```
;; z(s) = sum(1..inf)(1/n^s)

(defun zeta-std (s &optional (inf 50))
  (let ((z 0))
    (dotimes (i inf)
      (setq z (+ z (/ 1 (expt (+ i 1) s)))))
    z))

;; http://www.physicsforums.com/showthread.php?t=535768

(defun zeta-euler-dirichlet (s &optional (inf 50))
  (defun expt-minus-one (n)
    (if (oddp n) -1 1))
  (* (/ 1 (- 1 (expt 2 (- 1 s))))
     (sum 0 inf (lambda (n)
                  (* (/ 1 (expt 2 (+ n 1)))
                     (sum 0 n (lambda (k) (* (expt-minus-one k) (comb n k) (expt (+ k 1) (- s))))))))))

;; http://mathworld.wolfram.com/RiemannZetaFunction.html
;; Valid for real(s) > 0

(defun zeta-dirichlet (s &optional (inf 50))
  (* (/ 1 (- 1 (expt 2 (- 1 s))))
     (sum 1 inf (lambda (n)
                  (* (expt -1 (- n 1))
                     (expt n (- s)))))))

(defun real-zeta-dirichlet-series-crit-line (tee &optional (inf 50))
  (- (* 2 (sum 1 inf (lambda (k)
                       (* (/ 1 (sqrt (* 2 k))) (cos (* (log (* 2 k)) tee))))))
     (sum 1 inf (lambda (n)
                  (* (expt -1 n)
                     (/ 1 (sqrt n))
                     (cos (* (log n) tee)))))))

(defun zeta-dirichlet-series (s &optional (inf 50))
  (- (* 2 (sum 1 inf (lambda (k)
                       (expt (* 2 k) (- s)))))
     (sum 1 inf (lambda (n)
                  (* (expt -1 n)
                     (expt n (- s)))))))

(defun zeta (s &optional (inf 100))
```

```
;;  (zeta-std s inf)
;;  (zeta-euler-dirichlet s inf)
  (zeta-dirichlet s inf)
)

(defun emit-zeta-along-crit-line (real-file imag-file n)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10)))
          (let ((v (zeta (+ (* (sqrt -1) tee) 1/2) 300)))
            (let ((r (realpart v))
                  (i (imagpart v)))
              (format sr "~a ~a~%" (float tee) r)
              (format si "~a ~a~%" (float tee) i))))))))


(defun emit-real-zeta-dirichlet-series-along-crit-line (real-file n)
  (with-open-file (sr real-file :direction :output)
    (dotimes (i n)
      (let ((tee (/ i 10)))
        (let ((r (real-zeta-dirichlet-series-crit-line tee 300)))
          (format sr "~a ~a~%" (float tee) r))))))

(defun emit-zeta-dirichlet-series-along-crit-line (real-file imag-file n)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10)))
          (let ((v (zeta-dirichlet-series (+ (* (sqrt -1) tee) 1/2) 1000)))
            (let ((r (realpart v))
                  (i (imagpart v)))
              (format sr "~a ~a~%" (float tee) r)
              (format si "~a ~a~%" (float tee) i))))))))

(defun emit-zeta-along-3/4-line (real-file imag-file n)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10)))
          (let ((v (zeta (+ (* (sqrt -1) tee) 3/4))))
            (let ((r (realpart v))
                  (i (imagpart v)))
              (format sr "~a ~a~%" (float tee) r)
              (format si "~a ~a~%" (float tee) i))))))))
```

```
(defun emit-std-zeta-along-line (real-file imag-file n line)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10)))
          (let ((v (zeta-std (+ (* (sqrt -1) tee) line))))
            (let ((r (realpart v))
                  (i (imagpart v)))
              (format sr "~a ~a~%" (float tee) r)
              (format si "~a ~a~%" (float tee) i))))))))

(defun emit-zeta (real-file imag-file n)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10))
              (epsilon-factor (- 1 1e-9)))
          (let ((z (complex (+ (* 1/4 (* epsilon-factor (cos tee))) 3/4)
                            (+ (* 1/4 (* epsilon-factor (sin tee))) (* .05 tee)))))
            (let ((v (zeta z)))
              (let ((r (realpart v))
                    (i (imagpart v)))
                (format sr "~a ~a~%" (float tee) r)
                (format si "~a ~a~%" (float tee) i)))))))))

(defun emit-real-zeta-along-crit-line-wave (real-file n)
  (with-open-file (sr real-file :direction :output :element-type 'binary)
    (write-wave sr (lambda (i)
                     (when (= (mod i 100) 0)
                       (print i))
                     (if (>= i n)
                         nil
                         (let ((tee (/ i 200))) ;; Was 10, 100, 500
                           (let ((v (zeta (+ (* (sqrt -1) tee) 1/2) 1000))) ;; Was not passed, 300
                             (let ((r (realpart v))
                                   (i (imagpart v)))
                               (* 2000 r))))))))) ;; Was 1000

(defun emit-real-imag-sum-zeta-along-crit-line-wave (real-file n)
  (with-open-file (sr real-file :direction :output :element-type 'binary)
    (write-wave sr (lambda (i)
                     (when (= (mod i 100) 0)
                       (print i))
                     (if (>= i n)
```

```
                        nil
                        (let ((tee (/ i 100))) ;; Was 10
                          (let ((v (zeta (+ (* (sqrt -1) tee) 1/2) 300))) ;; Was not passed
                            (let ((r (realpart v))
                                  (i (imagpart v)))
                              (* 1000 (+ r i)))))))))

(defun emit-real-zeta-along-3/4-line-wave (real-file n)
  (with-open-file (sr real-file :direction :output :element-type 'binary)
    (write-wave sr (lambda (i)
                     (when (= (mod i 100) 0)
                       (print i))
                     (if (>= i n)
                         nil
                         (let ((tee (/ i 100))) ;; Was 10
                           (let ((v (zeta (+ (* (sqrt -1) tee) 3/4))))
                             (let ((r (realpart v))
                                   (i (imagpart v)))
                               (* 1000 r)))))))))

(defun emit-real-zeta-along->1-line-wave (real-file n line)
  (with-open-file (sr real-file :direction :output :element-type 'binary)
    (write-wave sr (lambda (i)
                     (when (= (mod i 100) 0)
                       (print i))
                     (if (>= i n)
                         nil
                         (let ((tee (/ i 100))) ;; Was 10
                           (let ((v (zeta-std (+ (* (sqrt -1) tee) line))))
                             (let ((r (realpart v))
                                   (i (imagpart v)))
                               (* 1000 r)))))))))

(defun emit-array-real-wave (real-file a &key (scale 1))
  (with-open-file (sr real-file :direction :output)
    (let ((n (length a)))
      (write-wave sr (lambda (i)
                       (if (>= i n)
                           nil
                           (* scale (realpart (svref a i)))))))))

;; N = points in dft
;; Tee = sampling period
;; inf = points in reimann series
```

```
(defun log-freq-spectrum  (N Tee inf)
  (let ((x (make-array N :initial-element #c(0.0 0.0)))
        (omega-T (/ (* (/ 1 Tee) (* 2 pi)) N)))
    (dotimes (j inf)
      (let ((i (1+ j)))
        (let ((k (round (* 200 (/ (log i) omega-T)))))
;;        (print (list k (log i) omega-T))
          (when (not (= k 0))
            (setf (svref x k) (complex (* 1000 (/ 1 (sqrt i))) 0))))))
    x))

(defun dirichlet-spectrum  (N Tee inf)
  (let ((x (make-array N :initial-element #c(0.0 0.0)))
        (omega-T (/ (* (/ 1 Tee) (* 2 pi)) N)))
    (dotimes (j inf)
      (let ((i (1+ j)))
        (let ((k (round (* 200 (/ (log i) omega-T)))))
          (when (not (= k 0))
            (setf (svref x k) (complex (* 1000 (/ 1 (sqrt i))) 0))))))
    (dotimes (j inf)
      (let ((i (1+ j)))
        (let ((k (round (* 200 (/ (log (* 2 i)) omega-T)))))
          (when (not (= k 0))
            (setf (svref x k) (complex (* (sqrt 2) 1000 (/ 1 i)) 0))))))
    x))

(defun mag-zeta (tee &optional n)
  (abs (zeta (+ (* (sqrt -1) tee) 1/2) n)))

(defun emit-mag-zeta (file n)
  (with-open-file (s file :direction :output)
    (dotimes (i n)
      (let ((tee (/ i 10)))
        (format s "~a ~a~%" (float tee) (mag-zeta tee 100))))))

;; http://mathworld.wolfram.com/Riemann-SiegelFunctions.html

(defun theta (tee)
  (- (imagpart (log (gamma (+ 1/4 (* 1/2 (sqrt -1) tee)))))
     (* 1/2 (log pi) tee)))

(defun riemann-siegel (tee)
  (* (exp (* (sqrt -1) (theta tee)))
     (zeta (+ (* 1/2 (* (sqrt -1) tee))))))
```

```
(defun emit-riemann-siegel (real-file imag-file n)
  (with-open-file (sr real-file :direction :output)
    (with-open-file (si imag-file :direction :output)
      (dotimes (i n)
        (let ((tee (/ i 10)))
          (let ((v (riemann-siegel tee)))
            (let ((r (realpart v))
                  (i (imagpart v)))
              (format sr "~a ~a~%" (float tee) r)
              (format si "~a ~a~%" (float tee) i))))))))

(defun emit-real-riemann-siegel-wave (real-file n)
  (with-open-file (sr real-file :direction :output)
    (write-wave sr (lambda (i)
                     (if (>= i n)
                         nil
                         (let ((tee (/ i 10)))
                           (let ((v (riemann-siegel tee)))
                             (let ((r (realpart v))
                                   (i (imagpart v)))
                               (* 1000 r)))))))))

(defun sum (start end fcn)
  (let ((r 0))
    (dotimes (i (+ (- end start) 1))
      (let ((n (+ i start)))
        (setq r (+ r (funcall fcn n)))))
    r))

(defun prod (start end fcn)
  (let ((r 1))
    (dotimes (i (+ (- end start) 1))
      (let ((n (+ i start)))
        (setq r (* r (funcall fcn n)))))
    r))

(defun fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))

(let ((comb-hash (make-hash-table :size 1024 :test #'equal))
      (key (cons nil nil)))
  (defun dump-comb-hash ()
    (maphash (lambda (k v)
               (print (list k v)))
             comb-hash))
  (defun comb (n k)
```

```
    (setf (first key) n)
    (setf (rest key) k)
    (let ((v (gethash key comb-hash)))
      (if v
          v
          (let ((v (/ (fact n) (* (fact k) (fact (- n k)))))
                (key (cons n k)))
            (setf (gethash key comb-hash) v)
            v)))))

(defun comb-orig (n k)
  (/ (fact n) (* (fact k) (fact (- n k)))))

;; http://functions.wolfram.com/GammaBetaErf/Gamma/introductions/Gamma/ShowAll.html

(defun gamma (z &optional (inf 500)) ;; 50
  (let ((g 0.5772156))
    (/ 1
       (* z (exp (* g z))
          (prod 1 inf (lambda (k) (* (+ 1 (/ z k)) (exp (- (/ z k))))))))))

#|
(defun gamma (z &optional (inf 50))
  (* (/ 1 z)
     (prod 1 inf (lambda (k) (/ (expt (+ (/ 1 k)) z) (+ 1 (/ z k)))))))
|#

;;;;;; Wave file writers
;; All int writers are little-endian
;; Written in 2's complement
;; All strings are big-endian
;;
;; https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

(defun write-int32 (s n)
  (let ((m n))
    (dotimes (i 4)
      (let ((b (logand m 255)))
        (write-byte b s))
      (setq m (ash m -8)))))

(defun write-int16 (s n)
  (let ((m n))
    (dotimes (i 2)
      (let ((b (logand m 255)))
```

```
        (write-byte b s))
      (setq m (ash m -8)))))

(defun order (s)
  s)

;; (write-wave s (lambda (n) (returns value-to-write))
;;
;; return nil for undefined value, ie, "eof" (expected to be called
;; sequentially, although not required)

(defun write-wave (s fcn)
  (let ((subchunk1size 16)
        (sampling-rate 176400)  ;; Was 44100, 88200
        (nchannels 1)
        (bitspersample 16)
        (nsamples 0))
    (format s (order "RIFF"))
    (let ((totalsize-file-pos (file-position s)))
      (write-int32 s 0)                                          ;; totalsize
      (format s (order  "WAVE"))
      (format s (order "fmt "))                                  ;; subchunk1id -- pcm format
      (write-int32 s 16)                                         ;; subchunk1 size -- constant for pcm format
      (write-int16 s 1)                                          ;; pcm
      (write-int16 s nchannels)                                  ;; n channels (= 1)
      (write-int32 s sampling-rate)                              ;; sampling rate
      (write-int32 s (* sampling-rate nchannels bitspersample)) ;; byte rate
      (write-int16 s (* nchannels bitspersample))                ;; block align
      (write-int16 s bitspersample)                              ;; bitspersample
      (format s (order "data"))                                  ;; subchunk2id -- data
      (let ((subchunk2size-file-pos (file-position s)))
        (write-int32 s 0)                                        ;; subchunk2 size
        (let ((n 0))
          (block xxx
                 (loop
                  (let ((v (funcall fcn n)))
                    (if (null v)
                        (return-from xxx nil)
                        (progn
#|
                         (when (= (mod n 100) 0)
                           (print (list (get-universal-time) n)))
|#
;;                         (print (round v))
                         (write-int16 s (round v)))))))
```

```
                  (setq n (+ n 1))))
          (let ((nsamples n))
            (let ((subchunk2size (* nsamples nchannels (/ bitspersample 8))))
              (file-position s subchunk2size-file-pos)
              (write-int32 s subchunk2size)
              (let ((totalsize (+ 4 (+ 8 subchunk1size) (+ 8 subchunk2size))))
                (file-position s totalsize-file-pos)
                (write-int32 s totalsize)))))))))

#|
(defun test-wave-fcn (n)
  ;; Note is root at zero, each integer step a semitone up or down
  (defun note-freq (note)
    (expt (expt 2 1/12) note))
  (defun note-freq-fcn (tee modulus)
    (let ((n (floor tee modulus)))
      (let ((m (mod n 4)))
        (let ((note
               (cond
                ((= m 0) 0)
                ((= m 1) 4)
                ((= m 2) 7)
                ((= m 3) 12))))
          (note-freq note)))))
  (defun amp (tee)
    (exp (- (* .0001 (mod tee 10000.0)))))
  (defun freq (tee)
    (note-freq-fcn tee 10000.0))
  (defun xfreq (tee)
    (expt (expt 2 1/12) (floor tee 10000.0)))
  (if (>= n 1000000)
      nil
    (let ((tee (/ n 10.0)))
      (let ((a (amp tee))
            (f (* 0.1 (freq tee))))
        (round (* a 5000 (+ (* (sin (* .001 tee)) (sin (* f tee)))
                            (* 0.5 (sin (* 2.17 (* f tee))))
                            (* 0.4 (sin (* 3.17 (* f tee))))
                            (* 0.3 (sin (* 4.17 (* f tee))))
                            (* 0.2 (sin (* 5.17 (* f tee)))))))))))
|#

(defun test-wave-fcn (n)
  ;; Note is root at zero, each integer step a semitone up or down
  (defun note-freq (note)
```

```
    (expt (expt 2 1/12) note))
  (defun note-freq-fcn (tee modulus)
    (let ((n (floor tee modulus)))
      (let ((m (mod n 4)))
        (let ((note
               (cond
                ((= m 0) 0)
                ((= m 1) 4)
                ((= m 2) 7)
                ((= m 3) 12))))
          (note-freq note)))))
  (defun amp (tee)
    (exp (- (* .0001 (mod tee 10000.0)))))
  (defun freq (tee)
    (note-freq-fcn tee 10000.0))
  (defun xfreq (tee)
    (expt (expt 2 1/12) (floor tee 10000.0)))
  (if (>= n 1000000)
      nil
      (let ((tee (/ n 10.0)))
        (let ((a (amp tee))
              (f (* 0.05 (freq tee))))
          (round (* a 5000 (let ((l '((0.3 0.5)
                                      (1.0 1.000)
                                      (0.5 1.183)
                                      (0.4 1.506)
                                      (0.3 2.000)
                                      (0.25 2.514)
                                      (0.25 2.662)
                                      (0.2 3.011)
                                      (0.2 4.166)
                                      (0.2 5.433)
                                      (0.2 6.796)
                                      (0.2 8.215))))
                             (sum 0 (- (length l) 1) (lambda (n) (* (first (nth n l)) (sin (* (second (nth n l)) (* f
tee))))))))))))

;; Basic 1/sqrt(n) harmonic amplitude drop, and logn increase in harmonic frequency

(defun test-wave-fcn1 (n)
  (if (>= n 1000000)
      nil
    (let ((tee (/ n 10.0)))
      (let ((f .05))
        (round (* 5000
```

```
                  (sum 1 10 (lambda (n)
                              (* (/ 1 (sqrt n)) (sin (* (log n) f tee)))))))))))

(defun test-wave-fcn2 (n)
  (when (= (mod n 1000)0)
    (print n))
  (if (>= n 10000000)
      nil
    (let ((tee (/ n 200)))
      (let ((f 1)) ;; freq (omega)
        (round (* 2000
                  (sum 1 300 (lambda (n)
                               (- (* (/ 2 n) (cos (* (+ (log n) (log 2)) f tee)))
                                  (* (expt -1 n)
                                     (/ 1 (sqrt n))
                                     (cos (* (log n) f tee))))))))))))
```