation to call, based on the type of the `n0` argument.

If it's a number, use it as the `n0` seed for a geometrically increasing series of estimates. Else, assume it's a sequence and pass it to `general-estimate-seq`.

```
(defn incrementalize
  "Function that generalizes the ability to create successively-refined estimates
  of an integral, given:

  - ‘S-fn‘: a function of ‘n‘ that generates a numerical integral estimate from
  ‘n‘ slices of some region, and
  - ‘next-S-fn‘: a function of (previous estimate, previous ‘n‘) => new estimate
  - ‘factor‘: the factor by which ‘next-S-fn‘ increases ‘n‘ in its returned estimate
  - ‘n‘: EITHER a number, or a monotonically increasing sequence of ‘n‘ slices to use.

  If ‘n‘ is a sequence, returns a (lazy) sequence of estimates generated for
  each entry in ‘n‘.

  If ‘n‘ is a number, returns a lazy sequence of estimates generated for each
  entry in a geometrically increasing series of inputs $n, n(factor),
  n(factor^2), ....$

  Internally decides whether or not to use ‘S-fn‘ or ‘next-S-fn‘ to generate
  successive estimates."
  [S-fn next-S-fn factor n]
  (let [f (if (number? n)
            geometric-estimate-seq
            general-estimate-seq)]
    (f S-fn next-S-fn factor n)))
```

2.1.7 Final Incremental Implementations

We can use `incrementalize` to write our final version of `left-sequence`, along with a matching version for `right-sequence`.

Notice that we're using `accelerate` from above. The interface should make more sense now:

```
(defn left-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed-open interval $a, b$ by taking left-Riemann sums.

  ## Optional Arguments

  ‘:n‘: If ‘n‘ is a number, returns estimates with $n, 2n, 4n, ...$ slices,
  geometrically increasing by a factor of 2 with each estimate.

  If ‘n‘ is a sequence, the resulting sequence will hold an estimate for each
  integer number of slices in that sequence.

  ‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
  convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
  is ignored."
  ([f a b] (left-sequence f a b {}))
  ([f a b opts]
   (let [S      (left-sum f a b)
         next-S (Sn->S2n f a b)]
     (-> (incrementalize S next-S 2 (:n opts 1))
         (accelerate opts)))))

(defn right-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed-open interval $a, b$ by taking right-Riemann sums.

  ## Optional Arguments

  ‘:n‘: If ‘n‘ is a number, returns estimates with $n, 2n, 4n, ...$ slices,
  geometrically increasing by a factor of 2 with each estimate.

  If ‘n‘ is a sequence, the resulting sequence will hold an estimate for each
  integer number of slices in that sequence.

  ‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
  convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
  is ignored."
```

```
  ([f a b] (right-sequence f a b {}))
  ([f a b opts]
   (let [S      (right-sum f a b)
         next-S (Sn->S2n f a b)]
     (-> (incrementalize S next-S 2 (:n opts 1))
         (accelerate opts)))))
```

`lower-sequence` and `upper-sequence` are similar. They can't take advantage of any incremental speedup, so we generate a sequence of `n~s` internally and map `~lower-sum` and `upper-sum` directly across these.

```
(defn lower-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $(a, b)$ by taking lower-Riemann sums.

  ## Optional Arguments

  ‘:n‘: If ‘n‘ is a number, returns estimates with $n, 2n, 4n, ...$ slices,
  geometrically increasing by a factor of 2 with each estimate.

  If ‘n‘ is a sequence, the resulting sequence will hold an estimate for each
  integer number of slices in that sequence.

  ‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
  convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
  is ignored."
  ([f a b] (lower-sequence f a b {}))
  ([f a b {:keys [n] :or {n 1} :as opts}]
   (let [n-seq (if (number? n)
                 (us/powers 2 n)
                 n)]
     (-> (map (lower-sum f a b) n-seq)
         (accelerate opts)))))

(defn upper-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $(a, b)$ by taking upper-Riemann sums.

  ## Optional Arguments
```

```
‘:n‘: If ‘n‘ is a number, returns estimates with $n, 2n, 4n, ...$ slices,
geometrically increasing by a factor of 2 with each estimate.

If ‘n‘ is a sequence, the resulting sequence will hold an estimate for each
integer number of slices in that sequence.

‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
is ignored."
([f a b] (upper-sequence f a b {}))
([f a b {:keys [n] :or {n 1} :as opts}]
 (let [n-seq (if (number? n)
               (us/powers 2 n)
               n)]
   (-> (map (upper-sum f a b) n-seq)
       (accelerate opts)))))
```

2.1.8 Integral API

Finally, we expose four API methods for each of the {left, right, lower, upper}-Riemann sums.

Each of these makes use a special `qc/defintegrator` "macro"; This style allows us to adopt one final improvement. If the interval a, b is below some threshold, the integral API will take a single slice using the supplied `:area-fn` below and not attempt to converge. See `common.cljc` for more details.

These API interfaces are necessarily limiting. They force the assumptions that you:

- only want to use geometrical sequences that start with $n_0 = 1$
- only want to (optionally) accelerate using Richardson extrapolation

I can imagine a better API, where it's much easier to configure generic sequence acceleration! This will almost certainly show up in the library at some point. For now, here are some notes:

- Richardson extrapolation requires a geometric series of estimates. If you want to use some *other* geometry series with `left-sequence` or `right-sequence`, you can still accelerate with Richardson. Just pass your new factor as `t`.

- For each of {left, right, lower, upper}-Riemann sums, the order of the error terms is 1, 2, 3, 4..., so always provide `p=1` and `q=1` to `richardson-sequence`. `accelerate` does this above.
- If you want to use some NON-geometric seq, you'll need to use the methods in `polynomial.cljc` and `rational.cljc`, which are more general forms of sequence acceleration that use polynomial or rational function extrapolation. Your sequence of `xs` for each of those methods should be `n-seq`.

```
(qc/defintegrator left-integral
  "Returns an estimate of the integral of ‘f‘ across the closed-open interval $a,
  b$ using a left-Riemann sum with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘left-sequence‘ for information on the optional args in ‘opts‘ that
  customize this function's behavior."
  :area-fn (fn [f a b] (* (f a) (- b a)))
  :seq-fn left-sequence)

(qc/defintegrator right-integral
  "Returns an estimate of the integral of ‘f‘ across the closed-open interval $a,
  b$ using a right-Riemann sum with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘right-sequence‘ for information on the optional args in ‘opts‘ that
  customize this function's behavior."
  :area-fn (fn [f a b] (* (f b) (- b a)))
  :seq-fn right-sequence)
```

Upper and lower Riemann sums have the same interface; internally, they're not able to take advantage of incremental summation, since it's not possible to know in advance whether or not the left or right side of the interval should get reused.

```
(qc/defintegrator lower-integral
  "Returns an estimate of the integral of ‘f‘ across the closed-open interval $a,
```

```
  b$ using a lower-Riemann sum with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘lower-sequence‘ for information on the optional args in ‘opts‘ that
  customize this function's behavior."
  :area-fn (fn [f a b] (* (min (f a) (f b)) (- b a)))
  :seq-fn lower-sequence)

(qc/defintegrator upper-integral
  "Returns an estimate of the integral of ‘f‘ across the closed-open interval $a,
  b$ using an upper-Riemann sum with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘upper-sequence‘ for information on the optional args in ‘opts‘ that
  customize this function's behavior."
  :area-fn (fn [f a b] (* (max (f a) (f b)) (- b a)))
  :seq-fn upper-sequence)
```

2.1.9 Next Steps

For a discussion and implementation of the more advanced methods (the workhorse methods that you should actually use!), see `midpoint.cljc` and `trapezoid.cljc`. The midpoint method is the standard choice for open intervals, where you can't evaluate the function at its endpoints. The trapezoid method is standard for closed intervals.

2.2 Midpoint Rule

```
(ns quadrature.midpoint
  (:require [quadrature.interpolate.richardson :as ir]
            [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.riemann :as qr]
            [sicmutils.generic :as g]
            [sicmutils.util :as u]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]))
```

This namespace builds on the ideas introduced in `riemann.cljc`.

`riemann.cljc` described four different integration schemes ({left, right, upper, lower} Riemann sums) that were each conceptually simple, but aren't often used in practice, even in their "accelerated" forms.

One reason for this is that their error terms fall off as h, h^2, h^3, where h is the width of an integration slice. Each order of sequence acceleration can cancel out one of these terms at a time; but still, the performance is not great.

It turns out that by taking the *midpoint* if each interval, instead of either side, you can reduce the order of the error series to $O(h^2)$. This is too good to pass up.

Additionally, because the error terms fall off as $h^2, h^4, h^6, \ldots$, each order of acceleration is worth quite a bit more than in the Riemann sum case.

This namespace follows the same development as `riemann.cljc`:

- implement a simple, easy-to-understand version of the Midpoint method
- make the computation more efficient
- write an incremental version that can reuse prior results
- wrap everything up behind a nice, exposed API

2.2.1 Simple Midpoint Rule

Here's an implementation of a function that can take the midpoint of a single slice:

```
(defn single-midpoint [f a b]
  (let [width      (g/- b a)
        half-width (g// width 2)
        midpoint   (g/+ a half-width)]
    (g/* width (f midpoint))))
```

And a full (though inefficient) integrator using `windowed-sum`:

```
(defn- midpoint-sum* [f a b]
  (let [area-fn (partial single-midpoint f)]
    (qr/windowed-sum area-fn a b)))
```

Let's integrate a triangle!

```
((midpoint-sum* identity 0.0 10.0) 10)
```

```
50.0
```

2.2.2 Efficient Midpoint Method

It turns out that we already had to implement an efficient version of `midpoint-sum` in `riemann.cljc`; the incremental version of left and right Riemann sums added the midpoints of each interval when doubling the number of slices.

We can check our implementation against `qr/midpoint-sum`:

```
(= ((midpoint-sum* identity 0.0 100.0) 10)
   ((qr/midpoint-sum identity 0.0 100.0) 10))
```

```
true
```

We'll use `qr/midpoint-sum` in the upcoming functions.

2.2.3 Incremental Midpoint Method

Unlike the left and right Riemann sums, the Midpoint method can't reuse function evaluations when the number of slices doubles. This is because each evaluation point, on a doubling, becomes the new border between slices:

```
n = 1 |-------x-------|
n = 2 |---x---|---x---|
```

If you *triple* the number of slices from n to $3n$, you can in fact reuse the previous n evaluations:

```
n = 1 |--------x--------|
n = 3 |--x--|--x--|--x--|
```

By scaling `Sn` down by a factor of 3, and adding it to a new sum that only includes the new points (using the new slice width).

BTW: The only place I found this idea mentioned is in Section 4.4 of Press's "Numerical Recipes". I haven't found other references to this trick, or implementations. I'd love to hear about them (via a Github issue) if you find any!

We'll follow the interface we used for `qr/Sn->S2n` and write `Sn->S3n`. This function of f, a, b will return a function that performs the incremental update.

The returned function generates `S3n` across (a, b) with n intervals, and picking out two new points at $\frac{h}{6}$ and $\frac{5h}{6}$ of the way across the old interval. These are the midpoints of the two new slices with width $\frac{h}{3}$.

Sum them all up and add them to $\frac{S_n}{3}$ to generate S_{3n}:

```
(defn- Sn->S3n [f a b]
  (let [width (- b a)]
    (fn [Sn n]
      (let [h        (/ width n)
            delta    (/ h 6)
            l-offset (+ a delta)
            r-offset (+ a (* 5 delta))
            fx (fn [i]
                 (let [ih (* i h)]
                   (+ (f (+ l-offset ih))
                      (f (+ r-offset ih)))))]
        (-> (+ Sn (* h (ua/sum fx 0 n)))
            (/ 3.0))))))
```

Now we can write `midpoint-sequence`, analogous to `qr/left-sequence`. This implementation reuses all the tricks from `qr/incrementalize`; this means it will be smart about using the new incremental logic any time it sees any n multiple of 3, just as the docstring describes.

```
(defn midpoint-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the open interval $(a, b)$ using the Midpoint method.

  ## Optional arguments:

  ‘:n‘: If ‘:n‘ is a number, returns estimates with $n, 3n, 9n, ...$ slices,
  geometrically increasing by a factor of 3 with each estimate.

  If ‘:n‘ is a sequence, the resulting sequence will hold an estimate for each
  integer number of slices in that sequence.

  ‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
  convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
  is ignored."
  ([f a b] (midpoint-sequence f a b {:n 1}))
```

```
  ([f a b {:keys [n accelerate?] :or {n 1}}]
   (let [S      (qr/midpoint-sum f a b)
         next-S (Sn->S3n f a b)
         xs     (qr/incrementalize S next-S 3 n)]
     (if (and accelerate? (number? n))
       (ir/richardson-sequence xs 3 2 2)
       xs))))
```

The following example shows that for the sequence \$2, 3, 4, 6, ... \$ (used in the Bulirsch-Stoer method!), the incrementally-augmented `midpoint-sequence` only performs 253 function evaluations, vs the 315 of the non-incremental `(midpoint-sum f2 0 1)` mapped across the points.

```
(let [f (fn [x] (/ 4 (+ 1 (* x x))))
      [counter1 f1] (u/counted f)
      [counter2 f2] (u/counted f)
      n-seq (interleave
             (iterate (fn [x] (* 2 x)) 2)
             (iterate (fn [x] (* 2 x)) 3))]
  (doall (take 12 (midpoint-sequence f1 0 1 {:n n-seq})))
  (doall (take 12 (map (qr/midpoint-sum f2 0 1) n-seq)))
  [@counter1 @counter2])

[253 315]
```

2.2.4 Final Midpoint API

The final version is analogous the `qr/left-integral` and friends, including an option to `:accelerate?` the final sequence with Richardson extrapolation.

I'm not sure what to call this accelerated method. Accelerating the trapezoid method in this way is called "Romberg integration". Using an n sequence of powers of 2 and accelerating the midpoint method by a single step - taking the second column (index 1) of the Richardson tableau - produces "Milne's method".

The ability to combine these methods makes it easy to produce powerful methods without known names. Beware, and enjoy!

1. Note on Richardson Extrapolation

   We noted above that the the terms of the error series for the midpoint method increase as $h^2, h^4, h^6 \ldots$ Because of this, we pass $p = q = 2$ into

`ir/richardson-sequence` below. Additionally, `integral` hardcodes the factor of `3` and doesn't currently allow for a custom sequence of n. This requires passing $t = 3$ into `ir/richardson-sequence`.

If you want to accelerate some other geometric sequence, call `ir/richardson-sequence` with some other value of `t`.

To accelerate an arbitrary sequence of midpoint evaluations, investigate `polynomial.cljc` or `rational.cljc`. The "Bulirsch-Stoer" method uses either of these to extrapolate the midpoint method using a non-geometric sequence.

```
(qc/defintegrator integral
  "Returns an estimate of the integral of ‘f‘ over the open interval $(a, b)$
  using the Midpoint method with $1, 3, 9 ... 3^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘midpoint-sequence‘ for information on the optional args in ‘opts‘ that
  customize this function’s behavior."
  :area-fn single-midpoint
  :seq-fn midpoint-sequence)
```

2.2.5 Next Steps

If you start with the midpoint method, one single step of Richardson extrapolation (taking the second column of the Richardson tableau) is equivalent to "Milne's rule" (see `milne.cljc`).

The full Richardson-accelerated Midpoint method is an open-interval variant of "Romberg integration" (see `romberg.cljc`).

See the wikipedia entry on Open Newton-Cotes Formulas for more details.

2.3 Trapezoid Rule

same idea but for closed intervals.

```
(ns quadrature.trapezoid
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.riemann :as qr]
```

```
            [quadrature.interpolate.richardson :as ir]
            [sicmutils.function :as f]
            [sicmutils.generic :as g]
            [sicmutils.util :as u]
            [sicmutils.simplify]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]))
```

This namespace builds on the ideas introduced in `riemann.cljc` and `midpoint.cljc`, and follows the pattern of those namespaces:

- implement a simple, easy-to-understand version of the Trapezoid method
- make the computation more efficient
- write an incremental version that can reuse prior results
- wrap everything up behind a nice, exposed API

Let's begin.

2.3.1 Simple Implementation

A nice integration scheme related to the Midpoint method is the "Trapezoid" method. The idea here is to estimate the area of each slice by fitting a trapezoid between the function values at the left and right sides of the slice.

Alternatively, you can think of drawing a line between $f(x_l)$ and $f(x_r)$ and taking the area under the line.

What's the area of a trapezoid? The two slice endpoints are

- $(x_l, f(x_l))$ and
- $(x_r, f(x_r))$

The trapezoid consists of a lower rectangle and a capping triangle. The lower rectangle's area is:

$$(b - a)f(a)$$

.

Just like in the left Riemann sum. The upper triangle's area is one half base times height:

$$\frac{1}{2}(x_r - x_l)(f(x_r) - f(x_l))$$

The sum of these simplifies to:

$$\frac{1}{2}(x_r - x_l)(f(x_l) + f(x_r))$$

Or, in Clojure:

```
(defn single-trapezoid [f xl xr]
  (g// (g/* (g/- xr xl)
            (g/+ (f xl) (f xr)))
       2))
```

We can use the symbolic algebra facilities in the library to show that this simplification is valid:

```
(let [f (f/literal-function 'f)
      square    (g/* (f 'x_l)
                     (g/- 'x_r 'x_l))
      triangle  (g/* (g// 1 2)
                     (g/- 'x_r 'x_l)
                     (g/- (f 'x_r) (f 'x_l)))]
  (g/simplify
   (g/- (single-trapezoid f 'x_l 'x_r)
        (g/+ square triangle))))
```

```
0
```

We can use `qr/windowed-sum` to turn this function into an (inefficient) integrator:

```
(defn- trapezoid-sum* [f a b]
  (qr/windowed-sum (partial single-trapezoid f)
                   a b))
```

Fitting triangles is easy:

```
((trapezoid-sum* identity 0.0 10.0) 10)
```

```
50.0
```

In fact, we can even use our estimator to estimate π:

```
(def ^:private pi-estimator*
  (let [f (fn [x] (/ 4 (+ 1 (* x x))))]
    (trapezoid-sum* f 0.0 1.0)))
```

The accuracy is not bad, for 10 slices:

```
(pi-estimator* 10)
```

```
3.1399259889071587
```

Explicit comparison:

```
(- Math/PI (pi-estimator* 10))
```

```
0.0016666646826344333
```

10,000 slices gets us closer:

```
(< (- Math/PI (pi-estimator* 10000))
   1e-8)
```

```
true
```

Fun fact: the trapezoid method is equal to the *average* of the left and right Riemann sums. You can see that in the equation, but lets verify:

```
(defn- basically-identical? [l-seq r-seq]
  (every? #(< % 1e-15)
          (map - l-seq r-seq)))

(let [points  (take 5 (iterate inc 1))
      average (fn [l r]
                (/ (+ l r) 2))
      f       (fn [x] (/ 4 (+ 1 (* x x))))
      [a b]   [0 1]
      left-estimates  (qr/left-sequence f a b {:n points})
      right-estimates (qr/right-sequence f a b {:n points})]
  (basically-identical? (map (trapezoid-sum f a b) points)
                        (map average
                             left-estimates
                             right-estimates)))
```

```
true
```

2.3.2 Efficient Trapezoid Method

Next let's attempt a more efficient implementation. Looking at `single-trapezoid`, it's clear that each slice evaluates both of its endpoints. This means that each point on a border between two slices earns a contribution of $\frac{f(x)}{2}$ from each slice.

A more efficient implementation would evaluate both endpoints once and then sum (without halving) each interior point.

This interior sum is identical to a left Riemann sum (without the $f(a)$ evaluation), or a right Riemann sum (without $f(b)$).

Here is this idea implemented in Clojure:

```
(defn trapezoid-sum
  "Returns a function of ‘n‘, some number of slices of the total integration
  range, that returns an estimate for the definite integral of $f$ over the
  range $(a, b)$ using the trapezoid method."
  [f a b]
  (let [width (- b a)]
    (fn [n]
      (let [h  (/ width n)
            fx (fn [i] (f (+ a (* i h))))]
        (* h (+ (/ (+ (f a) (f b)) 2)
                (ua/sum fx 1 n)))))))
```

We can define a new `pi-estimator` and check it against our less efficient version:

```
(def ^:private pi-estimator
  (let [f (fn [x] (/ 4 (+ 1 (* x x))))]
    (trapezoid-sum* f 0.0 1.0)))

(basically-identical?
 (map pi-estimator (range 1 100))
 (map pi-estimator* (range 1 100)))

true
```

2.3.3 Incremental Trapezoid Rule

Next let's develop an incremental updater for the Trapezoid rule that lets us reuse evaluation points as we increase the number of slices.

Because interior points of the Trapezoid method mirror the interior points of the left and right Riemann sums, we can piggyback on the incremental implementations for those two methods in developing an incremental Trapezoid implementation.

Consider the evaluation points of the trapezoid method with 2 slices, next to the points of a 4 slice pass:

```
x-------x-------x
x---x---x---x---x
```

The new points are simply the *midpoints* of the existing slices, just like we had for the left (and right) Riemann sums. This means that we can reuse `qr/Sn->S2n` in our definition of the incrementally-enabled `trapezoid-sequence`:

```
(defn trapezoid-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the open interval $(a, b)$ using the Trapezoid method.

  ## Optional arguments:

  ‘:n‘: If ‘:n‘ is a number, returns estimates with $n, 2n, 4n, ...$ slices,
  geometrically increasing by a factor of 2 with each estimate.

  If ‘:n‘ is a sequence, the resulting sequence will hold an estimate for each
  integer number of slices in that sequence.

  ‘:accelerate?‘: if supplied (and ‘n‘ is a number), attempts to accelerate
  convergence using Richardson extrapolation. If ‘n‘ is a sequence this option
  is ignored."
  ([f a b] (trapezoid-sequence f a b {:n 1}))
  ([f a b {:keys [n accelerate?] :or {n 1}}]
   (let [S      (trapezoid-sum f a b)
         next-S (qr/Sn->S2n f a b)
         xs     (qr/incrementalize S next-S 2 n)]
     (if (and accelerate? (number? n))
       (ir/richardson-sequence xs 2 2 2)
       xs))))
```

The following example shows that for the sequence $1, 2, 4, 8, ..., 2^n$, the incrementally-augmented `trapezoid-sequence` only performs $2^n + 1$ function evaluations; ie, the same number of evaluations as the non-incremental

`(trapezoid-sum f2 0 1)` would perform for 2^n slices. (why $2^n + 1$? each interior point is shared, so each trapezoid contributes one evaluation, plus a final evaluation for the right side.)

The example also shows that evaluating *every* n in the sequence costs $\sum_{i=0}^{n} 2^i + 1 = 2^{n+1} + n$ evaluations. As n gets large, this is roughly twice what the incremental implementation costs.

When $n = 11$, the incremental implementation uses 2049 evaluations, while the non-incremental takes 4017.

```
(let [n-elements 11
      f (fn [x] (/ 4 (+ 1 (* x x))))
      [counter1 f1] (u/counted f)
      [counter2 f2] (u/counted f)
      [counter3 f3] (u/counted f)
      n-seq (take (inc n-elements)
                  (iterate (fn [x] (* 2 x)) 1))]
  ;; Incremental version evaluating every `n` in the sequence $1, 2, 4, ...$:
  (doall (trapezoid-sequence f1 0 1 {:n n-seq}))

  ;; Non-incremental version evaluating every `n` in the sequence $1, 2, 4, ...$:
  (doall (map (trapezoid-sum f2 0 1) n-seq))

  ;; A single evaluation of the final `n`
  ((trapezoid-sum f3 0 1) (last n-seq))

  (let [two**n+1 (inc (g/expt 2 n-elements))
        n+2**n (+ n-elements (g/expt 2 (inc n-elements)))]
    (= [2049 4107 2049]
       [two**n+1 n+2**n two**n+1]
       [@counter1 @counter2 @counter3])))

true
```

Another short example that hints of work to come. The incremental implementation is useful in cases where the sequence includes doublings nested in among other values.

For the sequence \$2, 3, 4, 6, ...\$ (used in the Bulirsch-Stoer method!), the incrementally-augmented `trapezoid-sequence` only performs 162 function evaluations, vs the 327 of the non-incremental `(trapezoid-sum f2 0 1)` mapped across the points.

This is a good bit more efficient than the Midpoint method's incremental savings, since factors of 2 come up more often than factors of 3.

```
(let [f (fn [x] (/ 4 (+ 1 (* x x))))
      [counter1 f1] (u/counted f)
      [counter2 f2] (u/counted f)
      n-seq (take 12 (interleave
                      (iterate (fn [x] (* 2 x)) 2)
                      (iterate (fn [x] (* 2 x)) 3)))]
  (doall (trapezoid-sequence f1 0 1 {:n n-seq}))
  (doall (map (trapezoid-sum f2 0 1) n-seq))
  [@counter1 @counter2])
```

```
[162 327]
```

2.3.4 Final Trapezoid API:

The final version is analogous the `qr/left-integral` and friends, including an option to `:accelerate?` the final sequence with Richardson extrapolation. (Accelerating the trapezoid method in this way is called "Romberg integration".)

1. Note on Richardson Extrapolation

   The terms of the error series for the Trapezoid method increase as $h^2, h^4, h^6 \ldots$ (see `https://en.wikipedia.org/wiki/Trapezoidal_rule#Error_analysis`). Because of this, we pass $p = q = 2$ into `ir/richardson-sequence` below. Additionally, `integral` hardcodes the factor of `2` and doesn't currently allow for a custom sequence of n. This is configured by passing $t = 2$ into `ir/richardson-sequence`.

   If you want to accelerate some other geometric sequence, call `ir/richardson-sequence` with some other value of `t`.

   To accelerate an arbitrary sequence of trapezoid evaluations, investigate `polynomial.cljc` or `rational.cljc`. The "Bulirsch-Stoer" method uses either of these to extrapolate the Trapezoid method using a non-geometric sequence.

   ```
   (qc/defintegrator integral
     "Returns an estimate of the integral of ‘f‘ over the closed interval $[a, b]$
     using the Trapezoid method with $1, 2, 4 ... 2^n$ windows for each estimate.
   ```

```
    Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
    passed on to ‘us/seq-limit‘ to configure convergence checking.

    See ‘trapezoid-sequence‘ for information on the optional args in ‘opts‘ that
    customize this function's behavior."
    :area-fn single-trapezoid
    :seq-fn trapezoid-sequence)
```

2.3.5 Next Steps

If you start with the trapezoid method, one single step of Richardson extrapolation (taking the second column of the Richardson tableau) is equivalent to "Simpson's rule". One step using `t=3`, ie, when you *triple* the number of integration slices per step, gets you "Simpson's 3/8 Rule". Two steps of Richardson extrapolation gives you "Boole's rule".

The full Richardson-accelerated Trapezoid method is also known as "Romberg integration" (see `romberg.cljc`).

These methods will appear in their respective namespaces in the `quadrature` package.

See the wikipedia entry on Closed Newton-Cotes Formulas for more details.

3 Sequence Acceleration

3.1 Richardson Extrapolation

is a special case, where we get more efficient by assuming that the x values for the polynomial interpolation go \$1, 1/2, 1/4, ...\$ and that we're extrapolating to 0.

```
(ns quadrature.interpolate.richardson
  "Richardson interpolation is a special case of polynomial interpolation; knowing
  the ratios of successive ‘x‘ coordinates in the point sequence allows a more
  efficient calculation."
  (:require [quadrature.interpolate.polynomial :as ip]
            [sicmutils.generic :as g]
            [sicmutils.util :as u]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]
            [sicmutils.value :as v]))
```

3.1.1 Richardson Interpolation

This approach (and much of this numerical library!) was inspired by Gerald Sussman's "Abstraction in Numerical Methods" paper.

That paper builds up to Richardson interpolation as a method of "series acceleration". The initial example concerns a series of the side lengths of an N-sided polygon inscribed in a unit circle.

The paper derives this relationship between the sidelength of an N- and $2N$-sided polygon:

```
(defn- refine-by-doubling
  "‘s‘ is the side length of an N-sided polygon inscribed in the unit circle. The
  return value is the side length of a 2N-sided polygon."
  [s]
  (/ s (g/sqrt (+ 2 (g/sqrt (- 4 (g/square s)))))))
```

If we can increase the number of sides => infinity, we should reach a circle. The "semi-perimeter" of an N-sided polygon is

$$P_n = \frac{n}{2} S_n$$

In code:

```
(defn- semi-perimeter
  "Returns the semi-perimeter length of an ‘n‘-sided regular polygon with side
  length ‘side-len‘."
  [n side-len]
  (* (/ n 2) side-len))
```

so as $n \to \infty$, P_n should approach π, the half-perimeter of a circle.

Let's start with a square, ie, $n = 4$ and $s_4 = \sqrt{2}$. Clojure's `iterate` function will let us create an infinite sequence of side lengths:

```
(def ^:private side-lengths
  (iterate refine-by-doubling (Math/sqrt 2)))
```

and an infinite sequence of the number of sides:

```
(def ^:private side-numbers
  (iterate #(* 2 %) 4))
```

Mapping a function across two sequences at once generates a new infinite sequence, of semi-perimeter lengths in this case:

```
(def ^:private archimedean-pi-sequence
  (map semi-perimeter side-numbers side-lengths))
```

The following code will print the first 20 terms:

```
(us/pprint 10 archimedean-pi-sequence)
```

```
2.8284271247461903
3.0614674589207183
3.1214451522580524
3.1365484905459393
3.140331156954753
3.141277250932773
3.1415138011443013
3.141572940367092
3.14158772527716
3.1415914215112
```

Unfortunately (for Archimedes, by hand!), as the paper notes, it takes 26 iterations to converge to machine precision:

```
(-> archimedean-pi-sequence
    (us/seq-limit {:tolerance v/machine-epsilon}))
```

```
{:converged? true, :terms-checked 26, :result 3.1415926535897944}
```

Enter Sussman:

> "Imagine poor Archimedes doing the arithmetic by hand: square roots without even the benefit of our place value system! He would be interested in knowing that full precision can be reached on the fifth term, by forming linear combinations of the early terms that allow the limit to be seized by extrapolation." (p4, Abstraction in Numerical Methods)

Sussman does this by noting that you can also write the side length as:

$$S_n = 2 \sin \frac{\pi}{n}$$

Then the taylor series expansion for P_n becomes:

$$\frac{P_n = \frac{n}{2} S_n \ = \frac{n}{2} 2 \sin \frac{\pi}{n} \ = \pi + A \ overn^2 + B}{n^4 ...}$$

A couple things to note:

- At large N, the $\frac{A}{n^2}$ term dominates the truncation error.
- when we double n by taking P_n, that term becomes $\frac{A}{4n^2}$, 4x smaller.

The big idea is to multiply P_{2n} by 4 and subtract P_n (then divide by 3 to cancel out the extra factor). This will erase the $\frac{A}{n^2}$ term and leave a *new* sequence with $\frac{B}{n^4}$ as the dominant error term.

Now keep going and watch the error terms drain away.

Before we write code, let's follow the paper's example and imagine instead some general sequence of \$R(h), R(h/t), R(h/t^2)...\$ (where $t = 2$ in the example above), with a power series expansion that looks like

$$R(h) = A + Bh^{p_1} + Ch^{p_2}...$$

where the exponents \$p$_1$, p$_2$, ...\$ are some OTHER series of error growth. (In the example above, because the taylor series expanson of $n \sin n$ only has even factors, the sequence was the even numbers.)

In that case, the general way to cancel error between successive terms is:

$$R(h/t) - t^{p_1}R(h) = t^{p_1} - 1A + C_1h^{p_2} + ...$$

or:

$$\frac{R(h/t) - t^{p_1}R(h)}{t^{p_1} - 1} = A + C_2h^{p_2} + ...$$

Let's write this in code:

```
(defn- accelerate-sequence
  "Generates a new sequence by combining each term in the input sequence ‘xs‘
  pairwise according to the rules for richardson acceleration.

  ‘xs‘ is a sequence of evaluations of some function of $A$ with its argument
  smaller by a factor of ‘t‘ each time:

  $$A(h), A(h/t), ...$$

  ‘p‘ is the order of the dominant error term for the sequence."
  [xs t p]
  (let [t**p   (Math/pow t p)
        t**p-1 (dec t**p)]
    (map (fn [ah ah-over-t]
```

```
      (/ (- (* t**p ah-over-t) ah)
         t**p-1))
    xs
    (rest xs))))
```

If we start with the original sequence, we can implement Richardson extrapolation by using Clojure's `iterate` with the `accelerate-sequence` function to generate successive columns in the "Richardson Tableau". (This is starting to sound familiar to the scheme for polynomial interpolation, isn't it?)

To keep things general, let's take a general sequence `ps`, defaulting to the sequence of natural numbers.

```
(defn- make-tableau
  "Generates the 'tableau' of succesively accelerated Richardson interpolation
  columns."
  ([xs t] (make-tableau xs t (iterate inc 1)))
  ([xs t ps]
   (->> (iterate (fn [[xs [p & ps]]]
                   [(accelerate-sequence xs t p) ps])
                 [xs ps])
        (map first)
        (take-while seq))))
```

All we really care about are the FIRST terms of each sequence. These approximate the sequence's final value with small and smaller error (see the paper for details).

Polynomial interpolation in `polynomial.cljc` has a similar tableau structure (not by coincidence!), so we can use `ip/first-terms` in the implementation below to fetch this first row.

Now we can put it all together into a sequence transforming function, with nice docs:

```
(defn richardson-sequence
  "Takes:

  - `xs`: a (potentially lazy) sequence of points representing function values
  generated by inputs continually decreasing by a factor of `t`. For example:
  `[f(x), f(x/t), f(x/t^2), ...]`
  - `t`: the ratio between successive inputs that generated `xs`.
```

```
And returns a new (lazy) sequence of 'accelerated' using [Richardson
extrapolation](https://en.wikipedia.org/wiki/Richardson_extrapolation) to
cancel out error terms in the taylor series expansion of `f(x)` around the
value the series to which the series is trying to converge.

Each term in the returned sequence cancels one of the error terms through a
linear combination of neighboring terms in the sequence.

### Custom P Sequence

The three-arity version takes one more argument:

- `p-sequence`: the orders of the error terms in the taylor series expansion
of the function that `xs` is estimating. For example, if `xs` is generated
from some `f(x)` trying to approximate `A`, then `[p_1, p_2...]` etc are the
correction terms:

  $$f(x) = A + B x^{p_1} + C x^{p_2}...$$

The two-arity version uses a default `p-sequence` of `[1, 2, 3, ...]`

### Arithmetic Progression

The FOUR arity version takes `xs` and `t` as before, but instead of
`p-sequence` makes the assumption that `p-sequence` is an arithmetic
progression of the form `p + iq`, customized by:

- `p`: the exponent on the highest-order error term
- `q`: the step size on the error term exponent for each new seq element

## Notes

Richardson extrapolation is a special case of polynomial extrapolation,
implemented in `polynomial.cljc`.

Instead of a sequence of `xs`, if you generate an explicit series of points of
the form `[x (f x)]` with successively smaller `x` values and
polynomial-extrapolate it forward to x == 0 (with,
say, `(polynomial/modified-neville xs 0)`) you'll get the exact same result.
```

```
  Richardson extrapolation is more efficient since it can make assumptions about
  the spacing between points and pre-calculate a few quantities. See the
  namespace for more discussion.

  References:

  - Wikipedia: https://en.wikipedia.org/wiki/Richardson_extrapolation
  - GJS, 'Abstraction in Numerical Methods': https://dspace.mit.edu/bitstream/handle/17
  ([xs t]
   (ip/first-terms
    (make-tableau xs t)))
  ([xs t p-sequence]
   (ip/first-terms
    (make-tableau xs t p-sequence)))
  ([xs t p q]
   (let [arithmetic-p-q (iterate #(+ q %) p)]
     (richardson-sequence xs t arithmetic-p-q))))
```

We can now call this function, combined with `us/seq-limit` (a general-purpose tool that takes elements from a sequence until they converge), to see how much acceleration we can get:

```
(-> (richardson-sequence archimedean-pi-sequence 2 2 2)
    (us/seq-limit {:tolerance v/machine-epsilon}))
```

```
{:converged? true, :terms-checked 7, :result 3.1415926535897936}
```

Much faster!

3.1.2 Richardson Columns

Richardson extrapolation works by cancelling terms in the error terms of a function's taylor expansion about `0`. To cancel the nth error term, the nth derivative has to be defined. Non-smooth functions aren't going to play well with `richardson-sequence` above.

The solution is to look at specific *columns* of the Richardson tableau. Each column is a sequence with one further error term cancelled.

`rational.cljc` and `polynomial.cljc` both have this feature in their tableau-based interpolation functions. The feature here requires a different function, because the argument vector is a bit crowded already in `richardson-sequence` above.

```
(defn richardson-column
  "Function with an identical interface to ‘richardson-sequence‘ above, except for
  an additional second argument ‘col‘.

  ‘richardson-column‘ will return that /column/ offset the interpolation tableau
  instead of the first row. This will give you a sequence of nth-order
  Richardson accelerations taken between point ‘i‘ and the next ‘n‘ points.

  As a reminder, this is the shape of the Richardson tableau:

   p0 p01 p012 p0123 p01234
   p1 p12 p123 p1234 .
   p2 p23 p234 .     .
   p3 p34 .    .     .
   p4 .   .    .     .

  So supplying a ‘column‘ of ‘1‘ gives a single acceleration by combining points
  from column 0; ‘2‘ kills two terms from the error sequence, etc.

  NOTE Given a better interface for ‘richardson-sequence‘, this function could
  be merged with that function."
  ([xs col t]
   (nth (make-tableau xs t) col))
  ([xs col t p-seq]
   (nth (make-tableau xs t p-seq) col))
  ([xs col t p q]
   (let [arithmetic-p-q (iterate #(+ q %) p)]
     (richardson-column xs col t arithmetic-p-q))))
```

3.1.3 Richardson Extrapolation and Polynomial Extrapolation

It turns out that the Richardson extrapolation is a special case of polynomial extrapolation using Neville's algorithm (as described in `polynomial/neville`), evaluated at x == 0.

Neville's algorithm looks like this:

$$P(x) = [(x - x_r)P_l(x) - (x - x_l)P_r(x)]/[x_l - x_r]$$

Where:

- $P(x)$ is a polynomial estimate from some sequence of points $(a, b, c, ...)$ where a point a has the form $(x_a, f(x_a))$
- x_l is the coordinate of the LEFTmost point, x_a
- x_r is the rightmost point, say, x_c in this example
- x is the coordinate where we want to evaluate $P(x)$
- $P_l(x)$ is the estimate with all points but the first, ie, $P_{bc}(x)$
- $P_l(x)$ is the estimate with all points but the LAST, ie, $P_{ab}(x)$

Fill in $x = 0$ and rearrange:

$$\frac{P(0) = [(x_l P_r(0)) - (x_r P_l(x))]}{[x_l - x_r]}$$

In the Richardson extrapolation scheme, one of our parameters was `t`, the ratio between successive elements in the sequence. Now multiply through by $\frac{1=\frac{1}{x_r}}{\frac{1}{x_r}}$ so that our formula contains ratios:

$$\frac{P(0) = [(\frac{x_l}{x_r} P_r(0)) - P_l(x)]}{[\frac{x_l}{x_r} - 1]}$$

Because the sequence of x_i elements looks like $x, x/t, x/t^2$, every recursive step separates x_l and x_r by another factor of t. So

$$\frac{x_l}{x_r} = \frac{x}{\frac{x}{t^n}} = t^n$$

Where n is the difference between the positions of x_l and x_r. So the formula simplifies further to:

$$\frac{P(0) = [(t^n P_r(0)) - P_l(x)]}{[t^n - 1]}$$

Now it looks exactly like Richardson extrapolation. The only difference is that Richardson extrapolation leaves `n` general (and calls it p_1, p_2 etc), so that you can customize the jumps in the error series. (I'm sure there is some detail I'm missing here, so please feel free to make a PR and jump in!)

For the example above, we used a geometric series with $p, q = 2$ to fit the archimedean π sequence. Another way to think about this is that we're fitting a polynomial to the SQUARE of `h` (the side length), not to the actual side length.

Let's confirm that polynomial extrapolation to 0 gives the same result, if we generate squared x values:

```
(let [h**2 (fn [i]
             ;; (1/t^{i + 1})^2
             (-> (/ 1 (Math/pow 2 (inc i)))
                 (Math/pow 2)))
      xs (map-indexed (fn [i fx] [(h**2 i) fx])
                      archimedean-pi-sequence)]
  (= (us/seq-limit
      (richardson-sequence archimedean-pi-sequence 4 1 1))

     (us/seq-limit
      (ip/modified-neville xs 0.0))))
```

```
true
```

Success!

3.2 Polynomial Extrapolation

The general thing that "richardson extrapolation" is doing below. Historically cool and used to accelerate arbitrary integration sequences.

```
(ns quadrature.interpolate.polynomial
  "This namespace contains a discussion of polynomial interpolation, and different
  methods for fitting a polynomial of degree N-1 to N points and evaluating that
  polynomial at some different `x`."
  (:require [sicmutils.generic :as g]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]))
```

First, a Lagrange interpolation polynomial:

```
(defn lagrange
  "Generates a lagrange interpolating polynomial that fits every point in the
  supplied sequence `points` (of form `[x (f x)]`) and returns the value of the
  polynomial evaluated at `x`.

  The Lagrange polynomial has this form:
```

```
g(x) =  (f(a) * [(x-b)(x-c)...] / [(a-b)(a-c)...])
      + (f(b) * [(x-a)(x-c)...] / [(b-a)(b-c)...])
      + ...

for points `[a f(a)], [b f(b)], [c f(c)]` etc.

This particular method of interpolating `x` into the polynomial is
inefficient; any new calculation requires fully recomputing. Takes O(n^2)
operations in the number of points.
"
[points x]
(let [points     (vec points)
      n          (count points)
      build-term (fn [i [a fa]]
                   (let [others (for [j (range n) :when (not= i j)]
                                  (get-in points [j 0]))
                         p (reduce g/* (map #(g/- x %) others))
                         q (reduce g/* (map #(g/- a %) others))]
                     (g// (g/* fa p) q)))]
  (transduce (map-indexed build-term)
             g/+
             points)))
```

Lagrange's interpolating polynomial is straightforward, but not terribly efficient; every time we change `points` or `x` we have to redo the entire calculation. Ideally we'd like to be able to perform:

1. Some computation on `points` that would let us efficiently evaluate the fitted polynomial for different values of `x` in O(n) time, or
2. A computation on a particular `x` that would let us efficiently add new points to the set we use to generate the interpolating polynomial.

"Neville's algorithm" lets us generate the same interpolating polynomial recursively. By flipping the recursion around and generating values from the bottom up, we can achieve goal #2 and add new points incrementally.

3.2.1 Neville's Algorithm

Start the recursion with a single point. Any point $(x, f(x))$ has a unique 0th order polynomial passing through it - the constant function $P(x) = f(x)$. For points x_a, x_b, let's call this P_a, P_b, etc.

P_{ab} is the unique FIRST order polynomial (ie, a line) going through points x_a and x_b.

this first recursive step gives us this rule:

$$P_{ab}(x) = [(x - x_b)P_a(x) - (x - x_a)P_b(x)]/[x_a - x_b]$$

For higher order terms like P_{abcd}, let's call P_{abc} '$\mathrm{P_l}$', and P_{bcd} '$\mathrm{P_r}$' (the polynomial fitted through the left and right set of points).

Similarly, the left and rightmost inputs - x_a and x_b - will be x_l and x_r.

Neville's algorithm states that:

$$P(x) = [(x - x_r)P_l(x) - (x - x_l)P_r(x)]/[x_l - x_r]$$

This recurrence works because the two parents P_l and P_r already agree at all points except x_l and x_r.

```
(defn neville-recursive
  "Top-down implementation of Neville's algorithm.

  Returns the value of `P(x)`, where `P` is a polynomial fit (using Neville's
  algorithm) to every point in the supplied sequence `points` (of form `[x (f
  x)]`)

  The efficiency and results should be identical to
  `quadrature.interpolate/lagrange`. This function represents a step on
  the journey toward more incremental methods of polynomial interpolation.

  References:

  - Press's Numerical Recipes (p103), chapter 3: http://phys.uri.edu/nigh/NumRec/bookf
  - Wikipedia: https://en.wikipedia.org/wiki/Neville%27s_algorithm"
  [points x]
  (letfn [(evaluate [points]
            (if (= 1 (count points))
              (let [[[_ y]] points]
                y)
              (let [l-branch (pop points)
                    r-branch (subvec points 1)
                    [xl]     (first points)
                    [xr]     (peek points)]
                (g// (g/+ (g/* (g/- x xr) (evaluate l-branch))
```

```
                        (g/* (g/- xl x) (evaluate r-branch)))
                    (g/- xl xr)))))]
    (evaluate (vec points))))
```

3.2.2 Tableau-based Methods

Neville's algorithm generates each new polynomial from P_l and P_r, using this recursion to incorporate the full set of points.

You can write these out these relationships in a "tableau":

```
p0
 \
 p01
 /  \
p1  p012
 \  /  \
p12   p0123
 /  \  /  \
p2  p123   p01234
 \  /  \  /
p23   p1234
 /  \  /
p3  p234
 \  /
 p34
 /
p4
```

The next few functions will discuss "rows" and "columns" of the tableau. That refers to the rows and columns of this representation;

```
p0 p01 p012 p0123 p01234
p1 p12 p123 p1234 .
p2 p23 p234 .     .
p3 p34 .    .     .
p4 .   .    .     .
.  .   .    .     .
.  .   .    .     .
.  .   .    .     .
```

The first column here is the initial set of points. Each entry in each successive column is generated through some operation between the entry to its left, and the entry one left and one up.

Look again at Neville's algorithm:

$$P(x) = [(x - x_r)P_l(x) - (x - x_l)P_r(x)]/[x_l - x_r]$$

l refers to the entry in the same row, previous column, while r is one row higher, previous column.

If each cell in the above tableau tracked:

- the value of P(x) for the cell
- x_l, the x value of the leftmost point incorporated so far
- x_r, the right point

we could build up Neville's rule incrementally. Let's attempt to build a function of this signature:

```
(defn neville-incremental*
  "Takes a potentially lazy sequence of ‘points‘ and a point ‘x‘ and generates a
  lazy sequence of approximations of P(x).

  entry N in the returned sequence is the estimate using a polynomial generated
  from the first N points of the input sequence."
  [points x]
  ,,,)
```

First, write a function to process each initial point into a vector that contains each of those required elements:

```
(defn- neville-prepare
  "Processes each point of the form [x, (f x)] into:

  $$[x_l, x_r, p]$$

  where $p$ is the polynomial that spans all points from $l$ to $r$. The
  recursion starts with $p = f(x)$.
  "
  [[x fx]]
  [x x fx])
```

Next, a function that generates the next entry, given l and r:

```
(defn- neville-combine-fn
  "Given some value $x$, returns a function that combines $l$ and $r$ entries in
  the tableau, arranged like this:

  l -- return
     /
    /
   /
  r

  generates the ‘return‘ entry of the form

  $$[x_l, x_r, p]$$."
  [x]
  (fn [[xl _ pl] [_ xr pr]]
    (let [plr (g// (g/+ (g/* (g/- x xr) pl)
                        (g/* (g/- xl x) pr))
                   (g/- xl xr))]
      [xl xr plr])))
```

We can use higher-order functions to turn this function into a NEW function that can transform an entire column:

```
(defn- neville-next-column
  "This function takes some point $x$, and returns a new function that takes some
  column in the tableau and generates the next column."
  [x]
  (fn [prev-column]
    (map (neville-combine-fn x)
         prev-column
         (rest prev-column))))
```

`neville-tableau` will generate the entire tableau:

```
(defn- neville-tableau [points x]
  (->> (map neville-prepare points)
       (iterate (neville-next-column x))
       (take-while seq)))
```

Really, we're only interested in the first row:

$$p_0, p_{01}, p_{012}, p_{0123}, p_{01234}$$

So define a function to grab that:

```
(defn first-terms [tableau]
  (map first tableau))
```

the final piece we need is a function that will extract the estimate from our row of $[x_l, x_r, p]$ vectors:

```
(defn- neville-present [row]
  (map (fn [[_ _ p]] p) row))
```

Putting it all together:

```
(defn neville-incremental*
  "Takes a potentially lazy sequence of ‘points‘ and a point ‘x‘ and generates a
  lazy sequence of approximations of P(x).

  entry N in the returned sequence is the estimate using a polynomial generated
  from the first N points of the input sequence."
  [points x]
  (neville-present
   (first-terms
    (neville-tableau points x))))
```

How do we know this works? We can prove it by using generic arithmetic to compare the full symbolic lagrange polynomial to each entry in the successive approximation.

```
(defn- lagrange-incremental
  "Generates a sequence of estimates of ‘x‘ to polynomials fitted to ‘points‘;
  each entry uses one more point, just like ‘neville-incremental*‘."
  [points x]
  (let [n (count points)]
    (map (fn [i]
           (lagrange (take i points) x))
         (range 1 (inc n)))))
```

Every point is the same:

```
(let [points [['x_1 'y_1] ['x_2 'y_2] ['x_3 'y_3] ['x_4 'y_4]]]
  (map (fn [neville lagrange]
         (g/simplify
          (g/- neville lagrange)))
       (neville-incremental* points 'x)
       (lagrange-incremental points 'x)))

(0 0 0 0)
```

3.2.3 Generic Tableau Processing

The above pattern, of processing tableau entries, is general enough that we can abstract it out into a higher order function that takes a `prepare` and `merge` function and generates a tableau. Any method generating a tableau can use a `present` function to extract the first row, OR to process the tableau in any other way that they like.

This is necessarily more abstract! But we'll specialize it shortly, and rebuild `neville-incremental` into its final form.

I'm keeping `points` in the argument vector for now, vs returning a new function; if you want to do this yourself, curry the function with `(partial tableau-fn prepare merge present)`.

```
(defn tableau-fn
  "Returns a Newton-style approximation tableau, given:

  - ‘prepare‘: a fn that processes each element of the supplied ‘points‘ into
  the state necessary to calculate future tableau entries.

  - ‘merge‘: a fn of ‘l‘and ‘r‘ the tableau entries:

  l -- return
     /
    /
   /
  r

  the inputs are of the same form returned by ‘prepare‘. ‘merge‘ should return a
  new structure of the same form.

  - ‘points‘: the (potentially lazy) sequence of points used to generate the
```

```
  first column of the tableau.
  "
  [prepare merge points]
  (let [next-col (fn [previous-col]
                   (map merge
                        previous-col
                        (rest previous-col)))]
    (->> (map prepare points)
         (iterate next-col)
         (take-while seq))))
```

Redefine `neville-merge` to make it slightly more efficient, with baked-in native operations:

```
(defn- neville-merge
  "Returns a tableau merge function. Identical to ‘neville-combine-fn‘ but uses
  native operations instead of generic operations."
  [x]
  (fn [[xl _ pl] [_ xr pr]]
    (let [p (/ (+ (* (- x xr) pl)
                  (* (- xl x) pr))
               (- xl xr))]
      [xl xr p])))
```

And now, `neville`, identical to `neville-incremental*` except using the generic tableau generator.

The form of the tableau also makes it easy to select a particular *column* instead of just the first row. Columns are powerful because they allow you to successively interpolate between pairs, triplets etc of points, instead of moving onto very high order polynomials.

I'm not sure it's the best interface, but we'll add that arity here.

```
(defn neville
  "Takes:

  - a (potentially lazy) sequence of ‘points‘ of the form ‘[x (f x)]‘ and
  - a point ‘x‘ to interpolate

  and generates a lazy sequence of approximations of P(x). Each entry in the
  return sequence incorporates one more point from ‘points‘ into the P(x)
```

```
estimate.

Said another way: the Nth in the returned sequence is the estimate using a
polynomial generated from the first N points of the input sequence:

p0 p01 p012 p0123 p01234

This function generates each estimate using Neville's algorithm:

$$P(x) = [(x - x_r) P_l(x) - (x - x_l) P_r(x)] / [x_l - x_r]$$

## Column

If you supply an integer for the third `column` argument, `neville` will
return that /column/ of the interpolation tableau instead of the first row.
This will give you a sequence of nth-order polynomial approximations taken
between point `i` and the next `n` points.

As a reminder, this is the shape of the tableau:

 p0 p01 p012 p0123 p01234
 p1 p12 p123 p1234 .
 p2 p23 p234 .     .
 p3 p34 .    .     .
 p4 .   .    .     .

So supplying a `column` of `1` gives a sequence of linear approximations
between pairs of points; `2` gives quadratic approximations between successive
triplets, etc.

References:

- Press's Numerical Recipes (p103), chapter 3: http://phys.uri.edu/nigh/NumRec/bookf
- Wikipedia: https://en.wikipedia.org/wiki/Neville%27s_algorithm
"
([points x]
 (neville-present
  (first-terms
   (tableau-fn neville-prepare
               (neville-merge x)
```

```
                points))))
  ([points x column]
   (-> (tableau-fn neville-prepare
                   (neville-merge x)
                   points)
       (nth column)
       (neville-present))))
```

3.2.4 Modified Neville

Press's Numerical Recipes, chapter 3 (p103) (`http://phys.uri.edu/nigh/NumRec/bookfpdf/f3-1.pdf`) describes a modified version of Neville's algorithm that is slightly more efficient than the version above.

Allan Macleod, in "A comparison of algorithms for polynomial interpolation", discusses this variation under the name "Modified Neville".

By generating the *delta* from each previous estimate in the tableau, Modified Neville is able to swap one of the multiplications above for an addition.

To make this work, instead of tracking the previous p estimate, we track two quantities:

- C_{abc} is the delta between P_{abc} and P_{ab}, ie, P_l.
- D_{abc} is the delta between P_{abc} and P_{bc}, ie, P_r.

We can recover the estimates generated by the original Neville's algorithm by summing C values across the first tableau row.

Equation 3.1.5 in Numerical recipes gives us the equations we need:

$$C_{abc} = [(x_a - x)(C_{bc} - D_{ab})]/[x_a - x_c] \ = [(x_l - x)(C_r - D_l)]/[x_l - x_r]$$

$$D_{abc} = [(x_c - x)(C_{bc} - D_{ab})]/[x_a - x_c] \ = [(x_r - x)(C_r - D_l)]/[x_l - x_r]$$

These equations describe a `merge` function for a tableau processing scheme, with state == `[x_l, x_r, C, D]`.

Let's implement each method, and then combine them into final form. The following methods use the prefix `mn` for "Modified Neville".

```
(defn- mn-prepare
  "Processes an initial point [x (f x)] into the required state:
```

```
  [x_l, x_r, C, D]

  The recursion starts with $C = D = f(x)$."
  [[x fx]]
  [x x fx fx])

(defn- mn-merge
  "Implements the recursion rules described above to generate x_l, x_r, C and D
  for a tableau node, given the usual left and left-up tableau entries."
  [x]
  (fn [[xl _ _ dl] [_ xr cr _]]
    (let [diff   (- cr dl)
          den    (- xl xr)
          factor (/ diff den)
          c      (* factor (- xl x))
          d      (* factor (- xr x))]
      [xl xr c d])))

(defn mn-present
  "Returns a (lazy) sequence of estimates by successively adding C values from the
  first entry of each tableau column. Each C value is the delta from the
  previous estimate."
  [row]
  (ua/scanning-sum
   (map (fn [[_ _ c _]] c) row)))
```

`tableau-fn` allows us to assemble these pieces into a final function that has an interface identical to `neville` above. The implementation is more obfuscated but slightly more efficient.

```
(defn modified-neville
  "Similar to ‘neville‘ (the interface is identical) but slightly more efficient.
  Internally this builds up its estimates by tracking the delta from the
  previous estimate.

  This non-obvious change lets us swap an addition in for a multiplication,
  making the algorithm slightly more efficient.

  See the ‘neville‘ docstring for usage information, and info about the required
  structure of the arguments.
```

```
  The structure of the ‘modified-neville‘ algorithm makes it difficult to select
  a particular column. See ‘neville‘ if you’d like to generate polynomial
  approximations between successive sequences of points.

  References:

  - \"A comparison of algorithms for polynomial interpolation\", A. Macleod,
    https://www.sciencedirect.com/science/article/pii/0771050X82900511
  - Press’s Numerical Recipes (p103), chapter 3: http://phys.uri.edu/nigh/NumRec/bookf
  "
  [points x]
  (mn-present
   (first-terms
    (tableau-fn mn-prepare
                (mn-merge x)
                points))))
```

3.2.5 Folds and Tableaus by Row

The advantage of the method described above, where we generate an entire tableau and lazily pull the first entry off of each column, is that we can pass a lazy sequence in as `points` and get a lazy sequence of successive estimates back. If we don't pull from the result sequence, no computation will occur.

One problem with that structure is that we have to have our sequence of points available when we call a function like `neville`. What if we want to pause, save the current estimate and pick up later where we left off?

Look at the tableau again:

```
p0 p01 p012 p0123 p01234
p1 p12 p123 p1234 .
p2 p23 p234 .     .
p3 p34 .    .     .
p4 .   .    .     .
.  .   .    .     .
.  .   .    .     .
.  .   .    .     .
```

If you stare at this for a while, you might notice that it should be possible to use the `merge` and `present` functions we already have to build the tableau one *row* at a time, given ONLY the previous row:

```
(f [p1 p12 p123 p1234] [x0 fx0]) ;; => [p0 p01 p012 p0123 p01234]
```

Here's something close, using our previous `merge` and `prepare` definitions:

```
(defn- generate-new-row* [prepare merge]
  (fn [prev-row point]
    ;; the new point, once it's prepared, is the first entry in the new row.
    ;; From there, we can treat the previous row as a sequence of "r" values.
    (reduce merge (prepare point) prev-row)))
```

there's a problem here. `reduce` only returns the FINAL value of the aggregation:

```
  (let [f (generate-new-row* prepare present)]
    (f [p1 p12 p123 p1234] [x0 fx0]))
  ;; => p01234
```

We want the entire new row! Lucky for us, Clojure has a version of `reduce`, called `reductions`, that returns each intermediate aggregation result:

```
(defn- generate-new-row [prepare merge]
  (fn [prev-row point]
    (reductions merge (prepare point) prev-row)))

(let [f (generate-new-row prepare present)]
  (f [p1 p12 p123 p1234] [x0 fx0]))
;; => [p0 p01 p012 p0123 p01234]
```

Quick aside here, as we've stumbled across a familiar pattern. The discussion above suggests the idea of a "fold" from functional programming: `https://en.wikipedia.org/wiki/Fold`$_{\text{(higher-order}_{\text{function}})}$

A fold consists of:

- `init`, an initial piece of state called an "accumulator"
- a binary `merge` function that combines ("folds") a new element `x` into the accumulator and returns a value of the same shape / type as `init`.
- a `present` function that transforms the accumulator into a final value.

In Clojure, you perform a fold on a sequence with the `reduce` function:

```
(reduce merge init xs)
```

For example:

```
(reduce + 0.0 (range 10))
```

```
45.0
```

Our `generate-new-row` function from above is exactly the `merge` function of a fold. The accumulator is the latest tableau row:

`init = [], the initial empty row.` `~present~` = the same present function as before (`neville-present` or `mn-present`)

Now that we've identified this new pattern, redefine `generate-new-row` with a new name:

```
(defn tableau-fold-fn
  "Transforms the supplied ‘prepare‘ and ‘merge‘ functions into a new function
  that can merge a new point into a tableau row (generating the next tableau
  row).

  More detail on the arguments:

  - ‘prepare‘: a fn that processes each element of the supplied ‘points‘ into
  the state necessary to calculate future tableau entries.

  - ‘merge‘: a fn of ‘l‘and ‘r‘ the tableau entries:

  l -- return
     /
    /
   /
  r

  the inputs are of the same form returned by ‘prepare‘. ‘merge‘ should return a
  new structure of the same form."
  [prepare merge]
  (fn [prev-row point]
    (reductions merge (prepare point) prev-row)))
```

Next, we can use this to generate specialized fold functions for our two incremental algorithms above - `neville` and `modified-neville`:

```
(defn- neville-fold-fn
  "Returns a function that accepts:

  - ‘previous-row‘: previous row of an interpolation tableau
  - a new point of the form ‘[x (f x)]‘

  and returns the next row of the tableau using the algorithm described in
  ‘neville‘."
  [x]
  (tableau-fold-fn neville-prepare
                   (neville-merge x)))

(defn- modified-neville-fold-fn
  "Returns a function that accepts:

  - ‘previous-row‘: previous row of an interpolation tableau
  - a new point of the form ‘[x (f x)]‘

  and returns the next row of the tableau using the algorithm described in
  ‘modified-neville‘."
  [x]
  (tableau-fold-fn mn-prepare
                   (mn-merge x)))
```

This final function brings back in the notion of `present`. It returns a function that consumes an entire sequence of points, and then passes the final row into the exact `present-fn` we used above:

```
(defn tableau-fold
  "Returns a function that accepts a sequence of points and processes them into a
  tableau by generating successive rows, one at a time.

  The final row is passed into ‘present-fn‘, which generates the final return
  value.

  This is NOT appropriate for lazy sequences! Fully consumes the input."
  [fold-fn present-fn]
  (fn [points]
    (present-fn
     (reduce fold-fn [] points))))
```

Note that these folds process points in the OPPOSITE order as the column-wise tableau functions! Because you build up one row at a time, each new point is PRE-pended to the interpolations in the previous row.

The advantage is that you can save the current row, and then come back and absorb further points later.

The disadvantage is that if you `present` p123, you'll see successive estimates for `[p1, p12, p123]...` but if you then prepend `0`, you'll see estimates for `[p0, p01, p012, p0123]`. These don't share any elements, so they'll be totally different.

If you *reverse* the incoming point sequence, the final row of the fold will in fact equal the row of the column-based method.

If you want a true incremental version of the above code, reverse points! We don't do this automatically in case points is an infinite sequence.

3.2.6 Fold Utilities

`tableau-scan` below will return a function that acts identically to the non-fold, column-wise version of the interpolators. It does this by folding in one point at a time, but processing EVERY intermediate value through the presentation function.

```
(defn tableau-scan
  "Takes a folding function and a final presentation function (of accumulator type
  => return value) and returns a NEW function that:

  - accepts a sequence of incoming points
  - returns the result of calling ‘present‘ on each successive row."
  [fold-fn present-fn]
  (fn [xs]
    (->> (reductions fold-fn [] xs)
         (map present-fn)
         (rest))))
```

And finally, we specialize to our two incremental methods.

```
(defn neville-fold
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of successive approximations of ‘x‘ using polynomials fitted to the
  points in reverse order.

  This function uses the ‘neville‘ algorithm internally."
```

```
  [x]
  (tableau-fold (neville-fold-fn x)
                neville-present))

(defn neville-scan
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of SEQUENCES of successive polynomial approximations of ‘x‘; one
  for each of the supplied points.

  For a sequence a, b, c... you'll see:

  [(neville [a] x)
   (neville [b a] x)
   (neville [c b a] x)
   ...]"
  [x]
  (tableau-scan (neville-fold-fn x)
                neville-present))

(defn modified-neville-fold
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of successive approximations of ‘x‘ using polynomials fitted to the
  points in reverse order.

  This function uses the ‘modified-neville‘ algorithm internally."
  [x]
  (tableau-fold (modified-neville-fold-fn x)
                mn-present))

(defn modified-neville-scan
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of SEQUENCES of successive polynomial approximations of ‘x‘; one
  for each of the supplied points.

  For a sequence a, b, c... you'll see:

  [(modified-neville [a] x)
   (modified-neville [b a] x)
   (modified-neville [c b a] x)
   ...]"
```

```
  [x]
  (tableau-scan (modified-neville-fold-fn x)
                mn-present))
```

Next, check out:

- `rational.cljc` to learn how to interpolate rational functions
- `richardson.cljc` for a specialized implementation of polynomial interpolation, when you know something about the ratios between successive `x` elements in the point sequence.

3.3 Rational Function Extrapolation

```
(ns quadrature.interpolate.rational
  "This namespace contains a discussion of rational function interpolation, and
  different methods for fitting rational functions N points and evaluating them
  at some value ‘x‘."
  (:require [quadrature.interpolate.polynomial :as ip]
            [sicmutils.generic :as g]
            [quadrature.util.aggregate :as ua]
            [quadrature.util.stream :as us]
            [taoensso.timbre :as log]))
```

This namespace contains implementations of rational function interpolation methods. The ALGLib user guide has a nice page on rational function interpolation, which suggests that the Bulirsch-Stoer method, included here, is NOT great, and that there are better methods. We'd love implementations of the others if you agree!

The main method in this package is an incremental version of the Bulirsch-Stoer algorithm.

Just like with polynomial interpolation, let's start with a straightforward implementation of the non-incremental recursive algorithm.

```
(defn bulirsch-stoer-recursive
  "Returns the value of ‘P(x)‘, where ‘P‘ is rational function fit (using the
  Bulirsch-Stoer algorithm, of similar style to Neville's algorithm described in
  ‘polynomial.cljc‘) to every point in the supplied sequence ‘points‘.

  ‘points‘: is a sequence of pairs of the form ‘[x (f x)]‘
```

```
\"The Bulirsch-Stoer algorithm produces the so-called diagonal rational
function, with the degrees of numerator and denominator equal (if m is even)
or with the degree of the denominator larger by one if m is odd.\" ~ Press,
Numerical Recipes, p105

The implementation follows Equation 3.2.3 on on page 105 of Press:
http://phys.uri.edu/nigh/NumRec/bookfpdf/f3-2.pdf.

References:

  - Stoer & Bulirsch, 'Introduction to Numerical Analysis': https://www.amazon.com/In
  - PDF of the same: http://www.math.uni.wroc.pl/~olech/metnum2/Podreczniki/(eBook)%2
  - Press's Numerical Recipes (p105), Section 3.2 http://phys.uri.edu/nigh/NumRec/boo
[points x]
(letfn [(evaluate [points x]
          (cond (empty? points) 0

                (= 1 (count points))
                (let [[[_ y]] points]
                  y)

                :else
                (let [l-branch (pop points)
                      r-branch (subvec points 1)
                      center   (pop r-branch)
                      [xl]     (first points)
                      [xr]     (peek points)
                      rl (evaluate l-branch x)
                      rr (evaluate r-branch x)
                      rc (evaluate center x)
                      p  (g/- rr rl)
                      q  (-> (/ (g/- x xl)
                                (g/- x xr))
                             (g/* (g/- 1 (g// p (g/- rr rc))))
                             (g/- 1))]
                  (g/+ rr (g// p q)))))]
  (let [point-array (vec points)]
    (evaluate point-array x))))
```

We can be a bit more clever, if we reuse the idea of the "tableau" de-

scribed in the polynomial namespace.

```
(defn bulirsch-stoer
  "Takes

  - a (potentially lazy) sequence of ‘points‘ of the form ‘[x (f x)]‘ and
  - a point ‘x‘ to interpolate

  and generates a lazy sequence of approximations of ‘P(x)‘. Each entry in the
  return sequence incorporates one more point from ‘points‘ into the P(x)
  estimate.

  ‘P(x)‘ is rational function fit (using the Bulirsch-Stoer algorithm, of
  similar style to Neville’s algorithm described in ‘polynomial.cljc‘) to every
  point in the supplied sequence ‘points‘.

  \"The Bulirsch-Stoer algorithm produces the so-called diagonal rational
  function, with the degrees of numerator and denominator equal (if m is even)
  or with the degree of the denominator larger by one if m is odd.\" ‘ Press,
  Numerical Recipes, p105

  The implementation follows Equation 3.2.3 on on page 105 of Press:
  http://phys.uri.edu/nigh/NumRec/bookfpdf/f3-2.pdf.

  ## Column

  If you supply an integer for the third (optional) ‘column‘ argument,
  ‘bulirsch-stoer‘ will return that /column/ offset the interpolation tableau
  instead of the first row. This will give you a sequence of nth-order
  polynomial approximations taken between point ‘i‘ and the next ‘n‘ points.

  As a reminder, this is the shape of the tableau:

   p0 p01 p012 p0123 p01234
   p1 p12 p123 p1234 .
   p2 p23 p234 .     .
   p3 p34 .    .     .
   p4 .   .    .     .

  So supplying a ‘column‘ of ‘1‘ gives a sequence of 2-point approximations
```

```
  between pairs of points; ‘2‘ gives 3-point approximations between successive
  triplets, etc.

  References:

    - Stoer & Bulirsch, 'Introduction to Numerical Analysis': https://www.amazon.com/In
    - PDF of the same: http://www.math.uni.wroc.pl/~olech/metnum2/Podreczniki/(eBook)%2
    - Press's Numerical Recipes (p105), Section 3.2 http://phys.uri.edu/nigh/NumRec/boo
  [points x & [column]]
  (let [prepare (fn [[x fx]] [x x 0 fx])
        merge   (fn [[xl _ _ rl] [_ xr rc rr]]
                  (let [p  (- rr rl)
                        q  (-> (/ (- x xl)
                                  (- x xr))
                               (* (- 1 (/ p (- rr rc))))
                               (- 1))]
                    [xl xr rl (+ rr (/ p q))]))
        present (fn [row] (map (fn [[_ _ _ r]] r) row))
        tableau (ip/tableau-fn prepare merge points)]
    (present
     (if column
       (nth tableau column)
       (ip/first-terms tableau)))))
```

3.3.1 Incremental Bulirsch-Stoer

Press, in Numerical Recipes section 3.2, describes a modification to the Bulirsch-Stoer that lets you track the differences from the left and left-up entries in the tableau, just like the modified Neville method in `polynomial.cljc`. the algorithm is implemented below.

```
(defn bs-prepare
  "Processes an initial point [x (f x)] into the required state:

  [x_l, x_r, C, D]

  The recursion starts with $C = D = f(x)$."
  [[x fx]] [x x fx fx])

(defn bs-merge
  "Implements the recursion rules described in Press's Numerical Recipes, section
```

```
  3.2 http://phys.uri.edu/nigh/NumRec/bookfpdf/f3-2.pdf to generate x_l, x_r, C
  and D for a tableau node, given the usual left and left-up tableau entries.

  This merge function ALSO includes a 'zero denominator fix used by Bulirsch and
  Stoer and Henon', in the words of Sussman from `rational.scm` in the scmutils
  package.

  If the denominator is 0, we pass along C from the up-left node and d from the
  previous entry in the row. Otherwise, we use the algorithm to calculate.

  TODO understand why this works, or where it helps!"
  [x]
  (fn [[xl _ _ dl] [_ xr cr _]]
    (let [c-d     (- cr dl)
          d*ratio (-> (/ (- x xl)
                         (- x xr))
                      (* dl))
          den  (- d*ratio cr)]
      (if (zero? den)
        (do (log/info "zero denominator!")
            [xl xr cr dl])
        (let [cnum (* d*ratio c-d)
              dnum (* cr c-d)]
          [xl xr (/ cnum den) (/ dnum den)])))))

(defn modified-bulirsch-stoer
  "Similar to `bulirsch-stoer` (the interface is identical) but slightly more efficient
  Internally this builds up its estimates by tracking the delta from the
  previous estimate.

  This non-obvious change lets us swap an addition in for a division,
  making the algorithm slightly more efficient.

  See the `bulirsch-stoer` docstring for usage information, and info about the
  required structure of the arguments.

  References:

   - Press's Numerical Recipes (p105), Section 3.2 http://phys.uri.edu/nigh/NumRec/book
  [points x]
```

```
  (ip/mn-present
   (ip/first-terms
    (ip/tableau-fn bs-prepare
                   (bs-merge x)
                   points))))
```

3.3.2 Rational Interpolation as a Fold

Just like in `polynomial.cljc`, we can write rational interpolation in the style of a functional fold:

```
(defn modified-bulirsch-stoer-fold-fn
  "Returns a function that accepts:

  - ‘previous-row‘: previous row of an interpolation tableau
  - a new point of the form ‘[x (f x)]‘

  and returns the next row of the tableau using the algorithm described in
  ‘modified-bulirsch-stoer‘."
  [x]
  (ip/tableau-fold-fn
   bs-prepare
   (bs-merge x)))

(defn modified-bulirsch-stoer-fold
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of successive approximations of ‘x‘ using rational functions fitted
  to the points in reverse order."
  [x]
  (ip/tableau-fold
   (modified-bulirsch-stoer-fold-fn x)
   ip/mn-present))

(defn modified-bulirsch-stoer-scan
  "Returns a function that consumes an entire sequence ‘xs‘ of points, and returns
  a sequence of SEQUENCES of successive rational function approximations of ‘x‘;
  one for each of the supplied points.

  For a sequence a, b, c... you'll see:

  [(modified-bulirsch-stoer [a] x)
```

```
   (modified-bulirsch-stoer [b a] x)
   (modified-bulirsch-stoer [c b a] x)
   ...]"
  [x]
  (ip/tableau-scan
   (modified-bulirsch-stoer-fold-fn x)
   ip/mn-present))
```

4 Higher-Order Calculus

4.1 Numerical Derivatives

derivatives using three kinds of central difference formulas... accelerated using Richardson extrapolation, with a nice technique for guarding against underflow.

```
(ns quadrature.derivative
  "Different numerical derivative implementations."
  (:require [sicmutils.calculus.derivative :as d]
            [quadrature.interpolate.richardson :as r]
            [sicmutils.function :as f]
            [sicmutils.generic :as g]
            [sicmutils.infix :as if]
            [sicmutils.util :as u]
            [quadrature.util.stream :as us]
            [sicmutils.value :as v]))
```

This module builds up to an implementation of numerical derivatives. The final function, `D-numeric`, uses Richardson extrapolation to speed up convergence of successively tighter estimates of $f'(x)$ using a few different methods.

The inspiration for this style was Sussman's "Abstraction in Numerical Methods", starting on page 10: `https://dspace.mit.edu/bitstream/handle/1721.1/6060/AIM-997.pdf?sequence=2`

We'll proceed by deriving the methods symbolically, and then implement them numerically.

First, a function that will print nicely rendered infix versions of (simplified) symbolic expressions:

```
(defn show [e]
```

```
  (let [eq (if/->TeX (g/simplify e))]
    (println
     (str "\\begin{equation}\n"
          eq
          "\n\\end{equation}"))))
```

And a function to play with:

```
(def ^:private func
  (f/literal-function 'f))
```

4.1.1 Approximating Derivatives with Taylor Series

The key to all of these methods involves the taylor series expansion of an arbitrary function f around a point x; we know the taylor series will include a term for $f'(x)$, so the goal is to see if we can isolate it.

Here's the taylor series expansions of $f(x+h)$:

```
(def ^:private fx+h
  (->> (d/taylor-series func 'x 'h)
       (take 5)
       (reduce g/+)))
```

Use `show` to print out its infix representation:

```
(show fx+h)
```

$$\frac{1}{24}\,h^4\,D^4f\,(x) + \frac{1}{6}\,h^3\,D^3f\,(x) + \frac{1}{2}\,h^2\,D^2f\,(x) + h\,Df\,(x) + f\,(x) \qquad (1)$$

We can solve this for $Df(x)$ by subtracting $f(x)$ and dividing out h:

```
(show (g// (g/- fx+h (func 'x)) 'h))
```

$$\frac{1}{24}\,h^3\,D^4f\,(x) + \frac{1}{6}\,h^2\,D^3f\,(x) + \frac{1}{2}\,h\,D^2f\,(x) + Df\,(x) \qquad (2)$$

Voila! The remaining terms include $Df(x)$ along with a series of progressively-smaller "error terms" (since $h \to 0$). The first of these terms is $\frac{1}{2}hD^2f(x)$. It will come to dominate the error as $h \to 0$, so we say that the approximation we've just derived has error of $O(h)$.

This particular formula, in the limit as $h \to 0$, is called the "forward difference approximation" to $Df(x)$. Here's the Clojure implementation:

```
(defn forward-difference
  "Returns a single-variable function of a step size ‘h‘ that calculates the
  forward-difference estimate of the the first derivative of ‘f‘ at point ‘x‘:

  f'(x) = [f(x + h) - f(x)] / h

  Optionally accepts a third argument ‘fx == (f x)‘, in case you've already
  calculated it elsewhere and would like to save a function evaluation."
  ([f x] (forward-difference f x (f x)))
  ([f x fx]
   (fn [h]
     (/ (- (f (+ x h)) fx) h))))
```

We could also expand $f(x-h)$:

```
(def ^:private fx-h
  (->> (d/taylor-series func 'x (g/negate 'h))
       (take 5)
       (reduce g/+)))

(show fx-h)
```

$$\frac{1}{24}\,h^4\,D^4f\left(x\right)+\frac{-1}{6}\,h^3\,D^3f\left(x\right)+\frac{1}{2}\,h^2\,D^2f\left(x\right)-h\,Df\left(x\right)+f\left(x\right) \tag{3}$$

and solve for $Df(x)$:

```
(show (g// (g/- (func 'x) fx-h) 'h))
```

$$\frac{-1}{24}\,h^3\,D^4f\left(x\right)+\frac{1}{6}\,h^2\,D^3f\left(x\right)+\frac{-1}{2}\,h\,D^2f\left(x\right)+Df\left(x\right) \tag{4}$$

To get a similar method, called the "backward difference" formula. Here's the implementation:

```
(defn backward-difference
  "Returns a single-variable function of a step size ‘h‘ that calculates the
  backward-difference estimate of the first derivative of ‘f‘ at point ‘x‘:

  f'(x) = [f(x) - f(x - h)] / h
```

```
  Optionally accepts a third argument `fx == (f x)`, in case you've already
  calculated it elsewhere and would like to save a function evaluation."
  ([f x] (backward-difference f x (f x)))
  ([f x fx]
   (fn [h]
     (/ (- fx (f (- x h))) h))))
```

Notice that the two expansions, of $f(x+h)$ and $f(x-h)$, share every term paired with an even power of h. The terms associated with odd powers of h alternate in sign (because of the $-h$ in the expansion of $f(x-h)$).

We can find yet another method for approximating $Df(x)$ if we subtract these two series. We're trying to solve for $Df(x)$, and $Df(x)$ appears paired with h, an odd-powered term... so subtracting $f(x-h)$ should double that term, not erase it. Let's see:

```
(show (g/- fx+h fx-h))
```

$$\frac{1}{3}\,h^3\,D^3f\,(x) + 2\,h\,Df\,(x) \tag{5}$$

Amazing! Now solve for $Df(x)$:

```
(show (g// (g/- fx+h fx-h)
           (g/* 2 'h)))
```

$$\frac{1}{6}\,h^2\,D^3f\,(x) + Df\,(x) \tag{6}$$

We're left with $Df(x) + O(h^2)$, a quadratic error term in h. (Of course if we'd expanded to more than initial terms in the taylor series we'd see a long error series with only even powers.)

This formula is called the "central difference" approximation to the first derivative. Here's the implementation:

```
(defn central-difference
  "Returns a single-variable function of a step size `h` that calculates the
  central-difference estimate of the first derivative of `f` at point `x`:

  f'(x) = [f(x + h) - f(x - h)] / 2h"
  [f x]
```

```
  (fn [h]
    (/ (- (f (+ x h)) (f (- x h)))
       (* 2 h))))
```

There's one more approximation we can extract from these two expansions. We noted earlier that the terms associated with odd powers of h alternate in sign. If we add the two series, these odd terms should all cancel out. Let's see:

```
(show (g/+ fx-h fx+h))
```

$$\frac{1}{12} h^4 D^4 f(x) + h^2 D^2 f(x) + 2 f(x) \tag{7}$$

Interesting. The $Df(x)$ term is gone. Remember that we have $f(x)$ available; the first unknown term in the series is now $D^2 f(x)$. Solve for that term:

```
(show (g// (g/- (g/+ fx-h fx+h) (g/* 2 (func 'x)))
           (g/square 'h)))
```

$$\frac{1}{12} h^2 D^4 f(x) + D^2 f(x) \tag{8}$$

This is the "central difference" approximation to the *second* derivative of f. Note that the error term here is quadratic in h. Here it is in code:

```
(defn central-difference-d2
  "Returns a single-variable function of a step size `h` that calculates the
  central-difference estimate of the second derivative of `f` at point `x`:

  f''(x) = [f(x + h) - 2f(x) + f(x - h)] / h^2

  Optionally accepts a third argument `fx == (f x)`, in case you've already
  calculated it elsewhere and would like to save a function evaluation."
  ([f x] (central-difference-d2 f x (f x)))
  ([f x fx]
   (let [fx*2 (* 2 fx)]
     (fn [h]
       (/ (- (+ (f (+ x h))
                (f (- x h)))
             fx*2)
          (* h h))))))
```

4.1.2 Taking Derivatives

Let's attempt to use these estimates and see how accurate they are. (This section follows Sussman starting on page 10.)

The following function returns a new function that approximates $Df(x)$ using the central difference method, with a fixed value of $h = 0.00001$:

```
(defn- make-derivative-fn
  [f]
  (fn [x]
    (let [h 1e-5]
      ((central-difference f x) h))))
```

The error here is not great, even for a simple function:

```
((make-derivative-fn g/square) 3)
```

```
6.000000000039306
```

Let's experiment instead with letting $h \to 0$. This next function takes a function f, a value of x and an initial h, and generates a stream of central difference approximations to $Df(x)$ using successively halved values of h, ie, $(h, h/2, h/4, h/8, ...)$

```
(defn- central-diff-stream [f x h]
  (map (central-difference f x)
       (us/zeno 2 h)))
```

Let's print 20 of the first 60 terms (taking every 3 so we see any pattern):

```
(->> (central-diff-stream g/sqrt 1 0.1)
     (take-nth 3)
     (us/pprint 20))
```

```
0.5006277505981893
0.5000097662926306
0.5000001525880649
0.5000000023844109
0.5000000000381988
0.5000000000109139
0.4999999998835847
0.5000000004656613
```

```
0.5000000074505806
0.49999989569187164
0.5000001192092896
0.4999971389770508
0.500030517578125
0.49957275390625
0.50048828125
0.48828125
0.625
0.0
0.0
0.0
```

At first, the series converges toward the proper value. But as h gets smaller, $f(x+h)$ and $f(x-h)$ get so close together that their difference is less than the minimum epsilon allowed by the system's floating point representation.

As Sussman states: "Hence we are in a race between truncation error, which starts out large and gets smaller, and roundoff error, which starts small and gets larger." ~Sussman, p12

4.1.3 Roundoff Error

We can actually analyze and quantify how many halvings we can apply to h before roundoff error swamps our calculation.

Why does roundoff error occur? From Sussman: "Any real number x, represented in the machine, is rounded to a value $x(1+e)$, where e is effectively a random variable whose absolute value is on the order of the machine epsilon ϵ: that smallest positive number for which 1.0 and $1.0+\epsilon$ can be distinguished."

In the current library, `v/machine-epsilon` holds this value.

Our goal, then, is to see if we can figure out when the error due to roundoff grows so large that it exceeds the tolerance we want to apply to our calculation.

For the central difference formula:

$$\frac{f'(x) = f(x+h) - f(x-h)}{2h}$$

without any roundoff error, the numerator *should* be equal to $2hf'(x)$. In reality, for small values of h, $f(x+h)$ and $f(x-h)$ both have machine

representations in error by about $f(x)\epsilon$. Their difference doesn't change the order, so we can say that their difference also has error of $f(x)\epsilon$.

Dividing these two together, the relative error is:

$$\epsilon \left| \frac{f(x)}{2hf'(x)} \right|$$

The relative error doubles each time h is halved. This is technically just the relative error of the numerator of the central difference method, but we know the denominator $2h$ to full precision, so we can ignore it here.

If we actually calculate this ratio, we'll find the INITIAL relative error due to roundoff for a given h. Also, because we want to make sure that we're working in integer multiples of machine epsilon, let's actually take the next-highest-integer of the ratio above. The following method takes the ratio above as an argument, and returns:

$$1 + floor(|ratio|)$$

```
(defn- roundoff-units
  "Returns the number of 'roundoff units', ie, multiples of the machine epsilon,
  that roundoff error contributes to the total relative error, given a relative
  error percentage estimated for some initial step size $h$."
  [rel-error-ratio]
  (inc
   (Math/floor
    (Math/abs
     (double rel-error-ratio)))))
```

That calculation, as the documentation states, returns the number of "roundoff units". Let's call it r.

Each iteration doubles the relative error contributed by roundoff. Given some tolerance, how many roundoff error doublings (or, equivalently, halvings of h) can we tolerate before roundoff error swamps our calculation?

Here's the solution:

```
(defn- max-iterations
  "Solution for `n`, in:

  `initial-error` * 2^n <= `tolerance`"
  [units tolerance]
  (let [initial-error (* v/machine-epsilon units)]
```

```
    (Math/floor
     (/ (Math/log (/ tolerance initial-error))
        (Math/log 2)))))
```

Let's combine these two ideas into a final function, `terms-before-roundoff`, that calculates how items we can pull from a sequence like `central-diff-stream` above before roundoff swamps us. (This is 1 + max iterations, because we need to include the first point.)

```
(defn- terms-before-roundoff
  "Generates a default max number of terms, based on roundoff error estimates."
  [ratio tolerance]
  (inc
   (max-iterations (roundoff-units ratio)
                   tolerance)))
```

How many terms are we allowed to examine for an estimate of the derivative of $f(x) = \sqrt(x)$, with an initial $h = 0.1$?

```
(let [f         g/sqrt
      x         1
      h         0.1
      tolerance 1e-13
      ratio     (/ (f x)
                   (- (f (+ x h))
                      (f (- x h))))]
  (terms-before-roundoff ratio tolerance))
```

```
6.0
```

6 terms, or 5 halvings, down to $\frac{h=0.1}{2^5=0.003125}$. How many terms does the sequence take to converge?

```
(-> (central-diff-stream g/sqrt 1 0.1)
    (us/seq-limit {:tolerance 1e-13}))
```

```
{:converged? true, :terms-checked 15, :result 0.5000000000109139}
```

15 is far beyond the level where roundoff error has rendered our results untrustworthy.

4.1.4 Richardson Extrapolation

We need a way to converge more quickly. `richardson.cljc` lays out a general method of "sequence acceleration" that we can use here, since we know the arithmetic progression of the terms in the error series for each of our methods above.

For the central difference method, our highest-order error term has an exponent of $p = 2$, and successive terms are all even. `r/richardson-sequence` takes `p` and `q` for an arithmetic sequence of error exponents $p, p + q, p + 2q...$

It also needs the initial size h of our sequence progression.

Given that information, we can transform our existing sequence of estimates into an accelerated sequence of estimates using Richardson extrapolation. Does it converge in fewer terms?

```
(let [h 0.1, p 2, q 2]
  (-> (central-diff-stream g/sqrt 1 h)
      (r/richardson-sequence h p q)
      (us/seq-limit {:tolerance 1e-13})))
```

```
{:converged? true, :terms-checked 6, :result 0.5006325594766895}
```

Happily, it does, in only 5 terms instead of 15! This brings convergence in under our limit of 6 total terms.

If you're interested in more details of Richardson extrapolation, please see `richardson.cljc`! For now we'll proceed.

4.1.5 Putting it All Together

We're ready to write our final numeric differentiation routine, `D-numeric`. First, some supporting structure. We support four methods, so let's describe them using keywords in a set:

```
(def valid-methods
  #{:central :central-d2 :forward :backward})
```

To apply one of the methods, we need to be able to:

- generate the method's estimate as a function of h
- calculate the "relative error ratio" that we used above to calculate a maximum number of terms to analyze

- know the order p of the highest order error term, and
- the increment q of successive error terms

Once again, `richardson.cljc` for a discussion of p and q.

This `configs` function bundles all of this together. I don't know that this is the best abstraction, but I don't know yet of any other methods for numeric differentiation, so it'll do for now.

Note here that $p = q = 2$ for both central difference methods, just like we determined above. the forward and backward difference methods both have all of the remaining terms from the taylor expansion in their error series, so they only get $p = q = 1$.

```
(defn- configs [method f x fx]
  (case method
    :forward
    {:p 1
     :q 1
     :function (forward-difference f x fx)
     :ratio-fn (fn [h] (/ fx (- (f (+ x h)) fx)))}

    :central
    {:p 2
     :q 2
     :function (central-difference f x)
     :ratio-fn (fn [h]
                 (/ fx (- (f (+ x h))
                          (f (- x h)))))}

    :backward
    {:p 1
     :q 1
     :function (backward-difference f x fx)
     :ratio-fn (fn [h]
                 (/ fx (- fx (f (- x h)))))}

    :central-d2
    {:p 2
     :q 2
     :function (central-difference-d2 f x fx)
     :ratio-fn (fn [h]
```

```
                (- (+ (f (+ x h))
                      (f (- x h)))
                   (* 2 fx)))}

    (u/illegal
     (str "Invalid method: " method ". Please try one of " valid-methods))))

(defn- fill-defaults
  "Fills in default values required by `D-numeric`. Any option not used by
  `D-numeric` gets passed on to `us/seq-limit`."
  [m]
  (let [defaults {:tolerance (Math/sqrt v/machine-epsilon)
                  :method    :central}
        {:keys [method] :as opts} (merge defaults m)]
    (assert (contains? valid-methods method)
            (str method " is not a valid method. Please try one of: " valid-methods))
    opts))

(defn D-numeric
  "Takes a function `f: R => R` (function of a single real variable), and returns
  a new function of `x` that approximates the derivative $Df(x)$ (or $D^2f(x)$
  if you pass `:method :central-d2`).

  Returns the estimated value of the derivative at `x`. If you pass `:info?
  true`, the fn returns a dictionary of the results of `us/seq-limit`:

  {:converged? <boolean>
   :terms-checked <int>
   :result <derivative estimate>}

  Make sure to visit `sicmutils.calculus.derivative/D` if you want symbolic or
  automatic differentiation.

  ## Roundoff Estimate

  The returned function will attempt to estimate how many times it can halve the
  step size used to estimate the derivative before roundoff error swamps the
  calculation, and force the function to return (with `:converged? false`, if
  you pass `:info?`)
```

```
## Optional Arguments

‘D-numeric‘ takes optional args as its second param. Any of these can be
overridden by passing a second argument to the function returned by
‘D-numeric‘; helpful for setting defaults and then overriding them later.

The returned function passes through these and any other options to
‘us/seq-limit‘, where they control the sequence of richardson
extrapolation-accelerated estimates.

Options:

- ‘:method‘: one of ‘:central‘, ‘:central-d2‘, ‘:forward‘ or ‘:backward‘.
‘:central-d2‘ forces a second derivative estimate; the other methods configure
a first derivative estimator.

- ‘:info?‘ if false (default), returns the estimated value of ‘x‘. If true,
returns a dictionary with more information (see ‘D-numeric‘’s docstring for
more info.)

- ‘:initial-h‘: the initial ‘h‘ to use for derivative estimates before $h \to
0$. Defaults to 0.1 * abs(x).

- ‘:tolerance‘: see ‘us/stream-limit‘ for a discussion of how this value
handles relative vs absolute tolerance. $\\sqrt(\\epsilon)$ by default, where
$\\epsilon$ = machine tolerance.

- ‘:maxterms‘: the maximum number of terms to consider when hunting for a
derivative estimate. This defaults to an estimate generated internally,
designed to prevent roundoff error from swamping the result. If you want to
disable this feature, set ‘:maxterms‘ to something moderately large, like
‘:maxterms 100‘. But do so carefully! See the surrounding namespace for a
larger discussion."
([f] (D-numeric f {}))
([f opts]
 (let [opts (fill-defaults opts)]
   (fn df
     ([x] (df x {}))
     ([x overrides]
      (let [{:keys [maxterms tolerance initial-h method info?] :as opts} (merge opts
```

```
            {:keys [ratio-fn function p q]} (configs method f x (f x))
            h (or initial-h (* 0.1 (g/abs x)))
            n (or maxterms (terms-before-roundoff
                            (ratio-fn h)
                            tolerance))
            estimates (map function (us/zeno 2 h))
            result    (-> (r/richardson-sequence estimates 2 p q)
                          (us/seq-limit (assoc opts :maxterms n)))]
        (if info? result (:result result))))))))
```

More resources about numerical differentiation:

- "Abstraction in Numerical Methods", Gerald Sussman, p10+: `https://dspace.mit.edu/bitstream/handle/1721.1/6060/AIM-997.pdf?sequence=2`
- "Numerical Differentiation and Richardson Extrapolation" lecture notes by Joseph Mahaffy `https://jmahaffy.sdsu.edu/courses/f16/math541/beamer/richard.pdf`
- UBC's "Mathematical Python" course: `https://www.math.ubc.ca/~pwalls/math-python/differentiation/differentiation/`

#+end$_{\text{src}}$

4.2 Simpson's Rule

fit a parabola to every slice. OR, "accelerate" the trapezoid method with one step of Richardson extrapolation!

```
(ns quadrature.simpson
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.trapezoid :as qt]
            [quadrature.interpolate.richardson :as ir]))
```

This numerical integration method is a closed Newton-Cotes formula; for each integral slice, Simpson's rule samples each endpoint and the midpoint and combines them into an area estimate for this slice using the following formula:

$$\frac{h}{3}(f_0 + 4f_1 + f_2)$$

Given a window of $[a, b]$ and a "step size" of $h = \frac{b-a}{2}$. The point f_i is the point i steps into the window.

There are a few simpler ways to understand this:

- Simpson's rule is simply the trapezoid method (see `trapezoid.cljc`), subject to a single refinement of "Richardson extrapolation".
- The trapezoid method fits a line to each integration slice. Simpson's rule fits a quadratic to each slice.
- Simpson's rule S is the weighted average of the Midpoint rule M and the trapezoid rule T:

$$S = \frac{2M + T}{3}$$

The test namespace contains a symbolic proof that the Richardson-extrapolated Trapezoid method is equivalent to using the formula above to calculate Simpson's rule directly.

```
(defn simpson-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $[a, b]$ using Simpson’s rule.

  Simpson’s rule is equivalent to the trapezoid method subject to one refinement
  of Richardson extrapolation. The trapezoid method fits a line to each
  integration slice. Simpson’s rule fits a quadratic to each slice.

  Returns estimates with $n, 2n, 4n, ...$ slices, geometrically increasing by a
  factor of 2 with each estimate.

  ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use."
  ([f a b] (simpson-sequence f a b {:n 1}))
  ([f a b {:keys [n] :or {n 1}}]
   {:pre [(number? n)]}
   (-> (qt/trapezoid-sequence f a b n)
       (ir/richardson-column 1 2 2 2))))

(qc/defintegrator integral
```

```
  "Returns an estimate of the integral of `f` over the closed interval $[a, b]$
  using Simpson's rule with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts `opts`, a dict of optional arguments. All of these get
  passed on to `us/seq-limit` to configure convergence checking.

  See `simpson-sequence` for more information about Simpson's rule, caveats that
  might apply when using this integration method and information on the optional
  args in `opts` that customize this function's behavior."
  :area-fn (comp first simpson-sequence)
  :seq-fn simpson-sequence)
```

4.3 Simpson's 3/8 Rule

```
(ns quadrature.simpson38
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.trapezoid :as qt]
            [quadrature.interpolate.richardson :as ir]
            [quadrature.util.stream :as us]))
```

This numerical integration method is a closed Newton-Cotes formula; for each integral slice, Simpson's 3/8 rule samples each endpoint and TWO interior, equally spaced points, and combines them into an area estimate for this slice using the following formula:

$$\frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3)$$

Given a window of $[a, b]$ and a "step size" of $h = \frac{b-a}{3}$. The point f_i is the point i steps into the window.

There are a few simpler ways to understand this:

- Simpson's 3/8 rule is simply the trapezoid method (see `trapezoid.cljc`), subject to a single refinement of "Richardson extrapolation", with an threefold-increase of integration slices at each step, from $n \to 3n$.
- The trapezoid method fits a line to each integration slice. Simpson's 3/8 rule fits a cubic to each slice.

The test namespace contains a symbolic proof that the Richardson-extrapolated Trapezoid method is equivalent to using the formula above to calculate Simpson's 3/8 rule directly.

```
(defn simpson38-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $[a, b]$ using Simpson’s 3/8 rule.

  Simpson’s 3/8 rule is equivalent to the trapezoid method subject to:

  - one refinement of Richardson extrapolation, and

  - a geometric increase of integration slices by a factor of 3 for each
  sequence element. (the Trapezoid method increases by a factor of 2 by
  default.)

  The trapezoid method fits a line to each integration slice. Simpson’s 3/8 rule
  fits a cubic to each slice.

  Returns estimates with $n, 3n, 9n, ...n3^i$ slices, geometrically increasing by a
  factor of 3 with each estimate.

    ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use.

  NOTE: the Trapezoid method is able to reuse function evaluations as its
  windows narrow /only/ when increasing the number of integration slices by 2.
  Simpson’s 3/8 rule increases the number of slices geometrically by a factor of
  2 each time, so it will never hit the incremental path. You may want to
  memoize your function before calling ‘simpson38-sequence‘."
  ([f a b] (simpson38-sequence f a b {:n 1}))
  ([f a b {:keys [n] :or {n 1}}]
   {:pre [(number? n)]}
   (-> (qt/trapezoid-sequence f a b (us/powers 3 n))
       (ir/richardson-column 1 3 2 2))))

(qc/defintegrator integral
  "Returns an estimate of the integral of ‘f‘ over the closed interval $[a, b]$
  using Simpson’s 3/8 rule with $1, 3, 9 ... 3^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.
```

```
  See ‘simpson38-sequence‘ for more information about Simpson's 3/8 rule, caveats
  that might apply when using this integration method and information on the
  optional args in ‘opts‘ that customize this function's behavior."
  :area-fn (comp first simpson38-sequence)
  :seq-fn simpson38-sequence)
```

4.4 Boole's Rule

trapezoid method plus two steps of Richardson extrapolation. (Are you starting to see the pattern??)

```
(ns quadrature.boole
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.trapezoid :as qt]
            [quadrature.interpolate.richardson :as ir]))
```

NOTE: Boole's Rule is commonly mis-spelled as "Bode's Rule"!

This numerical integration method is a closed Newton-Cotes formula; for each integral slice, Boole's rule samples:

- each endpoint
- three interior points

and combines them into an area estimate for this slice using the following formula:

$$\frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$$

Given a window of $[a, b]$ and a "step size" of $h = \frac{b-a}{4}$. The point f_i is the point i steps into the window.

There are a few simpler ways to understand this:

- Boole's rule is simply the trapezoid method (see `trapezoid.cljc`), subject to *two* refinements of "Richardson extrapolation".
- The trapezoid method fits a line to each integration slice. Boole's rule fits a quartic (4th-order) polynomial to each slice.

The test namespace contains a symbolic proof that the Richardson-extrapolated Trapezoid method is equivalent to using the formula above to calculate Boole's rule directly.

```
(defn boole-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $[a, b]$ using Boole’s rule.

  Boole’s rule is equivalent to the trapezoid method subject to two refinements
  of Richardson extrapolation. The trapezoid method fits a line to each
  integration slice. Boole’s rule fits a quartic to each slice.

  Returns estimates with $n, 2n, 4n, ...$ slices, geometrically increasing by a
  factor of 2 with each estimate.

  ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use."
  ([f a b] (boole-sequence f a b {:n 1}))
  ([f a b {:keys [n] :or {n 1}}]
   {:pre [(number? n)]}
   (-> (qt/trapezoid-sequence f a b n)
       (ir/richardson-column 2 2 2 2))))

(qc/defintegrator integral
  "Returns an estimate of the integral of ‘f‘ over the closed interval $[a, b]$
  using Boole’s rule with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘boole-sequence‘ for more information about Boole’s rule, caveats that
  might apply when using this integration method and information on the optional
  args in ‘opts‘ that customize this function’s behavior."
  :area-fn (comp first boole-sequence)
  :seq-fn boole-sequence)
```

4.5 Romberg Integration

```
(ns quadrature.romberg
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.midpoint :as qm]
            [quadrature.trapezoid :as qt]
```

```
[quadrature.interpolate.richardson :as ir]))
```

Romberg's method is a technique for estimating a definite integral over a closed (or open) range a, b:

$$\int_a^b f(x)dx$$

By applying Richardson extrapolation (see `richardson.cljc`) to either the Trapezoid method or the Midpoint method.

The implementation of Richardson extrapolation in this library can be applied to any methods; many of the numerical quadrature methods (Simpson, Simpson's 3/8, Milne, Boole) involve a single step of Richardson extrapolation.

Romberg integration goes all the way. A nice way to think about this algorithm is this:

- Generate a sequence of estimates of the definite integral using the Trapezoid or Midpoint methods on a geometrically increasing number of integration slices of width h. This gives you a sequence of N points of the form $(h, A(h))$, where A is the integral estimate.
- Each time a new point becomes available, fit a polynomial of order $N - 1$ to all N points... and then extrapolate to $A(0)$, the magical area estimate where the width of each integration slice is 0.

For a wonderful reference that builds up to the ideas of Richardson extrapolation and Romberg integration, see Sussman's "Abstraction in Numerical Methods".

References:

- Press's Numerical Recipes (p134), Section 4.3 `http://phys.uri.edu/nigh/NumRec/bookfpdf/f4-3.pdf`
- Numerical Recipes 4.4 for open-interval Romberg `http://phys.uri.edu/nigh/NumRec/bookfpdf/f4-4.pdf`
- Halfant & Sussman, "Abstraction in Numerical Methods".
- Wikipedia: `https://en.wikipedia.org/wiki/Romberg%27s_method`

```
(defn open-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the open interval $(a, b)$ by applying Richardson extrapolation to
  successive integral estimates from the Midpoint rule.

  Returns estimates formed by combining $n, 3n, 9n, ...$ slices, geometrically
  increasing by a factor of 3 with each estimate. This factor of 3 is because,
  internally, the Midpoint method is able to recycle old function evaluations
  through this factor of 3.

  Romberg integration converges quite fast by cancelling one error term in the
  taylor series expansion of $f$ with each examined term. If your function is
  /not/ smooth this may cause you trouble, and you may want to investigate a
  lower-order method.

  ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use."
  ([f a b] (open-sequence f a b {}))
  ([f a b {:keys [n] :or {n 1} :as opts}]
   {:pre [(number? n)]}
   (-> (qm/midpoint-sequence f a b opts)
       (ir/richardson-sequence 3 2 2))))

(defn closed-sequence
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the closed interval $[a, b]$ by applying Richardson extrapolation to
  successive integral estimates from the Trapezoid rule.

  Returns estimates formed by combining $n, 2n, 4n, ...$ slices, geometrically
  increasing by a factor of 2 with each estimate.

  Romberg integration converges quite fast by cancelling one error term in the
  taylor series expansion of $f$ with each examined term. If your function is
  /not/ smooth this may cause you trouble, and you may want to investigate a
  lower-order method.

  ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use."
```

```
  ([f a b] (closed-sequence f a b {}))
  ([f a b {:keys [n] :or {n 1} :as opts}]
   {:pre [(number? n)]}
   (-> (qt/trapezoid-sequence f a b opts)
       (ir/richardson-sequence 2 2 2))))

(defn romberg-sequence
  "Higher-level abstraction over `closed-sequence` and `open-sequence`. Identical
  to those functions (see their docstrings), but internally chooses either
  implementation based on the interval specified inside of `opts`.

  Defaults to the same behavior as `open-sequence`."
  ([f a b] (romberg-sequence f a b {}))
  ([f a b opts]
   (let [seq-fn (if (qc/closed?
                     (qc/interval opts))
                  closed-sequence
                  open-sequence)]
     (seq-fn f a b opts))))

(qc/defintegrator open-integral
  "Returns an estimate of the integral of `f` over the open interval $(a, b)$
  generated by applying Richardson extrapolation to successive integral
  estimates from the Midpoint rule.

  Considers $1, 3, 9 ... 3^n$ windows into $(a, b)$ for each successive
  estimate.

  Optionally accepts `opts`, a dict of optional arguments. All of these get
  passed on to `us/seq-limit` to configure convergence checking.

  See `open-sequence` for more information about Romberg integration, caveats
  that might apply when using this integration method and information on the
  optional args in `opts` that customize this function's behavior."
  :area-fn qm/single-midpoint
  :seq-fn open-sequence)

(qc/defintegrator closed-integral
  "Returns an estimate of the integral of `f` over the closed interval $[a, b]$
  generated by applying Richardson extrapolation to successive integral
```

```
  estimates from the Trapezoid rule.

  Considers $1, 2, 4 ... 2^n$ windows into $[a, b]$ for each successive
  estimate.

  Optionally accepts `opts`, a dict of optional arguments. All of these get
  passed on to `us/seq-limit` to configure convergence checking.

  See `closed-sequence` for more information about Romberg integration, caveats
  that might apply when using this integration method and information on the
  optional args in `opts` that customize this function's behavior."
  :area-fn qt/single-trapezoid
  :seq-fn closed-sequence)
```

4.6 Milne's Rule

```
(ns quadrature.milne
  (:require [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.midpoint :as qm]
            [quadrature.interpolate.richardson :as ir]
            [quadrature.util.stream :as us]))
```

This numerical integration method is an open Newton-Cotes formula; for each integral slice, Milne's rule samples three interior points (not the endpoints!) and combines them into an area estimate for this slice using the following formula:

$$\frac{4h}{3}(2f_1 - f_2 + 2f_3)$$

Given a window of (a, b) and a "step size" of $h = \frac{b-a}{3}$. The point f_i is the point i steps into the window.

There is a simpler way to understand this! Milne's method is, in fact, just the midpoint method (see `midpoint.cljc`), subject to a single refinement of "Richardson extrapolation".

The test namespace contains a symbolic proof that the Richardson-extrapolated Midpoint method is equivalent to using the formula above to calculate Milne's rule directly.

```
(defn milne-sequence
```

```
  "Returns a (lazy) sequence of successively refined estimates of the integral of
  ‘f‘ over the open interval $(a, b)$ using Milne's rule.

  Milne's rule is equivalent to the midpoint method subject to one refinement of
  Richardson extrapolation.

  Returns estimates with $n, 2n, 4n, ...$ slices, geometrically increasing by a
  factor of 2 with each estimate.

  ## Optional arguments:

  If supplied, ‘:n‘ (default 1) specifies the initial number of slices to use.

  NOTE: the Midpoint method is able to reuse function evaluations as its windows
  narrow /only/ when increasing the number of integration slices by 3. Milne's
  method increases the number of slices geometrically by a factor of 2 each
  time, so it will never hit the incremental path. You may want to memoize your
  function before calling ‘milne-sequence‘."
  ([f a b] (milne-sequence f a b {:n 1}))
  ([f a b {:keys [n] :or {n 1} :as opts}]
   {:pre [(number? n)]}
   (-> (qm/midpoint-sequence f a b (assoc opts :n (us/powers 2 n)))
       (ir/richardson-column 1 2 2 2))))

(qc/defintegrator integral
  "Returns an estimate of the integral of ‘f‘ over the open interval $(a, b)$
  using Milne's rule with $1, 2, 4 ... 2^n$ windows for each estimate.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘milne-sequence‘ for more information about Milne's rule, caveats that
  might apply when using this integration method and information on the optional
  args in ‘opts‘ that customize this function's behavior."
  :area-fn (comp first milne-sequence)
  :seq-fn milne-sequence)
```

4.7 Bulirsch-Stoer Integration

```
(ns quadrature.bulirsch-stoer
```

```
  (:require [quadrature.interpolate.polynomial :as poly]
            [quadrature.interpolate.rational :as rat]
            [quadrature.common :as qc
             #?@(:cljs [:include-macros true])]
            [quadrature.midpoint :as mid]
            [quadrature.trapezoid :as trap]
            [sicmutils.generic :as g]
            [sicmutils.util :as u]
            [quadrature.util.stream :as us]))
```

This quadrature method comes from the scmutils package that inspired this library.

The idea is similar to Romberg integration:

- use some simpler quadrature method like the Midpoint or Trapezoid method to approximate an integral with a successively larger number of integration slices
- Fit a curve to the pairs $(h, f(h))$, where h is the width of an integration slice and f is the integral estimator
- Use the curve to extrapolate forward to $h = 0$.

Romberg integration does this by fitting a polynomial to a geometric series of slices - $1, 2, 4\ldots$, for example - using Richardson extrapolation.

The Bulirsch-Stoer algorithm is exactly this, but:

- using rational function approximation instead of polynomial
- the step sizes increase like $2, 3, 4, 6, 8...2n_{i-2}$ by default

Here are the default step sizes:

```
(def bulirsch-stoer-steps
  (interleave
   (us/powers 2 2)
   (us/powers 2 3)))
```

The more familiar algorithm named "Bulirsch-Stoer" applies the same ideas to the solution of ODEs, as described on Wikipedia. scmutils adapted this into the methods you see here.

NOTE - The Wikipedia page states that "Hairer, Nørsett & Wanner (1993, p. 228), in their discussion of the method, say that rational extrapolation in this case is nearly never an improvement over polynomial interpolation (Deuflhard 1983)."

We can do this too! Passing `{:bs-extrapolator :polynomial}` enables polynomial extrapolation in the sequence and integration functions implemented below.

4.7.1 Even Power Series

One more detail is important to understand. You could apply the ideas above to any function that approximates an integral, but this namespace focuses on accelerating the midpoint and trapezoid methods.

As discussed in `midpoint.cljc` and `trapezoid.cljc`, the error series for these methods has terms that fall off as even powers of the integration slice width:

$$1/h^2, 1/h^4, ...$$

$$1/(h^2)1/(h^2)^2, ...$$

This means that the rational function approximation needs to fit the function to points of the form

$$(h^2, f(h))$$

to take advantage of the acceleration. This trick is baked into Richardson extrapolation through the ability to specify a geometric series. `richardson_test.cljc` shows that Richardson extrapolation is indeed equivalent to a polynomial fit using $h^2 \ldots$ the idea here is the same.

The following two functions generate a sequence of NON-squared h slice widths. `bs-sequence-fn` below squares each entry.

```
(defn- slice-width [a b]
  (let [width (- b a)]
    (fn [n] (/ width n))))

(defn- h-sequence
  "Defines the sequence of slice widths, given a sequence of ‘n‘ (number of
  slices) in the interval $(a, b)$."
  ([a b] (h-sequence a b bulirsch-stoer-steps))
```

```
  ([a b n-seq]
   (map (slice-width a b) n-seq)))
```

4.7.2 Bulirsch-Stoer Estimate Sequences

The next group of functions generates `open-sequence` and `closed-sequence` methods, analagous to all other quadrature methods in the library.

```
(defn- extrapolator-fn
  "Allows the user to specify polynomial or rational function extrapolation via
  the ‘:bs-extrapolator‘ option."
  [opts]
  (if (= :polynomial (:bs-extrapolator opts))
    poly/modified-neville
    rat/modified-bulirsch-stoer))

(defn- bs-sequence-fn
  "Accepts some function (like ‘mid/midpoint-sequence‘) that returns a sequence of
  successively better estimates to the integral, and returns a new function with
  interface ‘(f a b opts)‘ that accelerates the sequence with either

  - polynomial extrapolation
  - rational function extrapolation

  By default, The ‘:n‘ in ‘opts‘ (passed on to ‘integrator-seq-fn‘) is set to
  the sequence of step sizes suggested by Bulirsch-Stoer,
  ‘bulirsch-stoer-steps‘."
  [integrator-seq-fn]
  (fn call
    ([f a b]
     (call f a b {:n bulirsch-stoer-steps}))
    ([f a b opts]
     {:pre [(not (number? (:n opts)))]}
     (let [{:keys [n] :as opts} (-> {:n bulirsch-stoer-steps}
                                    (merge opts))
           extrapolate (extrapolator-fn opts)
           square      (fn [x] (* x x))
           xs          (map square (h-sequence a b n))
           ys          (integrator-seq-fn f a b opts)]
       (-> (map vector xs ys)
           (extrapolate 0))))))
```

```
(def ^{:doc "Returns a (lazy) sequence of successively refined estimates of the
  integral of `f` over the closed interval $[a, b]$ by applying rational
  polynomial extrapolation to successive integral estimates from the Midpoint
  rule.

  Returns estimates formed from the same estimates used by the Bulirsch-Stoer
  ODE solver, stored in `bulirsch-stoer-steps`.

  ## Optional arguments:

  `:n`: If supplied, `n` (sequence) overrides the sequence of steps to use.

  `:bs-extrapolator`: Pass `:polynomial` to override the default rational
  function extrapolation and enable polynomial extrapolation using the modified
  Neville's algorithm implemented in `poly/modified-neville`."}
  open-sequence
  (bs-sequence-fn mid/midpoint-sequence))

(def ^{:doc "Returns a (lazy) sequence of successively refined estimates of the
  integral of `f` over the closed interval $[a, b]$ by applying rational
  polynomial extrapolation to successive integral estimates from the Trapezoid
  rule.

  Returns estimates formed from the same estimates used by the Bulirsch-Stoer
  ODE solver, stored in `bulirsch-stoer-steps`.

  ## Optional arguments:

  `:n`: If supplied, `:n` (sequence) overrides the sequence of steps to use.

 `:bs-extrapolator`: Pass `:polynomial` to override the default rational
  function extrapolation and enable polynomial extrapolation using the modified
  Neville's algorithm implemented in `poly/modified-neville`."}
  closed-sequence
  (bs-sequence-fn trap/trapezoid-sequence))
```

4.7.3 Integration API

Finally, two separate functions that use the sequence functions above to converge quadrature estimates.

```
(qc/defintegrator open-integral
  "Returns an estimate of the integral of ‘f‘ over the open interval $(a, b)$
  generated by applying rational polynomial extrapolation to successive integral
  estimates from the Midpoint rule.

  Considers successive numbers of windows into $(a, b)$ specified by
  ‘bulirsch-stoer-steps‘.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘open-sequence‘ for more information about Bulirsch-Stoer quadrature,
  caveats that might apply when using this integration method and information on
  the optional args in ‘opts‘ that customize this function’s behavior."
  :area-fn mid/single-midpoint
  :seq-fn open-sequence)

(qc/defintegrator closed-integral
  "Returns an estimate of the integral of ‘f‘ over the closed interval $[a, b]$
  generated by applying rational polynomial extrapolation to successive integral
  estimates from the Trapezoid rule.

  Considers successive numbers of windows into $[a, b]$ specified by
  ‘bulirsch-stoer-steps‘.

  Optionally accepts ‘opts‘, a dict of optional arguments. All of these get
  passed on to ‘us/seq-limit‘ to configure convergence checking.

  See ‘closed-sequence‘ for more information about Bulirsch-Stoer quadrature,
  caveats that might apply when using this integration method and information on
  the optional args in ‘opts‘ that customize this function’s behavior."
  :area-fn trap/single-trapezoid
  :seq-fn closed-sequence)
```

4.7.4 References:

- Press, Numerical Recipes, section 16.4: `http://phys.uri.edu/nigh/NumRec/bookfpdf/f16-4.pdf`
- Wikipedia: `https://en.wikipedia.org/wiki/Bulirsch%E2%80%93Stoer_algorithm`

5 Combinators

5.1 Variable Substitutions

Implemented as functional wrappers that take an integrator and return a modified integrator.

```
(ns quadrature.substitute
  "## U Substitution and Variable Changes

  This namespace provides implementations of functions that accept an
  ‘integrator‘ and perform a variable change to address some singularity, like
  an infinite endpoint, in the definite integral.

  The strategies currently implemented were each described by Press, et al. in
  section 4.4 of ['Numerical
  Recipes'](http://phys.uri.edu/nigh/NumRec/bookfpdf/f4-4.pdf)."
  (:require [clojure.core.match :refer [match]]
            [quadrature.common :as qc]))
```

5.1.1 Infinite Endpoints

This first function, `infinitize`, transforms some integrator into a new integrator with the same interface that can handle an infinite endpoint.

This implementation can only handle one endpoint at a time, and, the way it's written, both endpoints have to have the same sign. For an easier interface to this transformation, see `infinite/evaluate-infinite-integral` in `infinite.cljc`.

```
(defn infinitize
  "Performs a variable substitution targeted at turning a single infinite endpoint
  of an improper integral evaluation an (open) endpoint at 0 by applying the
  following substitution:

  $$u(t) = {1 \\over t}$$ $$du = {-1 \\over t^2}$$

  This works when the integrand ‘f‘ falls off at least as fast as $1 \\over t^2$
  as it approaches the infinite limit.

  The returned function requires that ‘a‘ and ‘b‘ have the same sign, ie:
```

```
$$ab > 0$$

Transform the bounds with $u(t)$, and cancel the negative sign by changing
their order:

$$\\int_{a}^{b} f(x) d x=\\int_{1 / b}^{1 / a} \\frac{1}{t^{2}} f\\left(\\frac{1}{t}\

References:

- Mathworld, \"Improper Integral\": https://mathworld.wolfram.com/ImproperIntegral.ht
- Press, Numerical Recipes, Section 4.4: http://phys.uri.edu/nigh/NumRec/bookfpdf/f4-
[integrate]
(fn call
  ([f a b] (call f a b {}))
  ([f a b opts]
   {:pre [(not
           (and (qc/infinite? a)
                (qc/infinite? b)))]}
   (let [f' (fn [t]
              (/ (f (/ 1.0 t))
                 (* t t)))
         a' (if (qc/infinite? b) 0.0 (/ 1.0 b))
         b' (if (qc/infinite? a) 0.0 (/ 1.0 a))
         opts (qc/update-interval opts qc/flip)]
     (integrate f' a' b' opts)))))
```

5.1.2 Power Law Singularities

> "To deal with an integral that has an integrable power-law singularity at its lower limit, one also makes a change of variable." (Press, p138)

A "power-law singularity" means that the integrand diverges as $(x-a)^{-\gamma}$ near $x = a$.

We implement the following identity (from Press) if the singularity occurs at the lower limit:

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(t^{\frac{1}{1-\gamma}} + a\right) dt \quad (b > a)$$

And this similar identity if the singularity occurs at the upper limit:

;;

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f\left(b - t^{\frac{1}{1-\gamma}}\right) dt \quad (b > a)$$

If you have singularities at both sides, divide the interval at some interior breakpoint, take separate integrals for both sides and add the values back together.

```
(defn- inverse-power-law
  "Implements a change of variables to address a power law singularity at the
  lower or upper integration endpoint.

  An \"inverse power law singularity\" means that the integrand diverges as

  $$(x - a)^{-\\gamma}$$

  near $x=a$. Passing true for ‘lower?‘ to specify a singularity at the lower
  endpoint, false to signal an upper-endpoint singularity.

  References:

  - Mathworld, \"Improper Integral\": https://mathworld.wolfram.com/ImproperIntegral.ht
  - Press, Numerical Recipes, Section 4.4: http://phys.uri.edu/nigh/NumRec/bookfpdf/f4-
  - Wikipedia, \"Finite-time Singularity\": https://en.wikipedia.org/wiki/Singularity_(
  "
  [integrate gamma lower?]
  {:pre [(<= 0 gamma 1)]}
  (fn call
    ([f a b] (call f a b {}))
    ([f a b opts]
     (let [inner-pow (/ 1 (- 1 gamma))
           gamma-pow (* gamma inner-pow)
           a' 0
           b' (Math/pow (- b a) (- 1 gamma))
           t->t' (if lower?
                   (fn [t] (+ a (Math/pow t inner-pow)))
                   (fn [t] (- b (Math/pow t inner-pow))))
           f' (fn [t] (* (Math/pow t gamma-pow)
                         (f (t->t' t))))]
       (-> (integrate f' a' b' opts)
```

```
          (update-in [:result] (partial * inner-pow)))))))

(defn inverse-power-law-lower [integrate gamma]
  (inverse-power-law integrate gamma true))

(defn inverse-power-law-upper [integrate gamma]
  (inverse-power-law integrate gamma false))
```

5.1.3 Inverse Square Root singularities

The next two functions specialize the `inverse-power-law-*` functions to the common situation of an inverse power law singularity.

```
(defn inverse-sqrt-lower
  "Implements a change of variables to address an inverse square root singularity
  at the lower integration endpoint. Use this when the integrand diverges as

  $$1 \\over {\\sqrt{x - a}}$$

  near the lower endpoint $a$."
  [integrate]
  (fn call
    ([f a b] (call f a b {}))
    ([f a b opts]
     (let [f' (fn [t] (* t (f (+ a (* t t)))))]
       (-> (integrate f' 0 (Math/sqrt (- b a)) opts)
           (update-in [:result] (partial * 2)))))))

(defn inverse-sqrt-upper
  "Implements a change of variables to address an inverse square root singularity
  at the upper integration endpoint. Use this when the integrand diverges as

  $$1 \\over {\\sqrt{x - b}}$$

  near the upper endpoint $b$."
  [integrate]
  (fn call
    ([f a b] (call f a b {}))
    ([f a b opts]
     (let [f' (fn [t] (* t (f (- b (* t t)))))]
       (-> (integrate f' 0 (Math/sqrt (- b a)) opts)
           (update-in [:result] (partial * 2)))))))
```

5.1.4 Exponentially Diverging Endpoints

From Press, section 4.4: "Suppose the upper limit of integration is infinite, and the integrand falls off exponentially. Then we want a change of variable that maps

$$\exp -x dx$$

into +- dt (with the sign chosen to keep the upper limit of the new variable larger than the lower limit)."

The required identity is:

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t}$$

```
(defn exponential-upper
  "Implements a change of variables to address an exponentially diverging upper
  integration endpoint. Use this when the integrand diverges as $\\exp{x}$ near
  the upper endpoint $b$."
  [integrate]
  (fn call
    ([f a b] (call f a b {}))
    ([f a b opts]
     {:pre [(qc/infinite? b)]}
     (let [f' (fn [t] (* (- (Math/log t))
                         (/ 1 t)))
           opts (qc/update-interval opts qc/flip)]
       (integrate f 0 (Math/exp (- a)) opts)))))
```

5.2 Improper Integrals

A template for a combinator that enables infinite endpoints on any integrator, using variable substitution on an appropriate, tunable range.

```
(ns quadrature.infinite
  (:require [clojure.core.match :refer [match]]
            [quadrature.common :as qc]
            [quadrature.substitute :as qs]))
```

This namespace holds an implementation of an "improper" integral combinator (for infinite endpoints) usable with any quadrature method in the library.

The implementation was inspired by `evaluate-improper-integral` in `numerics/quadrature/quadrature.scm` file in the scmutils package.

5.2.1 Overview

To evaluate an improper integral with an infinite endpoint, the `improper` combinator applies an internal change of variables.

$$u(t) = \frac{1}{t}$$

$$du = \frac{-1}{t^2}$$

This has the effect of mapping the endpoints from a, b to $\frac{1}{b}, \frac{1}{a}$. Here's the identity we implement:

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt$$

This is implemented by `substitute/infinitize`.

The variable change only works as written when both endpoints are of the same sign; so, internally, `improper` only applies the variable change to the segment of a, b from `##-Inf => (- :infinite-breakpoint)` and `:infinite-breakpoint -> ##Inf`, where `:infinite-breakpoint` is an argument the user can specify in the returned integrator's options map.

Any segment of a, b *not* in those regions is evaluated normally.

NOTE: The ideas in this namespace could be implemented for other variable substitutions (see `substitute.cljc`) that only need to apply to certain integration intervals. The code below automatically cuts the range (a, b) to accomodate this for the particular variable change we've baked in, but there is a more general abstraction lurking.

5.2.2 Implementation

```
(defn improper
  "Accepts:

  - An ‘integrator‘ (function of ‘f‘, ‘a‘, ‘b‘ and ‘opts‘)
  - ‘a‘ and ‘b‘, the endpoints of an integration interval, and
  - (optionally) ‘opts‘, a dict of integrator-configuring options

  And returns a new integrator that’s able to handle infinite endpoints. (If you
```

```
don't specify `##-Inf` or `##Inf`, the returned integrator will fall through
to the original `integrator` implementation.)

All `opts` will be passed through to the supplied `integrator`.

## Optional arguments relevant to `improper`:

`:infinite-breakpoint`: If either `a` or `b` is equal to `##Inf` or `##-Inf`,
this function will internally perform a change of variables on the regions
from:

`(:infinite-breakpoint opts) => ##Inf`

or

`##-Inf => (- (:infinite-breakpoint opts))`

using $u(t) = {1 \\over t}$, as described in the `infinitize` method of
`substitute.cljc`. This has the effect of mapping the infinite endpoint to an
open interval endpoint of 0.

Where should you choose the breakpoint? According to Press in Numerical
Recipes, section 4.4: \"At a sufficiently large positive value so that the
function funk is at least beginning to approach its asymptotic decrease to
zero value at infinity.\"

References:

- Press, Numerical Recipes (p138), Section 4.4: http://phys.uri.edu/nigh/NumRec/booki
[integrator]
(fn rec
  ([f a b] (rec f a b {}))
  ([f a b opts]
   (let [{:keys [infinite-breakpoint] :as opts} (-> {:infinite-breakpoint 1}
                                                    (merge opts))
         call (fn [integrate l r interval]
                (let [m (qc/with-interval opts interval)]
                  (let [result (integrate f l r m)]
                    (:result result))))
         ab-interval   (qc/interval opts)
```

```
    integrate     (partial call integrator)
    inf-integrate (partial call (qs/infinitize integrator))
    r-break       (Math/abs infinite-breakpoint)
    l-break       (- r-break)]
(match [[a b]]
       [(:or [##-Inf ##-Inf] [##Inf ##Inf])]
       {:converged? true
        :terms-checked 0
        :result 0.0}

       [(:or [_ ##-Inf] [##Inf _])]
       (-> (rec f b a opts)
           (update-in [:result] -))

       ;; Break the region up into three pieces: a central closed core
       ;; and two open endpoints where we create a change of variables,
       ;; letting the boundary go to infinity. We use an OPEN interval on
       ;; the infinite side.
       [[##-Inf ##Inf]]
       (let [-inf->l (inf-integrate a l-break qc/open-closed)
             l->r    (integrate     l-break r-break qc/closed)
             r->+inf (inf-integrate r-break b qc/closed-open)]
         {:converged? true
          :result (+ -inf->l l->r r->+inf)})

       ;; If `b` lies to the left of the negative breakpoint, don't cut.
       ;; Else, cut the integral into two pieces at the negative
       ;; breakpoint and variable-change the left piece.
       [[##-Inf _]]
       (if (<= b l-break)
         (inf-integrate a b ab-interval)
         (let [-inf->l (inf-integrate a l-break qc/open-closed)
               l->b    (integrate     l-break b (qc/close-l ab-interval))]
           {:converged? true
            :result (+ -inf->l l->b)}))

       ;; If `a` lies to the right of the positive breakpoint, don't cut.
       ;; Else, cut the integral into two pieces at the positive breakpoint
       ;; and variable-change the right piece.
       [[_ ##Inf]]
```

```
(if (>= a r-break)
  (inf-integrate a b ab-interval)
  (let [a->r    (integrate     a r-break (qc/close-r ab-interval))
        r->+inf (inf-integrate r-break b qc/closed-open)]
    {:converged? true
     :result (+ a->r r->+inf)}))

;; This is a lot of machinery to use with NON-infinite endpoints;
;; but for completeness, if you reach this level the fn will attempt
;; to integrate the full range directly using the original
;; integrator.
:else (integrator f a b opts))))))
```

5.2.3 Suggestions for Improvement

The current implementation does not pass convergence information back up the line! Ideally we would merge results by:

- Adding results
- combining `:converged?` entries with `and`
- retaining all other keys

5.3 Adaptive Quadrature

```
(ns quadrature.adaptive
  (:require [quadrature.common :as qc]
            [quadrature.util.aggregate :as ua]))
```

This namespace holds an implementation of "adaptive quadrature" usable with any quadrature method in the library.

The implementation was inspired by the `numerics/quadrature/rational.scm` file in the scmutils package. In that library, adaptive quadrature was special-cased to the Bulirsch-Stoer algorithm, ported in `bulirsch_stoer.cljc`.

5.3.1 Overview

Most of the integrators in `quadrature` work by successively refining an integration interval a, b down into evenly-spaced integration slices. Some functions are very well behaved in some regions, and then oscillate wildly in others.

Adaptive quadrature methods partition a, b into intervals of different sizes, allowing the integrator to drill in more closely to areas that need attention.

The `adaptive` implementation below works like this:

- use a wrapped `integrate` function on the full a, b, capping the iterations at some configurable limit (~*adaptive-maxterms*~below)
- If `integrate` fails to converge, split a, b into two intervals and attempt to converge both sides using `integrate`
- Continue this process until convergence, or until the interval becomes small enough to fail the test in `common/narrow-slice?`, at which point `integrate` returns an estimate for a single slice.

The `*adaptive-maxterms*` variable is dynamic, which means you can adjust the behavior of `adaptive` by supplying the `:maxterms` option directly, or wrapping your call in `(binding [*adaptive-maxterms* 8] ,,,)`.

```
(def ^:dynamic *adaptive-maxterms* 10)
```

5.3.2 Fuzzy Midpoints

Symmetric intervals like $-1, 1$ often show up with integrands with singularities right at the center of the midpoint. For this reason, `adaptive` is able to customize its splitting behavior using the `*neighborhood-width*` dynamic variable.

By default, when partitioning an interval, `adaptive` will choose an interval within 5% of the midpoint. Override this behavior with the `:adaptive-neighborhood-width` key in the options dict, or by binding this dynamic variable.

```
(def ^:dynamic *neighborhood-width* 0.05)

(defn- split-point
  "Returns a point within‘fuzz-factor‘ of the midpoint of the interval $[a, b]$.
  ‘fuzz-factor‘ defaults to 0 (ie, ‘split-point‘ returns the midpoint)."
  ([a b] (split-point a b 0))
  ([a b fuzz-factor]
   {:pre [(>= fuzz-factor 0)
          (< fuzz-factor 1)]}
   (let [width  (- b a)
         offset (if (zero? fuzz-factor)
```

```
                0.5
                (+ 0.5 (* fuzz-factor (dec (rand 2.0)))))]
    (+ a (* offset width)))))
```

5.3.3 Main Implementation

The implementation below takes *two* integrate functions, not the one described above. This allows us to handle open and closed intervals, instead of introducing open endpoints at every subdivision. All internal intervals that don't touch an open endpoint are considered closed.

```
(defn- fill-defaults
  "Populates the supplied ‘opts‘ dictionary with defaults required by ‘adaptive‘.
  Two of these have values controlled by dynamic variables in ‘adaptive.cljc‘."
  [opts]
  (merge {:maxterms *adaptive-maxterms*
          :adaptive-neighborhood-width *neighborhood-width*
          :interval qc/open}
         opts))

(defn adaptive
  "Accepts one or two 'integrators', ie, functions of:

  - ‘f‘: some integrand
  - ‘a‘ and ‘b‘: the lower and upper endpoints of integration
  - ‘opts‘, a dictionary of configuration options

  And returns a new integrator that adaptively subdivides the region $a, b$ into
  intervals if integration fails to converge. If two integrators are supplied,
  the first is applied to any interval that's not explicitly closed on both
  ends, and the second integrator is applied to explicitly closed intervals.
  This behavior depends on the interval set in the supplied ‘opts‘ dictionary.

  All ‘opts‘ will be passed through to the supplied ‘integrate‘ functions.

  ## Optional arguments relevant to ‘adaptive‘:

  ‘:maxterms‘: defaults to ‘*adaptive-maxterms*‘. This is passed to the
  underlying integrators, and determines how long each interval attempts to
  converge before a subdivision.
```

```
‘:adaptive-neighborhood-width‘: When dividing an interval, the split point
will be within this factor of the midpoint. Set ‘:adaptive-neighborhood-width‘
to 0 for deterministic splitting."
([integrator] (adaptive integrator integrator))
([open-integrator closed-integrator]
 (fn rec
   ([f a b] (rec f a b {}))
   ([f a b opts]
    (let [opts      (fill-defaults opts)
          integrate (fn [l r interval]
                      (if (qc/closed? interval)
                        (closed-integrator f l r opts)
                        (open-integrator f l r opts)))]
      (loop [stack [[a b (:interval opts)]]
             sum   (ua/kahan-sum)
             iteration 0]
        (if (empty? stack)
          {:converged? true
           :iterations iteration
           :result (first sum)}
          (let [[l r interval] (peek stack)
                remaining      (pop stack)
                {:keys [converged? result]} (integrate l r interval)]
            (if converged?
              (recur remaining
                     (ua/kahan-sum sum result)
                     (inc iteration))
              (let [midpoint (split-point l r (:adaptive-neighborhood-width opts))]
                (recur (conj remaining
                             [midpoint r (qc/close-l interval)]
                             [l midpoint (qc/close-r interval)])
                       sum
                       (inc iteration)))))))))))
```

5.3.4 Suggestions for Improvement

1. Iteration Limit

   `adaptive` runs until it completes, with no facility available to bail out of computation. An iteration limit would be a great addition... but it won't be efficient without some way of prioritizing high-error

subintervals for refinement, as discussed next.

2. Priority Queue on Error

   Another nice upgrade would be a version of `adaptive` that is able to return an actual sequence of successive refinements. to the estimate. The current implementation uses an internal stack to track the subdivided intervals it needs to evaluate. If the integrator was able to provide an error estimate, we could instead use a priority queue, prioritized by error.

   `adaptive-sequence` could then return a sequence where every element processes a single refinement. Even without this upgrade, the priority queue idea would allow the estimate to converge quickly and be more accurate if we bailed out at some max number of iterations.

   This article holds a related implementation.

5.3.5 References

- SCMUtils Refman: `https://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt`
- Wikipedia, "Adaptive Simpson's Method": `https://en.wikipedia.org/wiki/Adaptive_Simpson%27s_method`

#+end$_{\text{src}}$

6 Conclusion

Conclusion coming.

7 Appendix: Common Code

```
(ns quadrature.common
  "Implements utilities shared by all integrators, for example:

  - code to wrap a sequence of progressively better estimates in a common ‘integrator‘
  - data structures implementing various integration intervals."
  #?(:cljs (:refer-clojure :exclude [infinite?]
                           :rename {infinite? core-infinite?}))
  (:require [quadrature.util.stream :as us]
            [taoensso.timbre :as log]))
```

This dynamic variable holds the default "roundoff cutoff" used by all integrators to decide when to return an estimate based on a single slice, vs attempting to converge a sequence of progressively finer estimates. When this condition is satisfied:

$$|b - a|/|a| + |b| <= \ cutoff$$

An integrator will estimate a single slice directly. Else, it will attempt to converge a sequence.

```
(def ^:dynamic *roundoff-cutoff* 1e-14)
```

NOTE - we don't have an interface yet to bind this dynamic variable. bind it manually to modify the cutoff for a specific call to some integrator:

```
(binding [*roundoff-cutoff* 1e-6]
  (integrate f a b))
```

7.1 Intervals

Implementations of the various intervals used by the adaptive integral interface. By default, integration endpoints are considered *open*.

```
(def open        [::open ::open])
(def closed      [::closed ::closed])
(def open-closed [::open ::closed])
(def closed-open [::closed ::open])
(def infinities #{##Inf ##-Inf})
(def infinite?
  #?(:cljs core-infinite?
     :clj (comp boolean infinities)))

(defn closed?
  "Returns true if the argument represents an explicit ‘closed‘ interval, false
  otherwise."
  [x] (= x closed))
(def open? (complement closed?))
```

These functions modify an interval by opening or closing either of its endpoints.

```
(defn close-l [[_ r]] [::closed r])
(defn close-r [[l _]] [l ::closed])
(defn open-l [[_ r]] [::open r])
(defn open-r [[l _]] [l ::open])
(defn flip [[l r]] [r l])

(defn interval
  "Extracts the interval (or ‘open‘ as a default) from the supplied integration
  options dict."
  [opts]
  (get opts :interval open))

(defn with-interval
  "Sets the specified interval to a key inside the suppled ‘opts‘ map of arbitrary
  integration options."
  [opts interval]
  (assoc opts :interval interval))

(defn update-interval
  "Accepts:

  - a dictionary of arbitrary options
  - one of the 4 interval modification functions

  and returns a dict of options with ‘f‘ applied to the contained interval (or
  ‘open‘ if no interval is set).
  "
  [opts f]
  (let [k :interval]
    (assoc opts k (f (interval opts)))))
```

7.2 Common Integration Interface

The following two functions define a shared interface that integration namespaces can use to create an "integrator" from:

- a fn that can estimate the area of a single integration slice, and
- a fn that can return a sequence of progressively finer estimates.

The first function is called in the case that the integration range (a, b)

(open or closed) is too fine for subdivision. The second function takes over in all other (most!) cases.

```
(defn- narrow-slice?
  "Returns true if the range $[a, b]$ is strip narrow enough to pass the following
  test:

  |b - a| / |a| + |b| <= `cutoff`

  False otherwise. This inequality measures how close the two floating point
  values are, scaled by the sum of their magnitudes."
  [a b cutoff]
  (let [sum (+ (Math/abs a)
               (Math/abs b))]
    (or (<= sum cutoff)
        (<= (Math/abs (- b a))
            (* cutoff sum)))))

(defn make-integrator-fn
  "Generates an `integrator` function from two functions with the following
  signatures and descriptions:

  - `(area-fn f a b)` estimates the integral of `f` over the interval `(a, b)`
  with no subdivision, nothing clever at all.

  - `(seq-fn f a b opts)` returns a sequence of successively refined estimates
  of the integral of `f` over `(a, b)`. `opts` can contain kv pairs that
  configure the behavior of the sequence function (a sequence of the number of
  integration slices to use, for example.)

  The returned function has the signature:

  `(f a b opts)`

  All `opts` are passed on to `seq-fn`, /and/ to `us/seq-limit` internally,
  where the options configure the checks on sequence convergence."
  [area-fn seq-fn]
  (fn call
    ([f a b] (call f a b {}))
    ([f a b {:keys [roundoff-cutoff]
```

```
            :or {roundoff-cutoff *roundoff-cutoff*}
            :as opts}]
    (if (narrow-slice? a b roundoff-cutoff)
      (do (log/info "Integrating narrow slice: " a b)
          {:converged? true
           :terms-checked 1
           :result (area-fn f a b)})
      (-> (seq-fn f a b opts)
          (us/seq-limit opts))))))

(defn- name-with-attributes
  "Taken from `clojure.tools.macro/name-with-attributes`.

  Handles optional docstrings and attribute maps for a name to be defined in a
  list of macro arguments. If the first macro argument is a string, it is added
  as a docstring to name and removed from the macro argument list. If afterwards
  the first macro argument is a map, its entries are added to the name's
  metadata map and the map is removed from the macro argument list. The return
  value is a vector containing the name with its extended metadata map and the
  list of unprocessed macro arguments."
  ([name body] (name-with-attributes name body {}))
  ([name body meta]
   (let [[docstring body] (if (string? (first body))
                            [(first body) (next body)]
                            [nil body])
         [attr body]      (if (map? (first body))
                            [(first body) (next body)]
                            [{} body])
         attr             (merge meta attr)
         attr             (if docstring
                            (assoc attr :doc docstring)
                            attr)
         attr             (if (meta name)
                            (conj (meta name) attr)
                            attr)]
     [(with-meta name attr) body])))

(defmacro defintegrator
  "Helper macro for defining integrators."
  [sym & body]
```

```
  (let [meta       {:arglists (list 'quote '([f a b] [f a b opts]))}
        [sym body] (name-with-attributes sym body meta)
        {:keys [area-fn seq-fn]} (apply hash-map body)]
    (assert seq-fn (str "defintegrator " sym ": seq-fn cannot be nil"))
    (assert area-fn (str "defintegrator " sym ": area-fn cannot be nil"))
    `(def ~sym
       (make-integrator-fn ~area-fn ~seq-fn))))
```

8 Appendix: Streams

```
(ns quadrature.util.stream
  "This namespace contains various standard sequences, as well as utilities for
  working with strict and lazy sequences."
  (:require [clojure.pprint :as pp]
            [sicmutils.generic :as g]
            [sicmutils.value :as v]))

(defn pprint
  "Realizes and pretty-prints `n` elements from the supplied sequence `xs`."
  [n xs]
  (doseq [x (take n xs)]
    (pp/pprint x)))

(defn powers
  "Returns an infinite sequence of `x * n^i`, starting with i == 0. `x` defaults
  to 1."
  ([n] (powers n 1))
  ([n x] (iterate #(* n %) x)))

(defn zeno
  "Returns an infinite sequence of x / n^i, starting with i == 0. `x` defaults to
  1."
  ([n] (zeno n 1))
  ([n x] (iterate #(/ % n) x)))

(defn scan
  "Returns a function that accepts a sequence `xs`, and performs a scan by:

  - Aggregating each element of `xs` into
  - the initial value `init`
```

```
  - transforming each intermediate result by ‘present‘ (defaults to ‘identity‘).

  The returned sequence contains every intermediate result, after passing it
  through ‘present‘ (not ‘init‘, though).

  This is what distinguishes a scan from a 'fold'; a fold would return the final
  result. A fold isn't appropriate for aggregating infinite sequences, while a
  scan works great.

  Arities:

  - the three-arity version takes ‘init‘ value, ‘f‘ fold function and ‘present‘.
  - the two arity version drops ‘init‘, and instead calls ‘f‘ with no arguments.
  The return value is the initial aggregator.
  - The 1-arity version only takes the ‘merge‘ fn and defaults ‘present‘ to
  ‘identity‘.
    "
  [f & {:keys [present init]
        :or {present identity}}]
  (let [init (or init (f))]
    (fn [xs]
      (->> (reductions f init xs)
           (map present)
           (rest)))))
```

8.1 Convergence Tests

This convergence tester comes from Gerald Sussman's "Abstraction in Numerical Methods".

We're planning on adding a number of these here and consolidating all of the available ideas about relative and maximum tolerance so we can share (and combine) them across different stream functions.

```
(defn- close-enuf?
  "relative closeness, transitioning to absolute closeness when we get
  significantly smaller than 1."
  [tolerance]
  (fn [h1 h2]
    (<= (g/abs (- h1 h2))
        (* 0.5 tolerance (+ 2 (g/abs h1) (g/abs h2))))))
```

I have a dream that a function like `seq-limit` could service most of the numerical methods in this library. Function minimization, root finding, definite integrals and numerical derivatives can all be expressed as successive approximations, with convergence tests (or other stopping conditions) checked and applied between iterations.

As of October 2020 we use this exclusively for various numerical integration routines. But it has more promise than this!

```
(defn seq-limit
  "Accepts a sequence, iterates through it and returns a dictionary of this form:

  {:converged? <boolean>
   :terms-checked <int>
   :result <sequence element>}

  ‘:converged?‘ is true if the sequence reached convergence by passing the tests
  described below, false otherwise.

  ‘:terms-checked‘ will be equal to the number of items examined in the
  sequence.

  ‘:result‘ holds the final item examined in the sequence.

  ## Optional keyword args:

  ‘:convergence-fn‘ user-supplied function of two successive elements in ‘xs‘
  that stops iteration and signals convergence if it returns true.

  ‘:fail-fn‘ user-supplied function of two successive elements in ‘xs‘ that
  stops iteration and signals NO convergence (failure!) if it returns true.

  ‘:minterms‘ ‘seq-limit‘ won’t return until at least this many terms from the
  sequence have been processed.

  ‘:maxterms‘ ‘seq-limit‘ will return (with ‘:converged? false‘) after
  processing this many elements without passing any other checks.

  ‘:tolerance‘ A combination of relative and absolute tolerance. defaults to
  ‘sqrt(machine epsilon)‘."
  ([xs] (seq-limit xs {}))
```

```
  ([xs {:keys [minterms
               maxterms
               tolerance
               convergence-fn
               fail-fn]
        :or {minterms       2
             tolerance      (Math/sqrt v/machine-epsilon)
             convergence-fn (close-enuf? tolerance)}}]
   (if (empty? xs)
     {:converged? false
      :terms-checked 0
      :result        nil}
     (let [stop? (if maxterms
                   (fn [i] (>= i maxterms))
                   (constantly false))]
       (loop [[x1 & [x2 :as more]] xs
              terms-checked 1]
         (if (empty? more)
           {:converged?    false
            :terms-checked terms-checked
            :result        x1}
           (let [terms-checked (inc terms-checked)
                 converged?    (convergence-fn x1 x2)]
             (if (and (>= terms-checked minterms)
                      (or converged?
                          (stop? terms-checked)))
               {:converged?    converged?
                :terms-checked terms-checked
                :result        x2}
               (recur more terms-checked)))))))))
```

9 Appendix: Aggregation

```
(ns quadrature.util.aggregate
  "Utilities for aggregating sequences.")
```

I learned about "Kahan's summation trick" from `rational.scm` in the `scmutils` package, where it shows up in the `sigma` function.

```
(defn kahan-sum
```

```
  "Implements a fold that tracks the summation of a sequence of floating point
  numbers, using Kahan's trick for maintaining stability in the face of
  accumulating floating point errors."
  ([] [0.0 0.0])
  ([[sum c] x]
   (let [y (- x c)
         t (+ sum y)]
     [t (- (- t sum) y)])))

(defn sum
  "Sums either:

  - a series `xs` of numbers, or
  - the result of mapping function `f` to `(range low high)`

  Using Kahan's summation trick behind the scenes to keep floating point errors
  under control."
  ([xs]
   (first
    (reduce kahan-sum [0.0 0.0] xs)))
  ([f low high]
   (sum (map f (range low high)))))

(defn scanning-sum
  "Returns every intermediate summation from summing either:

  - a series `xs` of numbers, or
  - the result of mapping function `f` to `(range low high)`

  Using Kahan's summation trick behind the scenes to keep floating point errors
  under control."
  ([xs]
   (->> (reductions kahan-sum [0.0 0.0] xs)
        (map first)
        (rest)))
  ([f low high]
   (scanning-sum
    (map f (range low high)))))
```