Package 'elliptic'

November 10, 2025

Version 1.5-1

Title Weierstrass and Jacobi Elliptic Functions

Depends R (>= 2.5.0)

Imports MASS

Suggests emulator, calibrator (>= 1.2-8), testthat, hypergeo

SystemRequirements pari/gp

Description A suite of elliptic and related functions including Weierstrass and Jacobi forms. Also includes various tools for manipulating and visualizing complex functions.

Maintainer Robin K. S. Hankin <hankin.robin@gmail.com>

License GPL-2

URL https://github.com/RobinHankin/elliptic, https://robinhankin.github.io/elliptic/

BugReports https://github.com/RobinHankin/elliptic/issues

NeedsCompilation no

Author Robin K. S. Hankin [aut, cre] (ORCID: <https://orcid.org/0000-0001-5982-0415>)

Repository CRAN

Date/Publication 2025-11-10 17:50:02 UTC

Contents

elliptic-package 2
amn 5
as.primitive 6
ck 7
congruence 8
coqueraux 9
divisor 10
e16.28.1 12

e18.10.9 . . . 13
e1e2e3 . . . 14
equianharmonic . . . 15
eta . . . 16
farey . . . 17
fpp . . . 18
g.fun . . . 19
half.periods . . . 21
J . . . 23
K.fun . . . 24
latplot . . . 25
lattice . . . 26
limit . . . 26
massage . . . 27
misc . . . 28
mob . . . 28
myintegrate . . . 29
near.match . . . 32
newton_raphson . . . 33
nome . . . 34
P.laurent . . . 35
p1.tau . . . 36
parameters . . . 37
pari . . . 38
sn . . . 40
sqrti . . . 42
theta . . . 42
theta.neville . . . 44
theta1.dash.zero . . . 46
theta1dash . . . 46
unimodular . . . 48
view . . . 49
WeierstrassP . . . 51

Index **54**

`elliptic-package` *Weierstrass and Jacobi Elliptic Functions*

Description

A suite of elliptic and related functions including Weierstrass and Jacobi forms. Also includes various tools for manipulating and visualizing complex functions.

Details

The DESCRIPTION file:

Package:	elliptic
Version:	1.5-1
Title:	Weierstrass and Jacobi Elliptic Functions
Authors@R:	person(given=c("Robin", "K. S."), family="Hankin", role = c("aut","cre"), email="hankin.robin@gma
Depends:	R (>= 2.5.0)
Imports:	MASS
Suggests:	emulator, calibrator (>= 1.2-8), testthat, hypergeo
SystemRequirements:	pari/gp
Description:	A suite of elliptic and related functions including Weierstrass and Jacobi forms. Also includes various
Maintainer:	Robin K. S. Hankin <hankin.robin@gmail.com>
License:	GPL-2
URL:	https://github.com/RobinHankin/elliptic, https://robinhankin.github.io/elliptic/
BugReports:	https://github.com/RobinHankin/elliptic/issues
Author:	Robin K. S. Hankin [aut, cre] (ORCID: <https://orcid.org/0000-0001-5982-0415>)

Index of help topics:

Im<-	Manipulate real or imaginary components of an object
J	Various modular functions
K.fun	quarter period K
P.laurent	Laurent series for elliptic and related functions
WeierstrassP	Weierstrass P and related functions
amn	matrix a on page 637
as.primitive	Converts basic periods to a primitive pair
ck	Coefficients of the Laurent expansion of the Weierstrass P function
congruence	Solves mx+by=1 for x and y
coqueraux	Fast, conceptually simple, iterative scheme for Weierstrass P functions
divisor	Number theoretic functions
e16.28.1	Numerical verification of equations 16.28.1 to 16.28.5
e18.10.9	Numerical checks of equations 18.10.9-11, page 650
e1e2e3	Calculate e1, e2, e3 from the invariants
elliptic-package	Weierstrass and Jacobi Elliptic Functions
equianharmonic	Special cases of the Weierstrass elliptic function
eta	Dedekind's eta function
farey	Farey sequences
fpp	Fundamental period parallelogram
g.fun	Calculates the invariants g2 and g3
half.periods	Calculates half periods in terms of e
latplot	Plots a lattice of periods on the complex plane
lattice	Lattice of complex numbers

limit	Limit the magnitude of elements of a vector
massage	Massages numbers near the real line to be real
mob	Moebius transformations
myintegrate	Complex integration
near.match	Are two vectors close to one another?
newton_raphson	Newton Raphson iteration to find roots of equations
nome	Nome in terms of m or k
p1.tau	Does the Right Thing (tm) when calling g2.fun() and g3.fun()
parameters	Parameters for Weierstrass's P function
pari	Wrappers for PARI functions
sn	Jacobi form of the elliptic functions
sqrti	Generalized square root
theta	Jacobi theta functions 1-4
theta.neville	Neville's form for the theta functions
theta1.dash.zero	Derivative of theta1
theta1dash	Derivatives of theta functions
unimodular	Unimodular matrices
view	Visualization of complex functions

The primary function in package **elliptic** is `P()`: this calculates the Weierstrass $\wp$ function, and may take named arguments that specify either the invariants `g` or half periods `Omega`. The derivative is given by function `Pdash` and the Weierstrass sigma and zeta functions are given by functions `sigma()` and `zeta()` respectively; these are documented in `?P`. Jacobi forms are documented under `?sn` and modular forms under `?J`.

Notation follows Abramowitz and Stegun (1965) where possible, although there only real invariants are considered; `?e1e2e3` and `?parameters` give a more detailed discussion. Various equations from AMS-55 are implemented (for fun); the functions are named after their equation numbers in AMS-55; all references are to this work unless otherwise indicated.

The package uses Jacobi's theta functions (`?theta` and `?theta.neville`) where possible: they converge very quickly.

Various number-theoretic functions that are required for (eg) converting a period pair to primitive form (`?as.primitive`) are implemented; see `?divisor` for a list.

The package also provides some tools for numerical verification of complex analysis such as contour integration (`?myintegrate`) and Newton-Raphson iteration for complex functions (`?newton_raphson`).

Complex functions may be visualized using `view()`; this is customizable but has an extensive set of built-in colourmaps.

Author(s)

Robin K. S. Hankin [aut, cre] (ORCID: <https://orcid.org/0000-0001-5982-0415>)

Maintainer: Robin K. S. Hankin <hankin.robin@gmail.com>

Examples

```
## Example 8, p666, RHS:
```

```
P(z=0.07 + 0.1i, g=c(10,2))

    ## Now a nice little plot of the zeta function:
x <- seq(from=-4,to=4,len=100)
z <- outer(x,1i*x,"+")
par(pty="s")
view(x,x,limit(zeta(z,c(1+1i,2-3i))),nlevels=3,scheme=1)
view(x,x,P(z*3,params=equianharmonic()),real=FALSE)

    ## Some number theory:
mobius(1:10)
plot(divisor(1:300,k=1),type="s",xlab="n",ylab="divisor(n,1)")

   ## Primitive periods:
as.primitive(c(3+4.01i , 7+10i))
as.primitive(c(3+4.01i , 7+10i),n=10)   # Note difference

   ## Now some contour integration:
f <- function(z){1/z}
u <- function(x){exp(2i*pi*x)}
udash <- function(x){2i*pi*exp(2i*pi*x)}
integrate.contour(f,u,udash) - 2*pi*1i


x <- seq(from=-10,to=10,len=200)
z <- outer(x,1i*x,"+")
view(x,x,P(z,params=lemniscatic()),real=FALSE)
view(x,x,P(z,params=pseudolemniscatic()),real=FALSE)
view(x,x,P(z,params=equianharmonic()),real=FALSE)
```

---

amn *matrix a on page 637*

Description

Matrix of coefficients of the Taylor series for $\sigma(z)$ as described on page 636 and tabulated on page 637.

Usage

```
amn(u)
```

Arguments

u	Integer specifying size of output matrix

Details

Reproduces the coefficients a_{mn} on page 637 according to recurrence formulae 18.5.7 and 18.5.8, p636. Used in equation 18.5.6.

Author(s)

Robin K. S. Hankin

Examples

```
amn(12)   #page 637
```

`as.primitive` *Converts basic periods to a primitive pair*

Description

Given a pair of basic periods, returns a primitive pair and (optionally) the unimodular transformation used.

Usage

```
as.primitive(p, n = 3, tol = 1e-05, give.answers = FALSE)
is.primitive(p, n = 3, tol = 1e-05)
```

Arguments

`p`	Two element vector containing the two basic periods
`n`	Maximum magnitude of matrix entries considered
`tol`	Numerical tolerance used to determine reality of period ratios
`give.answers`	Boolean, with `TRUE` meaning to return extra information (unimodular matrix and the magnitudes of the primitive periods) and default `FALSE` meaning to return just the primitive periods

Details

Primitive periods are not unique. This function follows Chandrasekharan and others (but not, of course, Abramowitz and Stegun) in demanding that the real part of `p1`, and the imaginary part of `p2`, are nonnegative.

Value

If `give.answers` is `TRUE`, return a list with components

`M`	The unimodular matrix used
`p`	The pair of primitive periods
`mags`	The magnitudes of the primitive periods

Note

Here, "unimodular" includes the case of determinant minus one.

Author(s)

Robin K. S. Hankin

References

K. Chandrasekharan 1985. *Elliptic functions*, Springer-Verlag

Examples

```
as.primitive(c(3+5i, 2+3i))
as.primitive(c(3+5i, 2+3i), n = 5)

##Rounding error:
is.primitive(c(1, 1i))

## Try
 is.primitive(c(1, 1.001i))
```

---

ck *Coefficients of the Laurent expansion of the Weierstrass P function*

Description

Calculates the coefficients of the Laurent expansion of the Weierstrass $\wp$ function in terms of the invariants

Usage

```
ck(g, n = 20)
```

Arguments

g	The invariants: a vector of length two with `g = c(g2, g3)`
n	length of series

Details

Calculates the series c_k as per equation 18.5.3, p635.

Author(s)

Robin K. S. Hankin

See Also

`P.laurent`

Examples

```
 #Verify 18.5.16, p636:
 x <- ck(g = c(0.1+1.1i, 4-0.63i))
14*x[2]*x[3]*(389*x[2]^3+369*x[3]^2)/3187041-x[11] #should be zero


# Now try a random example by comparing the default (theta function) method
# for P(z) with the Laurent expansion:

z <- 0.5 - 0.3i
g <- c(1.1-0.2i, 1+0.4i)
series <- ck(15, g = g)
1/z^2 + sum(series*(z^2)^(0:14)) - P(z, g  =g) # should be zero
```

`congruence` *Solves mx+by=1 for x and y*

Description

Solves the Diophantine equation $mx + by = 1$ for x and y. The function is named for equation 57 in Hardy and Wright.

Usage

```
congruence(a, l = 1)
```

Arguments

`a`	Two element vector with `a=c(m,n)`
`l`	Right hand side with default 1

Value

In the usual case of $(m, n) = 1$, returns a square matrix whose rows are `a` and `c(x,y)`. This matrix is a unimodular transformation that takes a pair of basic periods to another pair of basic periods.

If $(m, n) \neq 1$ then more than one solution is available (for example `congruence(c(4,6),2)`). In this case, extra rows are added and the matrix is no longer square.

Note

This function does not generate *all* unimodular matrices with a given first row (here, it will be assumed that the function returns a square matrix).

For a start, this function only returns matrices all of whose elements are positive, and if `a` is unimodular, then after `diag(a) <- -diag(a)`, both `a` and `-a` are unimodular (so if `a` was originally generated by `congruence()`, neither of the derived matrices could be).

Now if the first row is `c(1,23)`, for example, then the second row need only be of the form `c(n,1)` where `n` is any integer. There are thus an infinite number of unimodular matrices whose first row is

`c(1,23)`. While this is (somewhat) pathological, consider matrices with a first row of, say, `c(2,5)`. Then the second row could be `c(1,3)`, or `c(3,8)` or `c(5,13)`. Function `congruence()` will return only the first of these.

To systematically generate all unimodular matrices, use `unimodular()`, which uses Farey sequences.

Author(s)

Robin K. S. Hankin

References

G. H. Hardy and E. M. Wright 1985. *An introduction to the theory of numbers*, Oxford University Press (fifth edition)

See Also

`unimodular`

Examples

```
M <- congruence(c(4,9))
det(M)

o <- c(1,1i)
g2.fun(o) - g2.fun(o,maxiter=840)  #should be zero
```

`coqueraux` *Fast, conceptually simple, iterative scheme for Weierstrass P functions*

Description

Fast, conceptually simple, iterative scheme for Weierstrass $\wp$ functions, following the ideas of Robert Coqueraux

Usage

```
coqueraux(z, g, N = 5, use.fpp = FALSE, give = FALSE)
```

Arguments

`z`	Primary complex argument
`g`	Invariants; if an object of type `parameters` is supplied, the invariants will be extracted appropriately
`N`	Number of iterations to use
`use.fpp`	Boolean, with default `FALSE` meaning to *not* reduce `z` to the fpp. Setting to `TRUE` reduces `z` to the fpp via `parameters()`: this is more accurate (see example) but slower

give — Boolean, with TRUE meaning to return an estimate of the error, and FALSE meaning to return just the value

Author(s)

Robin K. S. Hankin

References

R. Coqueraux, 1990. *Iterative method for calculation of the Weierstrass elliptic function*, IMA Journal of Numerical Analysis, volume 10, pp119-128

Examples

```
z <- seq(from=1+1i,to=30-10i,len=55)
p <- P(z,c(0,1))
c.true <- coqueraux(z,c(0,1),use.fpp=TRUE)
c.false <- coqueraux(z,c(0,1),use.fpp=FALSE)
plot(1:55,abs(p-c.false))
points(1:55,abs(p-c.true),pch=16)
```

---

divisor *Number theoretic functions*

Description

Various useful number theoretic functions

Usage

```
divisor(n,k=1)
primes(n)
factorize(n)
mobius(n)
totient(n)
liouville(n)
```

Arguments

n, k — Integers

Details

Functions `primes()` and `factorize()` cut-and-pasted from Bill Venables's **con.design** package, version 0.0-3. Function `primes(n)` returns a vector of all primes not exceeding `n`; function `factorize(n)` returns an integer vector of nondecreasing primes whose product is `n`.

The others are multiplicative functions, defined in Hardy and Wright:

Function `divisor()`, also written $\sigma_k(n)$, is the divisor function defined on p239. This gives the sum of the k^{th} powers of all the divisors of `n`. Setting $k = 0$ corresponds to $d(n)$, which gives the number of divisors of `n`.

Function `mobius()` is the Moebius function (p234), giving zero if `n` has a repeated prime factor, and $(-1)^q$ where $n = p_1 p_2 \ldots p_q$ otherwise.

Function `totient()` is Euler's totient function (p52), giving the number of integers smaller than `n` and relatively prime to it.

Function `liouville()` gives the Liouville function.

Note

The divisor function crops up in `g2.fun()` and `g3.fun()`. Note that this function is not called `sigma()` to avoid conflicts with Weierstrass's σ function (which ought to take priority in this context).

Author(s)

Robin K. S. Hankin and Bill Venables (`primes()` and `factorize()`)

References

G. H. Hardy and E. M. Wright, 1985. *An introduction to the theory of numbers* (fifth edition). Oxford University Press.

Examples

```
mobius(1)
mobius(2)
divisor(140)
divisor(140,3)


plot(divisor(1:100,k=1),type="s",xlab="n",ylab="divisor(n,1)")

plot(cumsum(liouville(1:1000)),type="l",main="does the function ever exceed zero?")
```

e16.28.1 *Numerical verification of equations 16.28.1 to 16.28.5*

Description

Numerical verification of formulae 16.28.1 to 16.28.5 on p576

Usage

```
e16.28.1(z, m, ...)
e16.28.2(z, m, ...)
e16.28.3(z, m, ...)
e16.28.4(z, m, ...)
e16.28.5(m, ...)
```

Arguments

z	Complex number
m	Parameter m
...	Extra arguments passed to `theta[1-4]()`

Details

Returns the left hand side minus the right hand side of each formula. Each formula documented here is identically zero; nonzero values are returned due to numerical errors and should be small.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions.* New York, Dover.

Examples

```
plot(e16.28.4(z=1:6000,m=0.234))
plot(abs(e16.28.4(z=1:6000,m=0.234+0.1i)))
```

`e18.10.9` *Numerical checks of equations 18.10.9-11, page 650*

Description

Numerical checks of equations 18.10.9-11, page 650. Function returns LHS minus RHS.

Usage

```
e18.10.9(parameters)
```

Arguments

`parameters`	An object of class "parameters"

Value

Returns a complex vector of length three: e_1, e_2, e_3

Note

A good check for the three e's being in the right order.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions.* New York, Dover.

Examples

```
e18.10.9(parameters(g=c(0,1)))
e18.10.9(parameters(g=c(1,0)))
```

e1e2e3 *Calculate e1, e2, e3 from the invariants*

Description

Calculates e_1, e_2, e_3 from the invariants using either `polyroot` or Cardano's method.

Usage

```
e1e2e3(g, use.laurent=TRUE, AnS=is.double(g), Omega=NULL, tol=1e-6)
eee.cardano(g)
```

Arguments

`g`	Two-element vector with `g=c(g2,g3)`
`use.laurent`	Boolean, with default `TRUE` meaning to use `P.laurent()` to determine the correct ordering for the e: $\wp(\omega_1)$, $\wp(\omega_2)$, $\wp(\omega_3)$. Setting to `FALSE` means to return the solutions of the cubic equation directly: this is much faster, but is not guaranteed to find the e_i in the right order (the roots are found according to the vagaries of `polyroot()`)
`AnS`	Boolean, with default `TRUE` meaning to define ω_3 as per ams-55, and `FALSE` meaning to follow Whittaker and Watson, and define ω_1 and ω_2 as the primitive half periods, and $\omega_3 = -\omega_1 - \omega_2$. This is also consistent with Chandrasekharan except the factor of 2. Also note that setting `AnS` to `TRUE` forces the e to be real
`Omega`	A pair of primitive half periods, if known. If supplied, the function uses them to calculate approximate values for the three es (but supplies values calculated by `polyroot()`, which are much more accurate). The function needs the approximate values to determine in which order the es should be, as `polyroot()` returns roots in whichever order the polynomial solver gives them in
`tol`	Real, relative tolerance criterion for terminating Laurent summation

Value

Returns a three-element vector.

Note

Function `parameters()` calls `e1e2e3()`, so **do not use** `parameters()` **to determine argument** `g`**, because doing so will result in a recursive loop.**

Just to be specific: `e1e2e3(g=parameters(...))` will fail. It would be pointless anyway, because `parameters()` returns (inter alia) e_1, e_2, e_3.

There is considerable confusion about the order of e_1, e_2 and e_3, essentially due to Abramowitz and Stegun's definition of the half periods being inconsistent with that of Chandrasekharan's, and Mathematica's. It is not possible to reconcile A and S's notation for theta functions with Chandrasekharan's definition of a primitive pair. Thus, the convention adopted here is the rather strange-seeming

choice of $e_1 = \wp(\omega_1/2)$, $e_2 = \wp(\omega_3/2)$, $e_3 = \wp(\omega_2/2)$. This has the advantage of making equation 18.10.5 (p650, ams55), and equation 09.13.27.0011.01, return three identical values.

The other scheme to rescue 18.10.5 would be to define (ω_1, ω_3) as a primitive pair, and to require $\omega_2 = -\omega_1 - \omega_3$. This is the method adopted by Mathematica; it is no more inconsistent with ams55 than the solution used in package **elliptic**. However, this scheme suffers from the disadvantage that the independent elements of `Omega` would have to be supplied as `c(omega1,NA,omega3)`, and this is inimical to the precepts of R.

One can realize the above in practice by considering what this package calls "ω_2" to be *really* ω_3, and what this package calls "$\omega_1 + \omega_2$" to be *really* ω_2. Making function `half.periods()` return a three element vector with names `omega1`, `omega3`, `omega2` might work on some levels, and indeed might be the correct solution for a user somewhere; but it would be confusing. This confusion would dog my weary steps for ever more.

Author(s)

Robin K. S. Hankin

References

Mathematica

Examples

```
sum(e1e2e3(g=c(1,2)))
```

`equianharmonic` *Special cases of the Weierstrass elliptic function*

Description

Gives parameters for the equianharmonic case, the lemniscatic case, and the pseudolemniscatic case.

Usage

```
equianharmonic(...)
lemniscatic(...)
pseudolemniscatic(...)
```

Arguments

`...`	Ignored

Details

These functions return values from section 18.13, p652; 18.14, p658; and 18.15, p662. They use elementary functions (and the gamma function) only, so ought to be more accurate and faster than calling `parameters(g=c(1,0))` directly.

Note that the values for the half periods correspond to the general case for complex `g2` and `g3` so are simple linear combinations of those given in AnS.

One can use `parameters("equianharmonic")` *et seq* instead.

Value

Returns a list with the same elements as `parameters()`.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions.* New York, Dover.

See Also

`parameters`

Examples

```
P(z=0.1+0.1212i,params=equianharmonic())


x <- seq(from=-10,to=10,len=200)
z <- outer(x,1i*x,"+")
view(x,x,P(z,params=lemniscatic()),real=FALSE)
view(x,x,P(z,params=pseudolemniscatic()),real=FALSE)
view(x,x,P(z,params=equianharmonic()),real=FALSE)
```

---

eta *Dedekind's eta function*

Description

Dedekind's η function

Usage

```
eta(z, ...)
eta.series(z, maxiter=300)
```

Arguments

z	Complex argument
...	In function eta(), extra arguments sent to theta3()
maxiter	In function eta.series(), maximum value of iteration

Details

Function eta() uses Euler's formula, viz

$$\eta(z) = e^{\pi i z/12}\theta_3\left(\frac{1}{2}+\frac{z}{2}, 3z\right)$$

Function eta.series() is present for validation (and interest) only; it uses the infinite product formula:

$$\eta(z) = e^{\pi i z/12}\prod_{n=1}^{\infty}\left(1-e^{2\pi i n z}\right)$$

Author(s)

Robin K. S. Hankin

References

K. Chandrasekharan 1985. *Elliptic functions*, Springer-Verlag.

See Also

farey

Examples

```
z <- seq(from=1+1i, to=10+0.06i, len=999)
plot(eta(z))

max(abs(eta(z) - eta.series(z)))
```

---

farey *Farey sequences*

Description

Returns the Farey sequence of order n

Usage

```
farey(n, print=FALSE, give.series = FALSE)
```

Arguments

n	Order of Farey sequence
print	Boolean, with TRUE meaning to print out the text version of the Farey sequence in human-readable form. Default value of FALSE means not to print anything
give.series	Boolean, with TRUE meaning to return the series explicitly, and default FALSE meaning to return a 3 dimensional array as detailed below

Details

If give.series takes its default value of FALSE, return a matrix a of dimension c(2,u) where u is a (complicated) function of n. If v <- a[i,], then v[1]/v[2] is the i^{th} term of the Farey sequence. Note that det(a[(n):(n+1),])== -1

If give.series is TRUE, then return a matrix a of size c(4,u-1). If v <- a[i,], then v[1]/v[2] and v[3]/v[4] are successive pairs of the Farey sequence. Note that det(matrix(a[,i],2,2))== -1

Author(s)

Robin K. S. Hankin

References

G. H. Hardy and E. M. Wright 1985. *An introduction to the theory of numbers*, Oxford University Press (fifth edition)

See Also

unimodular

Examples

```
farey(3)
```

---

fpp *Fundamental period parallelogram*

Description

Reduce $z = x + iy$ to a congruent value within the fundamental period parallelogram (FPP). Function mn() gives (real, possibly noninteger) m and n such that $z = m \cdot p_1 + n \cdot p_2$.

Usage

```
fpp(z, p, give = FALSE)
mn(z, p)
```

Arguments

z	Primary complex argument
p	Vector of length two with first element the first period and second element the second period. Note that p is the period, so $p_1 = 2\omega_1$, where ω_1 is the half period
give	Boolean, with TRUE meaning to return M and N, and default FALSE meaning to return just the congruent values

Details

Function fpp() is fully vectorized.

Use function mn() to determine the "coordinates" of a point.

Use floor(mn(z, p)) %*% p to give the complex value of the (unique) point in the same period parallelogram as z that is congruent to the origin.

Author(s)

Robin K. S. Hankin

Examples

```
p <- c(1.01+1.123i, 1.1+1.43i)
mn(z=1:10, p) %*% p  ## should be close to 1:10

 #Now specify some periods:
 p2 <- c(1+1i, 1-1i)

 #Define a sequence of complex numbers that zooms off to infinity:
 u <- seq(from=0, by=pi+1i*exp(1), len=2007)

 #and plot the sequence, modulo the periods:
 plot(fpp(z=u, p=p2))

 #and check that the resulting points are within the qpp:
 polygon(c(-1, 0, 1, 0), c(0, 1, 0, -1))
```

---

g.fun *Calculates the invariants g2 and g3*

Description

Calculates the invariants g2 and g3 using any of a number of methods

Usage

```
g.fun(b, ...)
g2.fun(b, use.first=TRUE, ...)
g3.fun(b, use.first=TRUE, ...)
g2.fun.lambert(b, nmax=50, tol=1e-10, strict=TRUE)
g3.fun.lambert(b, nmax=50, tol=1e-10, strict=TRUE)
g2.fun.direct(b, nmax=50, tol=1e-10)
g3.fun.direct(b, nmax=50, tol=1e-10)
g2.fun.fixed(b, nmax=50, tol=1e-10, give=FALSE)
g3.fun.fixed(b, nmax=50, tol=1e-10, give=FALSE)
g2.fun.vectorized(b, nmax=50, tol=1e-10, give=FALSE)
g3.fun.vectorized(b, nmax=50, tol=1e-10, give=FALSE)
```

Arguments

b	Half periods. NB: the arguments are the **half periods** as per AMS55! In these functions, argument b is interpreted as per p1.tau()
nmax	Maximum number of terms to sum. See details section for more discussion
tol	Numerical tolerance for stopping: summation stops when adding an additional term makes less
strict	Boolean, with default (where taken) TRUE meaning to stop() if convergence is not achieved in nmax terms. Setting to FALSE returns the partial sum and a warning.
give	Boolean, with default (where taken) TRUE meaning to return the partial sums. See examples section for an example of this argument in use
...	In functions g.fun(), g2.fun() and g3.fun(), extra arguments passed to theta1() and friends
use.first	In function g2.fun() and g3.fun(), Boolean with default TRUE meaning to use Wolfram's first formula (**remember to cite this**) and FALSE meaning to use the second

Details

Functions g2.fun() and g3.fun() use theta functions which converge very quickly. These functions are the best in most circumstances. The theta functions include a loop that continues to add terms until the partial sum is unaltered by addition of the next term. Note that summation continues until *all* elements of the argument are properly summed, so performance is limited by the single worst-case element.

The following functions are provided for interest only, although there is a remote possibility that some weird circumstances may exist in which they are faster than the theta function approach.

Functions g2.fun.divisor() and g3.fun.divisor() use Chandrasekharan's formula on page 83. This is generally slower than the theta function approach

Functions g2.fun.lambert() and g3.fun.lambert() use a Lambert series to accelerate Chandrasekharan's formula. In general, it is a little better than the divisor form.

Functions g2.fun.fixed() and g2.fun.fixed() also use Lambert series. These functions are vectorized in the sense that the function body uses only vector operations. These functions do

not take a vector argument. They are called "fixed" because the number of terms used is fixed in advance (unlike `g2.fun()` and `g3.fun()`).

Functions `g2.fun.vectorized()` and `g3.fun.vectorized()` also use Lambert series. They are fully vectorized in that they take a vector of periods or period ratios, unlike the previous two functions. However, this can lead to loss of precision in some cases (specifically when the periods give rise to widely varying values of g2 and g3).

Functions `g2.fun.direct()` and `g3.fun.direct()` use a direct summation. These functions are absurdly slow. In general, the Lambert series functions converge much faster; and the "default" functions `g2.fun()` and `g3.fun()`, which use theta functions, converge faster still.

Author(s)

Robin K. S. Hankin

References

Mathematica website

Examples

```
g.fun(half.periods(g=c(8,4+1i)))  ## should be c(8,4+1i)


## Example 4, p664, LHS:
omega <- c(10,11i)
(g2 <- g2.fun(omega))
(g3 <- g3.fun(omega))
e1e2e3(Re(c(g2,g3)))

## Example 4, p664, RHS:
omega2 <- 10
omega2dash <- 11i
omega1 <- (omega2-omega2dash)/2   ## From figure 18.1, p630
(g2 <- g2.fun(c(omega1,omega2)))
(g3 <- g3.fun(c(omega1,omega2)))
e1e2e3(Re(c(g2,g3)))
```

`half.periods` *Calculates half periods in terms of e*

Description

Calculates half periods in terms of e

Usage

```
half.periods(ignore=NULL, e=NULL, g=NULL, primitive)
```

Arguments

e	e
g	g
ignore	Formal argument present to ensure that e or g is named (ignored)
primitive	Boolean, with default TRUE meaning to return primitive periods and FALSE to return the direct result of Legendre's iterative scheme

Details

Parameter e=c(e1, e2, e3) are the values of the Weierstrass $\wp$ function at the half periods:

$$e_1 = \wp(\omega_1) \qquad e_2 = \wp(\omega_2) \qquad e_3 = \wp(\omega_3)$$

where

$$\omega_1 + \omega_2 + \omega_3 = 0.$$

Also, e is given by the roots of the cubic equation $x^3 - g_2 x - g_3 = 0$, but the problem is finding which root corresponds to which of the three elements of e.

Value

Returns a pair of primitive half periods

Note

Function parameters() uses function half.periods() internally, so do not use parameters() to determine e.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions.* New York, Dover.

Examples

```
half.periods(g=c(8,4))               ## Example 6, p665, LHS

u <- half.periods(g=c(-10,2))
massage(c(u[1]-u[2] , u[1]+u[2]))    ## Example 6, p665, RHS

half.periods(g=c(10,2))              ## Example 7, p665, LHS

u <- half.periods(g=c(7,6))
massage(c(u[1],2*u[2]+u[1]))         ## Example 7, p665, RHS


half.periods(g=c(1, 1i, 1.1+1.4i))
```

```
half.periods(e=c(1, 1i, 2, 1.1+1.4i))


g.fun(half.periods(g=c(8,4)))         ##  should be c(8,4)
```

J *Various modular functions*

Description

Modular functions including Klein's modular function J (aka Dedekind's Valenz function J, aka the Klein invariant function, aka Klein's absolute invariant), the lambda function, and Delta.

Usage

```
J(tau, use.theta = TRUE, ...)
lambda(tau, ...)
```

Arguments

tau	τ; it is assumed that `Im(tau)>0`
use.theta	Boolean, with default `TRUE` meaning to use the theta function expansion, and `FALSE` meaning to evaluate `g2` and `g3` directly
...	Extra arguments sent to either `theta1()` et seq, or `g2.fun()` and `g3.fun()` as appropriate

Author(s)

Robin K. S. Hankin

References

K. Chandrasekharan 1985. *Elliptic functions*, Springer-Verlag.

Examples

```
 J(2.3+0.23i, use.theta=TRUE)
 J(2.3+0.23i, use.theta=FALSE)

 #Verify that J(z)=J(-1/z):
 z <- seq(from=1+0.7i, to=-2+1i, len=20)
 plot(abs((J(z)-J(-1/z))/J(z)))

 # Verify that lambda(z) = lambda(Mz) where M is a modular matrix with b,c
 # even and a,d odd:
```

```
M <- matrix(c(5, 4, 16, 13), 2, 2)
z <- seq(from=1+1i, to=3+3i, len=100)
plot(lambda(z)-lambda(M %mob% z, maxiter=100))


#Now a nice little plot; vary n to change the resolution:
n <- 50
x <- seq(from=-0.1, to=2,len=n)
y <- seq(from=0.02, to=2,len=n)

z <- outer(x, 1i*y, "+")
f <- lambda(z, maxiter=40)
g <- J(z)
view(x, y, f, scheme=04, real.contour=FALSE, main="try higher resolution")
view(x, y, g, scheme=10, real.contour=FALSE, main="try higher resolution")
```

`K.fun` *quarter period K*

Description

Calculates the K.fun in terms of either m (`K.fun()`) or k (`K.fun.k()`).

Usage

```
K.fun(m, strict=TRUE, maxiter=7, miniter=3)
```

Arguments

`m`	Real or complex parameter
`strict`	Boolean, with default `TRUE` meaning to return an error if the sequence has not converged exactly, and `FALSE` meaning to return the partial sum, and a warning
`maxiter`	Maximum number of iterations
`miniter`	Minimum number of iterations to guard against premature exit if an addend is zero exactly

Author(s)

Robin K. S. Hankin

References

R. Coquereaux, A. Grossman, and B. E. Lautrup. *Iterative method for calculation of the Weierstrass elliptic function*. IMA Journal of Numerical Analysis, vol 10, pp119-128, 1990

Examples

```
K.fun(0.09)  # AMS-55 give 1.60804862 in example 7 on page 581

# next example not run because: (i), it needs gsl; (ii) it gives a warning.
## Not run:
K.fun(0.4,strict=F, maxiter=4) - ellint_Kcomp(sqrt(0.4))

## End(Not run)
```

`latplot` *Plots a lattice of periods on the complex plane*

Description

Given a pair of basic periods, plots a lattice of periods on the complex plane

Usage

```
latplot(p, n=10, do.lines=TRUE, ...)
```

Arguments

`p`	Vector of length two with first element the first period and second element the second period. Note that $p_1 = 2\omega_1$
`n`	Size of lattice
`do.lines`	Boolean with default `TRUE` meaning to show boundaries between adjacent period parallelograms
`...`	Extra arguments passed to `plot()`. See examples section for working use

Author(s)

Robin K. S. Hankin

References

K. Chandrasekharan 1985. *Elliptic functions*, Springer-Verlag.

Examples

```
p1 <- c(1, 1i)
p2 <- c(2+3i, 5+7i)
latplot(p1)
latplot(p2, xlim=c(-4,4), ylim=c(-4,4), n=40)
```

lattice *Lattice of complex numbers*

Description

Returns a lattice of numbers generated by a given complex basis.

Usage

```
lattice(p,n)
```

Arguments

p	Complex vector of length two giving a basis for the lattice
n	size of lattice

Author(s)

Robin K. S. Hankin

Examples

```
 lattice(c(1+10i,100+1000i),n=2)
plot(lattice(c(1+1i,1.1+1.4i),5))
```

limit *Limit the magnitude of elements of a vector*

Description

Deals appropriately with objects with a few very large elements

Usage

```
limit(x, upper=quantile(Re(x),0.99,na.rm=TRUE),
         lower=quantile(Re(x),0.01,na.rm=TRUE),
         na = FALSE)
```

Arguments

x	Vector of real or complex values
upper	Upper limit
lower	Lower limit
na	Boolean, with default FALSE meaning to "clip" x (if real) by setting elements of x with x>high to high; if TRUE, set such elements to NA. If x is complex, this argument is ignored

Details

If x is complex, low is ignored and the function returns x, after executing x[abs(x)>high] <- NA.

Author(s)

Robin K. S. Hankin

Examples

```
x <- c(rep(1,5),300, -200)
limit(x,100)
limit(x,upper=200,lower= -400)
limit(x,upper=200,lower= -400,na=TRUE)
```

---

massage *Massages numbers near the real line to be real*

Description

Returns the Real part of numbers near the real line

Usage

```
massage(z, tol = 1e-10)
```

Arguments

z	vector of complex numbers to be massaged
tol	Tolerance

Author(s)

Robin K. S. Hankin

Examples

```
massage(1+1i)
massage(1+1e-11i)

massage(c(1,1+1e-11i,1+10i))
```

misc *Manipulate real or imaginary components of an object*

Description

Manipulate real or imaginary components of an object

Usage

```
Im(x) <- value
Re(x) <- value
```

Arguments

x	Complex-valued object
value	Real-valued object

Author(s)

Robin K. S. Hankin

Examples

```
x <- 1:10
Im(x) <- 1

x <- 1:5
Im(x) <- 1/x
```

mob *Moebius transformations*

Description

Moebius transformations

Usage

```
mob(M, x)
M %mob% x
```

Arguments

M	2-by-2 matrix of integers
x	vector of values to be transformed

Value

Returns a value with the same attributes as `x`. Elementwise, if

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

then `mob(M,x)` is $\frac{ax+b}{cx+d}$.

Note

This function does not check for `M` being having integer elements, nor for the determinant being unity.

Author(s)

Robin K. S. Hankin

References

Wikipedia contributors, "Mobius transformation," Wikipedia, The Free Encyclopedia (accessed February 13, 2011).

See Also

`unimodular`

Examples

```
M <- matrix(c(11,6,9,5),2,2)
x <- seq(from=1+1i,to=10-2i,len=6)

M %mob% x
plot(mob(M,x))
```

---

myintegrate *Complex integration*

Description

Integration of complex valued functions along the real axis (`myintegrate()`), along arbitrary paths (`integrate.contour()`), and following arbitrary straight line segments (`integrate.segments()`). Also, evaluation of a function at a point using the residue theorem (`residue()`). A vignette ("`residuetheorem`") is provided in the package.

Usage

```
myintegrate(f, lower, upper, ...)
integrate.contour(f, u, udash, ...)
integrate.segments(f, points, close=TRUE, ...)
residue(f, z0, r, O=z0, ...)
```

Arguments

f	function, possibly complex valued
lower, upper	Lower and upper limits of integration in `myintegrate()`; real numbers (for complex values, use `integrate.contour()` or `integrate.segments()`)
u	Function mapping $[0,1]$ to the contour. For a closed contour, require that $u(0) = u(1)$
udash	Derivative of `u`
points	In function `integrate.segments()`, a vector of complex numbers. Integration will be taken over straight segments joining consecutive elements of `points`
close	In function `integrate.segments()`, a Boolean variable with default `TRUE` meaning to integrate along the segment from `points[n]` to `points[1]` in addition to the internal segments
r, O, z0	In function `residue()` returns `f(z0)` by integrating $f(z)/(z - z0)$ around a circle of radius `r` and center `O`
...	Extra arguments passed to `integrate()`

Author(s)

Robin K. S. Hankin

Examples

```
f1 <- function(z){sin(exp(z))}
f2 <- function(z, p){p/z}

myintegrate(f1, 2, 3)  # that is, along the real axis


integrate.segments(f1, c(1, 1i, -1, -1i), close = TRUE)   # should be zero

# (following should be pi*2i; note secondary argument):
integrate.segments(f2, points = c(1, 1i, -1, -1i), close = TRUE, p = 1)


# To integrate round the unit circle, we need the contour and its
# derivative:

 u <- function(x){exp(pi*2i*x)}
 udash <- function(x){pi*2i * exp(pi*2i*x)}

# Some elementary functions, for practice:
```

```
# (following should be 2i*pi; note secondary argument 'p'):
integrate.contour(function(z, p){p/z}, u, udash, p = 1)
integrate.contour(function(z){log(z)}, u, udash)           # should be -2i*pi
integrate.contour(function(z){sin(z) + 1/z^2}, u, udash)   # should be zero



# residue() is a convenience wrapper integrating f(z)/(z-z0) along a
# circular contour:

residue(function(z){1/z}, 2, r = 0.1)  # should be 1/2=0.5


# Now, some elliptic functions:
g <- c(3, 2+4i)
Zeta <- function(z){zeta(z, g)}
Sigma <- function(z){sigma(z, g)}
WeierstrassP <- function(z){P(z, g)}

jj <- integrate.contour(Zeta, u, udash)
abs(jj - 2*pi*1i)                               # zero to numerical precision
abs(integrate.contour(Sigma, u, udash))        # zero to numerical precision
abs(integrate.contour(WeierstrassP, u, udash)) # zero to numerical precision


# Now integrate f(x) = exp(1i*x)/(1+x^2) from -Inf to +Inf along the
# real axis, using the Residue Theorem.  This tells us that integral of
# f(z) along any closed path is equal to pi*2i times the sum of the
# residues inside it.  Take a semicircular path P from -R to +R along
# the real axis, then following a semicircle in the upper half plane, of
# radius R to close the loop.  Now consider large R.  Then P encloses a
# pole at +1i [there is one at -1i also, but this is outside P, so
# irrelevant here] at which the residue is -1i/2e.  Thus the integral of
# f(z) = 2i*pi*(-1i/2e) = pi/e along P; the contribution from the
# semicircle tends to zero as R tends to infinity; thus the integral
# along the real axis is the whole path integral, or pi/e.

# We can now reproduce this result analytically.  First, choose an R:
R <- 400

# now define P.  First, the semicircle, u1:
u1     <- function(x){R*exp(pi*1i*x)}
u1dash <- function(x){R*pi*1i * exp(pi*1i*x)}

# and now the straight part along the real axis, u2:
u2     <- function(x){R*(2*x-1)}
u2dash <- function(x){R*2}

# Better define the function:
f <- function(z){exp(1i*z) / (1+z^2)}

# OK, now carry out the path integral.  I'll do it explicitly, but note
```

```
# that the contribution from the first integral should be small:

answer.approximate <-
    integrate.contour(f, u1, u1dash) +
    integrate.contour(f, u2, u2dash)

# And compare with the analytical value:
answer.exact <- pi/exp(1)
abs(answer.approximate - answer.exact)


# Now try the same thing but integrating over a triangle, using
# integrate.segments().  Use a path P' with base from -R to +R along the
# real axis, closed by two straight segments, one from +R to 1i*R, the
# other from 1i*R to -R:

abs(integrate.segments(f, c(-R, R, 1i*R))- answer.exact)


# Observe how much better one can do by integrating over a big square
# instead:

abs(integrate.segments(f, c(-R, R, R+1i*R, -R+1i*R))- answer.exact)


# Now in the interests of search engine findability, here is an
# application of Cauchy's integral formula, or Cauchy's formula.  I will
# use it to find sin(0.8):

u     <- function(x){exp(pi*2i*x)}
udash <- function(x){pi*2i * exp(pi*2i*x)}

g <- function(z){sin(z) / (z-0.8)}

a <- 1/(2i*pi) * integrate.contour(g, u, udash)

abs(a - sin(0.8))
```

`near.match` *Are two vectors close to one another?*

Description

Returns `TRUE` if each element of `x` and `y` are "near" one another

Usage

```
near.match(x, y, tol=NULL)
```

Arguments

x	First object
y	Second object
tol	Relative tolerance with default NULL meaning to use machine precision

Author(s)

Robin K. S. Hankin

Examples

```
x <- rep(1,6)
near.match(x, x+rnorm(6)/1e10)
```

---

newton_raphson *Newton Raphson iteration to find roots of equations*

Description

Newton-Raphson iteration to find roots of equations with the emphasis on complex functions

Usage

```
newton_raphson(initial, f, fdash, maxiter, give=TRUE, tol = .Machine$double.eps)
```

Arguments

initial	Starting guess
f	Function for which $f(z) = 0$ is to be solved for z
fdash	Derivative of function (note: Cauchy-Riemann conditions assumed)
maxiter	Maximum number of iterations attempted
give	Boolean, with default TRUE meaning to give output based on that of uniroot() and FALSE meaning to return only the estimated root
tol	Tolerance: iteration stops if $\|f(z)\| < tol$

Details

Bog-standard implementation of the Newton-Raphson algorithm

Value

If argument give is FALSE, returns z with $|f(z)| < tol$; if TRUE, returns a list with elements root (the estimated root), f.root (the function evaluated at the estimated root; should have small modulus), and iter, the number of iterations required.

Note

Previous versions of this function used the misspelling "Rapheson".

Author(s)

Robin K. S. Hankin

Examples

```
# Find the two square roots of 2+i:
f <- function(z){z^2-(2+1i)}
fdash <- function(z){2*z}
newton_raphson( 1.4+0.3i,f,fdash,maxiter=10)
newton_raphson(-1.4-0.3i,f,fdash,maxiter=10)

# Now find the three cube roots of unity:
g <- function(z){z^3-1}
gdash <- function(z){3*z^2}
newton_raphson(-0.5+1i, g, gdash, maxiter=10)
newton_raphson(-0.5-1i, g, gdash, maxiter=10)
newton_raphson(+0.5+0i, g, gdash, maxiter=10)
```

---

nome *Nome in terms of m or k*

Description

Calculates the nome in terms of either m (`nome()`) or k (`nome.k()`).

Usage

```
nome(m)
nome.k(k)
```

Arguments

`m`	Real parameter
`k`	Real parameter with $k = m^2$

Note

The nome is defined as $e^{-i\pi K'/K}$, where K and iK' are the quarter periods (see page 576 of AMS-55). These are calculated using function `K.fun()`.

Author(s)

Robin K. S. Hankin

See Also

K.fun

Examples

```
nome(0.09)  # AMS-55 give 0.00589414 in example 7 on page 581
```

P.laurent *Laurent series for elliptic and related functions*

Description

Laurent series for various functions

Usage

```
        P.laurent(z, g=NULL, tol=0, nmax=80)
    Pdash.laurent(z, g=NULL, nmax=80)
    sigma.laurent(z, g=NULL, nmax=8, give.error=FALSE)
sigmadash.laurent(z, g=NULL, nmax=8, give.error=FALSE)
     zeta.laurent(z, g=NULL, nmax=80)
```

Arguments

z	Primary argument (complex)
g	Vector of length two with g=c(g2,g3)
tol	Tolerance
give.error	In sigma.laurent(), Boolean with default FALSE meaning to return the computed value and TRUE to return the error (as estimated by the sum of the absolute values of the terms along the minor long diagonal of the matrix)
.	
nmax	Number of terms used (or, for sigma(), the size of matrix used)

Author(s)

Robin K. S. Hankin

Examples

```
sigma.laurent(z=1+1i,g=c(0,4))
```

`p1.tau` *Does the Right Thing (tm) when calling g2.fun() and g3.fun()*

Description

Takes vectors and interprets them appropriately for input to `g2.fun()` and `g3.fun()`. Not really intended for the end user.

Usage

```
p1.tau(b)
```

Arguments

`b`	Vector of periods

Details

If `b` is of length two, interpret the elements as ω_1 and ω_2 respectively.

If a two-column matrix, interpret the columns as ω_1 and ω_2 respectively.

Otherwise, interpret as a vector of $\tau = \omega_1/\omega_2$.

Value

Returns a two-component list:

`p1`	First period
`tau`	Period ratio

Author(s)

Robin K. S. Hankin

Examples

```
p1.tau(c(1+1i,1.1+23.123i))
```

parameters *Parameters for Weierstrass's P function*

Description

Calculates the invariants g_2 and g_3, the e-values e_1, e_2, e_3, and the half periods ω_1, ω_2, from any one of them.

Usage

```
parameters(Omega=NULL, g=NULL, description=NULL)
```

Arguments

Omega	Vector of length two, containing the **half periods** (ω_1, ω_2)
g	Vector of length two: (g_2, g_3)
description	string containing "equianharmonic", "lemniscatic", or "pseudolemniscatic", to specify one of A and S's special cases

Value

Returns a list with the following items:

Omega	A complex vector of length 2 giving the fundamental half periods ω_1 and ω_2. Notation follows Chandrasekharan: half period ω_1 is 0.5 times a (nontrivial) period of minimal modulus, and ω_2 is 0.5 times a period of smallest modulus having the property ω_2/ω_1 not real. The relevant periods are made unique by the further requirement that $\mathrm{Re}(\omega_1) > 0$, and $\mathrm{Im}(\omega_2) > 0$; but note that this often results in sign changes when considering cases on boundaries (such as real g_2 and g_3). **Note** Different definitions exist for ω_3! A and S use $\omega_3 = \omega_2 - \omega_1$, while Whittaker and Watson (eg, page 443), and Mathematica, have $\omega_1 + \omega_2 + \omega_3 = 0$
q	The nome. Here, $q = e^{\pi i \omega_2/\omega_1}$.
g	Complex vector of length 2 holding the invariants
e	Complex vector of length 3. Here e_1, e_2, and e_3 are defined by $$e_1 = \wp(\omega_1/2)m \qquad e_2 = \wp(\omega_2/2), \qquad e_3 = \wp(\omega_3/2)$$ where ω_3 is defined by $\omega_1 + \omega_2 + \omega_3 = 0$. Note that the es are also defined as the three roots of $x^3 - g_2 x - g_3 = 0$; but this method cannot be used in isolation because the roots may be returned in the wrong order.
Delta	The quantity $g_2^3 - 27 g_3^2$, often denoted Δ

Eta	Complex vector of length 3 often denoted η. Here $\eta = (\eta_1, \eta_2, \eta_3)$ are defined in terms of the Weierstrass zeta function with $\eta_i = \zeta(\omega_i)$ for $i = 1, 2, 3$. Note that the name of this element is capitalized to avoid confusion with function `eta()`
is.AnS	Boolean, with `TRUE` corresponding to real invariants, as per Abramowitz and Stegun
given	character string indicating which parameter was supplied. Currently, one of "o" (omega), or "g" (invariants)

Author(s)

Robin K. S. Hankin

Examples

```
## Example 6, p665, LHS
parameters(g=c(10,2))
## Not clear to me how AMS-55 justify 7 sig figs

## Example 7, p665, RHS
a <- parameters(g=c(7,6)) ;  attach(a)
c(omega2=Omega[1],omega2dash=Omega[1]+Omega[2]*2)


 ## verify 18.3.37:
 Eta[2]*Omega[1]-Eta[1]*Omega[2]   #should be close to pi*1i/2


## from Omega to g and and back;
## following should be equivalent to c(1,1i):
 parameters(g=parameters(Omega=c(1,1i))$g)$Omega
```

---

pari *Wrappers for PARI functions*

Description

Wrappers for the three elliptic functions of PARI

Usage

```
P.pari(z,Omega,pari.fun="ellwp",numerical=TRUE)
```

Arguments

z	Complex argument
Omega	Half periods
pari.fun	String giving the name of the function passed to PARI. Values of ellwp, ellsigma, and ellzeta, are acceptable here for the Weierstrass $\wp$ function, the σ function, and the ζ function respectively
numerical	Boolean with default TRUE meaning to return the complex value returned by PARI, and FALSE meaning to return the ascii string returned by PARI

Details

This function calls PARI via an R system() call.

Value

Returns an object with the same attributes as z.

Note

Function translates input into, for example, "ellwp([1+1*I,2+3*I],1.111+5.1132*I)" and pipes this string directly into gp.

The PARI system clearly has more powerful syntax than the basic version that I'm using here, but I can't (for example) figure out how to vectorize any of the calls.

Author(s)

Robin K. S. Hankin

References

http://www.parigp-home.de/

Examples

```
## Not run:  #this in a dontrun environment because it requires pari/gp
z  <- seq(from=1,to=3+2i,len=34)
p <- c(1,1i)
plot(abs(P.pari(z=z,Omega=p) - P(z=z,Omega=p)))
plot(zeta(z=z,params=parameters(Omega=p))- P.pari(z=z,Omega=c(p),pari.fun="ellzeta"))


## End(Not run)
```

sn *Jacobi form of the elliptic functions*

Description

Jacobian elliptic functions

Usage

```
ss(u,m, ...)
sc(u,m, ...)
sn(u,m, ...)
sd(u,m, ...)
cs(u,m, ...)
cc(u,m, ...)
cn(u,m, ...)
cd(u,m, ...)
ns(u,m, ...)
nc(u,m, ...)
nn(u,m, ...)
nd(u,m, ...)
ds(u,m, ...)
dc(u,m, ...)
dn(u,m, ...)
dd(u,m, ...)
```

Arguments

u	Complex argument
m	Parameter
...	Extra arguments, such as `maxiter`, passed to `theta.?()`

Details

All sixteen Jacobi elliptic functions.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of mathematical functions*. New York: Dover

See Also

theta

Examples

```
#Example 1, p579:
nc(1.9965,m=0.64)
# (some problem here)

# Example 2, p579:
dn(0.20,0.19)

# Example 3, p579:
dn(0.2,0.81)

# Example 4, p580:
cn(0.2,0.81)

# Example 5, p580:
dc(0.672,0.36)

# Example 6, p580:
Theta(0.6,m=0.36)

# Example 7, p581:
cs(0.53601,0.09)

# Example 8, p581:
sn(0.61802,0.5)

#Example 9, p581:
sn(0.61802,m=0.5)

#Example 11, p581:
cs(0.99391,m=0.5)
# (should be 0.75 exactly)

#and now a pretty picture:

n <- 300
K <- K.fun(1/2)
f <- function(z){1i*log((z-1.7+3i)*(z-1.7-3i)/(z+1-0.3i)/(z+1+0.3i))}
# f <- function(z){log((z-1.7+3i)/(z+1.7+3i)*(z+1-0.3i)/(z-1-0.3i))}
x <- seq(from=-K,to=K,len=n)
y <- seq(from=0,to=K,len=n)
z <- outer(x,1i*y,"+")

view(x, y, f(sn(z,m=1/2)), nlevels=44, imag.contour=TRUE,
     real.cont=TRUE, code=1, drawlabels=FALSE,
     main="Potential flow in a rectangle",axes=FALSE,xlab="",ylab="")
rect(-K,0,K,K,lwd=3)
```

sqrti *Generalized square root*

Description

Square root wrapper that keeps answer real if possible, coerces to complex if not.

Usage

```
sqrti(x)
```

Arguments

x	Vector to return square root of

Author(s)

Robin K. S. Hankin

Examples

```
sqrti(1:10)  #real
sqrti(-10:10) #coerced to complex (compare sqrt(-10:10))
sqrti(1i+1:10) #complex anyway
```

theta *Jacobi theta functions 1-4*

Description

Computes Jacobi's four theta functions for complex z in terms of the parameter m or q.

Usage

```
theta1  (z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta2  (z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta3  (z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta4  (z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta.00(z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta.01(z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta.10(z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
theta.11(z, ignore=NULL, m=NULL, q=NULL, give.n=FALSE, maxiter=30, miniter=3)
Theta (u, m, ...)
Theta1(u, m, ...)
H (u, m, ...)
H1(u, m, ...)
```

Arguments

z, u	Complex argument of function
ignore	Dummy variable whose intention is to force the user to name the second argument either m or q
m	Does not seem to have a name. The variable is introduced in section 16.1, p569
q	The nome q, defined in section 16.27, p576
give.n	Boolean with default FALSE meaning to return the function evaluation, and TRUE meaning to return a two element list, with first element the function evaluation, and second element the number of iterations used
maxiter	Maximum number of iterations used. Note that the series generally converge very quickly
miniter	Minimum number of iterations to guard against premature exit if an addend is zero exactly
...	In functions that take it, extra arguments passed to theta1() et seq; notably, maxiter

Details

Functions theta.00() et seq are just wrappers for theta1() et seq, following Whittaker and Watson's terminology on p487; the notation does not appear in Abramowitz and Stegun.

- theta.11() = theta1()
- theta.10() = theta2()
- theta.00() = theta3()
- theta.01() = theta4()

Value

Returns a complex-valued object with the same attributes as either z, or (m or q), whichever wasn't recycled.

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of mathematical functions*. New York: Dover

See Also

theta.neville

Examples

```
m <- 0.5
derivative <- function(small){(theta1(small,m=m)-theta1(0,m=m))/small}
right.hand.side1 <- theta2(0,m=m)*theta3(0,m=m)*theta4(0,m=m)
right.hand.side2 <- theta1.dash.zero(m)

derivative(1e-5) - right.hand.side1   # should be zero
derivative(1e-5) - right.hand.side2   # should be zero
```

`theta.neville` *Neville's form for the theta functions*

Description

Neville's notation for theta functions as per section 16.36 of Abramowitz and Stegun.

Usage

```
theta.s(u, m, method = "16.36.6", ...)
theta.c(u, m, method = "16.36.6", ...)
theta.d(u, m, method = "16.36.7", ...)
theta.n(u, m, method = "16.36.7", ...)
```

Arguments

`u`	Primary complex argument
`m`	Real parameter
`method`	Character string corresponding to A and S's equation numbering scheme
`...`	Extra arguments passed to the method function, such as `maxiter`

Details

I reproduce the relevant sections of AMS-55 here, for convenience:

$$\text{16.36.6a} \quad \vartheta_s(u) = \frac{2K\vartheta_1(v)}{\vartheta_1'(0)}$$

$$\text{16.36.6b} \quad \vartheta_c(u) = \frac{\vartheta_2(v)}{\vartheta_2(0)}$$

$$\text{16.36.7a} \quad \vartheta_d(u) = \frac{\vartheta_3(v)}{\vartheta_3(0)}$$

$$\text{16.36.7b} \quad \vartheta_n(u) = \frac{\vartheta_4(v)}{\vartheta_4(0)}$$

$$\text{16.37.1} \quad \vartheta_s(u) = \left(\frac{16q}{mm_1}\right)^{1/6} \sin v \prod_{n=1}^{\infty}\left(1-2q^{2n}\cos 2v + q^{4n}\right)$$

$$16.37.2 \qquad \vartheta_c(u) = \left(\frac{16qm_1^{1/2}}{m}\right)^{1/6} \cos v \prod_{n=1}^{\infty}\left(1+2q^{2n}\cos 2v + q^{4n}\right)$$

$$16.37.3 \qquad \vartheta_d(u) = \left(\frac{mm_1}{16q}\right)^{1/12} \prod_{n=1}^{\infty}\left(1+2q^{2n-1}\cos 2v + q^{4n-2}\right)$$

$$16.37.4 \qquad \vartheta_n(u) = \left(\frac{m}{16qm_1^2}\right)^{1/12} \prod_{n=1}^{\infty}\left(1-2q^{2n-1}\cos 2v + q^{4n-2}\right)$$

(in the above we have $v = \pi u/(2K)$ and $q = q(m)$).

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of mathematical functions*. New York: Dover

Examples

```
#Figure 16.4.
m <- 0.5
K <- K.fun(m)
Kdash <- K.fun(1-m)
x <- seq(from=0,to=4*K,len=100)
plot  (x/K,theta.s(x,m=m),type="l",lty=1,main="Figure 16.4, p578")
points(x/K,theta.n(x,m=m),type="l",lty=2)
points(x/K,theta.c(x,m=m),type="l",lty=3)
points(x/K,theta.d(x,m=m),type="l",lty=4)
abline(0,0)



#plot a graph of something that should be zero:
 x <- seq(from=-4,to=4,len=55)
 plot(x,(e16.37.1(x,0.5)-theta.s(x,0.5)),pch="+",main="error: note vertical scale")

#now table 16.1 on page 582 et seq:
 alpha <- 85
 m <- sin(alpha*pi/180)^2
## K <- ellint_Kcomp(sqrt(m))
 K <- K.fun(m)
 u <- K/90*5*(0:18)
 u.deg <- round(u/K*90)
 cbind(u.deg,"85"=theta.s(u,m))      # p582, last col.
 cbind(u.deg,"85"=theta.n(u,m))      # p583, last col.
```

theta1.dash.zero *Derivative of theta1*

Description

Calculates θ_1' as a function of either m or k

Usage

```
theta1.dash.zero(m, ...)
theta1.dash.zero.q(q, ...)
```

Arguments

m	real parameter
q	Real parameter
...	Extra arguments passed to theta1() et seq, notably maxiter

Author(s)

Robin K. S. Hankin

Examples

```
#Now, try and get 16.28.6, p576: theta1dash=theta2*theta3*theta4:

m <- 0.5
derivative <- function(small){(theta1(small,m=m)-theta1(0,m=m))/small}
right.hand.side <-  theta2(0,m=m)*theta3(0,m=m)*theta4(0,m=m)
derivative(1e-7)-right.hand.side
```

theta1dash *Derivatives of theta functions*

Description

First, second, and third derivatives of the theta functions

Usage

```
theta1dash(z, ignore = NULL, m = NULL, q = NULL, give.n = FALSE,
     maxiter = 30, miniter=3)
theta1dashdash(z, ignore = NULL, m = NULL, q = NULL, give.n = FALSE,
     maxiter = 30,miniter=3)
theta1dashdashdash(z, ignore = NULL, m = NULL, q = NULL, give.n = FALSE,
     maxiter = 30,miniter=3)
```

Arguments

z	Primary complex argument
ignore	Dummy argument to force the user to name the next argument either m or q
m	m as documented in theta1()
q	q as documented in theta1()
give.n	Boolean with default FALSE meaning to return the function evaluation, and TRUE meaning to return a two element list, with first element the function evaluation, and second element the number of iterations used
maxiter	Maximum number of iterations
miniter	Minimum number of iterations to guard against premature exit if an addend is zero exactly

Details

Uses direct expansion as for theta1() et seq

Author(s)

Robin K. S. Hankin

References

M. Abramowitz and I. A. Stegun 1965. *Handbook of Mathematical Functions.* New York, Dover

See Also

theta

Examples

```
m <- 0.3+0.31i
z <- seq(from=1,to=2+1i,len=7)
delta <- 0.001
deriv.numer <- (theta1dashdash(z=z+delta,m=m) - theta1dashdash(z=z,m=m))/delta
deriv.exact <- theta1dashdashdashdash(z=z+delta/2,m=m)
abs(deriv.numer-deriv.exact)
```

`unimodular` *Unimodular matrices*

Description

Systematically generates unimodular matrices; numerical verification of a function's unimodularness

Usage

```
unimodular(n)
unimodularity(n,o, FUN, ...)
```

Arguments

`n`	Maximum size of entries of matrices
`o`	Two element vector
`FUN`	Function whose unimodularity is to be checked
`...`	Further arguments passed to `FUN`

Details

Here, a 'unimodular' matrix is of size 2×2, with integer entries and a determinant of unity.

Value

Function `unimodular()` returns an array `a` of dimension `c(2,2,u)` (where `u` is a complicated function of `n`). Thus 3-slices of `a` (that is, `a[,,i]`) are unimodular.

Function `unimodularity()` returns the result of applying `FUN()` to the unimodular transformations of `o`. The function returns a vector of length `dim(unimodular(n))[3]`; if `FUN()` is unimodular and roundoff is neglected, all elements of the vector should be identical.

Note

In function `as.primitive()`, a 'unimodular' may have determinant minus one.

Author(s)

Robin K. S. Hankin

See Also

`as.primitive`

Examples

```
unimodular(3)

o <- c(1,1i)
plot(abs(unimodularity(3,o,FUN=g2.fun,maxiter=100)-g2.fun(o)))
```

---

view *Visualization of complex functions*

Description

Visualization of complex functions using colour maps and contours

Usage

```
view(x, y, z, scheme = 0, real.contour = TRUE, imag.contour = real.contour,
default = 0, col="black", r0=1, power=1, show.scheme=FALSE, ...)
```

Arguments

x, y	Vectors showing real and imaginary components of complex plane; same functionality as arguments to `image()`
z	Matrix of complex values to be visualized
scheme	Visualization scheme to be used. A numeric value is interpreted as one of the (numbered) provided schemes; see source code for details, as I add new schemes from time to time and the code would in any case dominate anything written here. A default of zero corresponds to Thaller (1998): see references. For no colour (ie a white background), set `scheme` to a negative number. If `scheme` does not correspond to a built-in function, the `switch()` statement "drops through" and provides a white background (use this to show just real or imaginary contours; a value of -1 will always give this behaviour) If not numeric, `scheme` is interpreted as a function that produces a colour; see examples section below. See details section for some tools that make writing such functions easier
real.contour, imag.contour	Boolean with default `TRUE` meaning to draw contours of constant $Re(z)$ (resp: $Im(z)$) and `FALSE` meaning not to draw them
default	Complex value to be assumed for colouration, if `z` takes `NA` or infinite values; defaults to zero. Set to `NA` for no substitution (ie plot `z` "as is"); usually a bad idea
col	Colour (sent to `contour()`)
r0	If `scheme=0`, radius of Riemann sphere as used by Thaller

power	Defines a slight generalization of Thaller's scheme. Use high values to emphasize areas of high modulus (white) and low modulus (black); use low values to emphasize the argument over the whole of the function's domain. This argument is also applied to some of the other schemes where it makes sense
show.scheme	Boolean, with default FALSE meaning to perform as advertized and visualize a complex function; and TRUE meaning to return the function corresponding to the value of argument scheme
...	Extra arguments passed to image() and contour()

Details

The examples given for different values of scheme are intended as examples only: the user is encouraged to experiment by passing homemade colour schemes (and indeed to pass such schemes to the author).

Scheme 0 implements the ideas of Thaller: the complex plane is mapped to the Riemann sphere, which is coded with the North pole white (indicating a pole) and the South Pole black (indicating a zero). The equator (that is, complex numbers of modulus r0) maps to colours of maximal saturation.

Function view() includes several tools that simplify the creation of suitable functions for passing to scheme.

These include:

breakup(): Breaks up a continuous map: function(x){ifelse(x>1/2,3/2-x,1/2-x)}

g(): maps positive real to $[0,1]$: function(x){0.5+atan(x)/pi}

scale(): scales range to $[0,1]$: function(x){(x-min(x))/(max(x)-min(x))}

wrap(): wraps phase to $[0,1]$: function(x){1/2+x/(2*pi)}

Note

Additional ellipsis arguments are given to both image() and contour() (typically, nlevels). The resulting warning() from one or other function is suppressed by suppressWarnings().

Author(s)

Robin K. S. Hankin

References

B. Thaller 1998. *Visualization of complex functions*, The Mathematica Journal, 7(2):163–180

Examples

```
n <- 100
x <- seq(from=-4,to=4,len=n)
y <- x
z <- outer(x,1i*y,"+")
view(x,y,limit(1/z),scheme=2)
view(x,y,limit(1/z),scheme=18)
```

```
view(x,y,limit(1/z+1/(z-1-1i)^2),scheme=5)
view(x,y,limit(1/z+1/(z-1-1i)^2),scheme=17)

view(x,y,log(0.4+0.7i+log(z/2)^2),main="Some interesting cut lines")


view(x,y,z^2,scheme=15,main="try finer resolution")
view(x,y,sn(z,m=1/2+0.3i),scheme=6,nlevels=33,drawlabels=FALSE)

view(x,y,limit(P(z,c(1+2.1i,1.3-3.2i))),scheme=2,nlevels=6,drawlabels=FALSE)
view(x,y,limit(Pdash(z,c(0,1))),scheme=6,nlevels=7,drawlabels=FALSE)
view(x,x,limit(zeta(z,c(1+1i,2-3i))),nlevels=6,scheme=4,col="white")

# Now an example with a bespoke colour function:

 fun <- function(z){hcl(h=360*wrap(Arg(z)),c= 100 * (Mod(z) < 1))}
 view(x,x,limit(zeta(z,c(1+1i,2-3i))),nlevels=6,scheme=fun)

view(scheme=10, show.scheme=TRUE)
```

`WeierstrassP` *Weierstrass P and related functions*

Description

Weierstrass elliptic function and its derivative, Weierstrass sigma function, and the Weierstrass zeta function

Usage

```
P(z, g=NULL, Omega=NULL, params=NULL, use.fpp=TRUE, give.all.3=FALSE, ...)
Pdash(z, g=NULL, Omega=NULL, params=NULL, use.fpp=TRUE, ...)
sigma(z, g=NULL, Omega=NULL, params=NULL, use.theta=TRUE, ...)
zeta(z, g=NULL, Omega=NULL, params=NULL, use.fpp=TRUE, ...)
```

Arguments

`z`	Primary complex argument
`g`	Invariants `g=c(g2,g3)`. Supply exactly one of (`g`, `Omega`, `params`)
`Omega`	Half periods
`params`	Object with class "`parameters`" (typically provided by `parameters()`)
`use.fpp`	Boolean, with default `TRUE` meaning to calculate $\wp(z^C)$ where z^C is congruent to z in the period lattice. The default means that accuracy is greater for large z but has the deficiency that slight discontinuities may appear near parallelogram boundaries
`give.all.3`	Boolean, with default `FALSE` meaning to return $\wp(z)$ and `TRUE` meaning to return the other forms given in equation 18.10.5, p650. Use `TRUE` to check for accuracy

use.theta	Boolean, with default TRUE meaning to use theta function forms, and FALSE meaning to use a Laurent expansion. Usually, the theta function form is faster, but not always
...	Extra parameters passed to theta1() and theta1dash()

Note

In this package, function sigma() is the Weierstrass sigma function. For the number theoretic divisor function also known as "sigma", see divisor().

Author(s)

Robin K. S. Hankin

References

R. K. S. Hankin. *Introducing Elliptic, an R package for Elliptic and Modular Functions*. Journal of Statistical Software, Volume 15, Issue 7. February 2006.

Examples

```
## Example 8, p666, RHS:
P(z=0.07 + 0.1i,g=c(10,2))

## Example 8, p666, RHS:
P(z=0.1 + 0.03i,g=c(-10,2))
## Right answer!

## Compare the Laurent series, which also gives the Right Answer (tm):
 P.laurent(z=0.1 + 0.03i,g=c(-10,2))


## Now a nice little plot of the zeta function:
x <- seq(from=-4,to=4,len=100)
z <- outer(x,1i*x,"+")
view(x,x,limit(zeta(z,c(1+1i,2-3i))),nlevels=6,scheme=1)


#now figure 18.5, top of p643:
p <- parameters(Omega=c(1+0.1i,1+1i))
n <- 40

f <- function(r,i1,i2=1)seq(from=r+1i*i1, to=r+1i*i2,len=n)
g <- function(i,r1,r2=1)seq(from=1i*i+r1,to=1i*i+2,len=n)

solid.lines <-
  c(
    f(0.1,0.5),NA,
    f(0.2,0.4),NA,
    f(0.3,0.3),NA,
    f(0.4,0.2),NA,
    f(0.5,0.0),NA,
```

```
    f(0.6,0.0),NA,
    f(0.7,0.0),NA,
    f(0.8,0.0),NA,
    f(0.9,0.0),NA,
    f(1.0,0.0)
    )
dotted.lines <-
  c(
    g(0.1,0.5),NA,
    g(0.2,0.4),NA,
    g(0.3,0.3),NA,
    g(0.4,0.2),NA,
    g(0.5,0.0),NA,
    g(0.6,0.0),NA,
    g(0.7,0.0),NA,
    g(0.8,0.0),NA,
    g(0.9,0.0),NA,
    g(1.0,0.0),NA
    )

plot(P(z=solid.lines,params=p),xlim=c(-4,4),ylim=c(-6,0),type="l",asp=1)
lines(P(z=dotted.lines,params=p),xlim=c(-4,4),ylim=c(-6,0),type="l",lty=2)
```

Index

* **Cauchy's formula**
  myintegrate, 29
* **Cauchy's integral theorem**
  myintegrate, 29
* **Cauchy's theorem**
  myintegrate, 29
* **Complex integration**
  myintegrate, 29
* **Contour integration**
  myintegrate, 29
* **Dedekind's valenz function J**
  J, 23
* **Dedekind's valenz function**
  J, 23
* **Dedekind**
  J, 23
* **Elliptic functions**
  WeierstrassP, 51
* **Jacobi elliptic functions**
  sn, 40
* **Jacobi's elliptic functions**
  sn, 40
* **Jacobian elliptic functions**
  sn, 40
* **Klein's invariant function**
  J, 23
* **Klein's modular function J**
  J, 23
* **Klein's modular function**
  J, 23
* **Multiplicative functions**
  divisor, 10
* **Neville's theta functions**
  theta.neville, 44
* **Path integration**
  myintegrate, 29
* **Residue theorem**
  myintegrate, 29
* **Thaller**
  view, 49
* **Weierstrass P function**
  WeierstrassP, 51
* **Weierstrass elliptic function**
  WeierstrassP, 51
* **Weierstrass sigma function**
  WeierstrassP, 51
* **Weierstrass zeta function**
  WeierstrassP, 51
* **Weierstrass**
  WeierstrassP, 51
* **array**
  as.primitive, 6
  e16.28.1, 12
  farey, 17
  theta, 42
  unimodular, 48
* **lambda function**
  J, 23
* **math**
  amn, 5
  ck, 7
  congruence, 8
  coqueraux, 9
  divisor, 10
  e18.10.9, 13
  e1e2e3, 14
  equianharmonic, 15
  eta, 16
  fpp, 18
  g.fun, 19
  half.periods, 21
  J, 23
  K.fun, 24
  latplot, 25
  lattice, 26
  limit, 26
  massage, 27
  misc, 28

mob, 28
myintegrate, 29
near.match, 32
newton_raphson, 33
nome, 34
P.laurent, 35
p1.tau, 36
parameters, 37
pari, 38
sn, 40
sqrti, 42
theta.neville, 44
theta1.dash.zero, 46
theta1dash, 46
view, 49
WeierstrassP, 51
∗ **package**
elliptic-package, 2
%mob% *(*mob*)*, 28
18.5.7 *(*amn*)*, 5
18.5.8 *(*amn*)*, 5

amn, 5
as.primitive, 6, *48*

cc *(*sn*)*, 40
cd *(*sn*)*, 40
ck, 7
cn *(*sn*)*, 40
congruence, 8
coqueraux, 9
cs *(*sn*)*, 40

dc *(*sn*)*, 40
dd *(*sn*)*, 40
divisor, 10
dn *(*sn*)*, 40
ds *(*sn*)*, 40

e16.1.1 *(*K.fun*)*, 24
e16.27.1 *(*theta*)*, 42
e16.27.2 *(*theta*)*, 42
e16.27.3 *(*theta*)*, 42
e16.27.4 *(*theta*)*, 42
e16.28.1, 12
e16.28.2 *(*e16.28.1*)*, 12
e16.28.3 *(*e16.28.1*)*, 12
e16.28.4 *(*e16.28.1*)*, 12
e16.28.5 *(*e16.28.1*)*, 12
e16.28.6 *(*theta1.dash.zero*)*, 46
e16.31.1 *(*theta*)*, 42
e16.31.2 *(*theta*)*, 42
e16.31.3 *(*theta*)*, 42
e16.31.4 *(*theta*)*, 42
e16.36.3 *(*sn*)*, 40
e16.36.6 *(*theta.neville*)*, 44
e16.36.6a *(*theta.neville*)*, 44
e16.36.6b *(*theta.neville*)*, 44
e16.36.7 *(*theta.neville*)*, 44
e16.36.7a *(*theta.neville*)*, 44
e16.36.7b *(*theta.neville*)*, 44
e16.37.1 *(*theta.neville*)*, 44
e16.37.2 *(*theta.neville*)*, 44
e16.37.3 *(*theta.neville*)*, 44
e16.37.4 *(*theta.neville*)*, 44
e16.38.1 *(*theta.neville*)*, 44
e16.38.2 *(*theta.neville*)*, 44
e16.38.3 *(*theta.neville*)*, 44
e16.38.4 *(*theta.neville*)*, 44
e18.1.1 *(*g.fun*)*, 19
e18.10.1 *(*WeierstrassP*)*, 51
e18.10.10 *(*e18.10.9*)*, 13
e18.10.10a *(*e18.10.9*)*, 13
e18.10.10b *(*e18.10.9*)*, 13
e18.10.11 *(*e18.10.9*)*, 13
e18.10.11a *(*e18.10.9*)*, 13
e18.10.11b *(*e18.10.9*)*, 13
e18.10.12 *(*e18.10.9*)*, 13
e18.10.12a *(*e18.10.9*)*, 13
e18.10.12b *(*e18.10.9*)*, 13
e18.10.2 *(*WeierstrassP*)*, 51
e18.10.3 *(*WeierstrassP*)*, 51
e18.10.4 *(*WeierstrassP*)*, 51
e18.10.5 *(*WeierstrassP*)*, 51
e18.10.6 *(*WeierstrassP*)*, 51
e18.10.7 *(*WeierstrassP*)*, 51
e18.10.9, 13
e18.10.9a *(*e18.10.9*)*, 13
e18.10.9b *(*e18.10.9*)*, 13
e18.3.1 *(*e1e2e3*)*, 14
e18.3.3 *(*parameters*)*, 37
e18.3.37 *(*parameters*)*, 37
e18.3.38 *(*parameters*)*, 37
e18.3.39 *(*parameters*)*, 37
e18.3.5 *(*parameters*)*, 37
e18.3.7 *(*e1e2e3*)*, 14
e18.3.8 *(*e1e2e3*)*, 14

e18.5.1 (P.laurent), 35
e18.5.16 (ck), 7
e18.5.2 (ck), 7
e18.5.3 (ck), 7
e18.5.4 (P.laurent), 35
e18.5.5 (P.laurent), 35
e18.5.6 (P.laurent), 35
e18.7.4 (parameters), 37
e18.7.5 (parameters), 37
e18.7.7 (parameters), 37
e18f.5.3 (P.laurent), 35
e1e2e3, 14
eee.cardano (e1e2e3), 14
elliptic (elliptic-package), 2
elliptic-package, 2
equianharmonic, 15
eta, 16

factorize (divisor), 10
farey, *17*, 17
fpp, 18

g.fun, 19
g2.fun (g.fun), 19
g3.fun (g.fun), 19
GP (pari), 38
Gp (pari), 38
gp (pari), 38

H (theta), 42
H1 (theta), 42
half.periods, 21

Im<- (misc), 28
integrate.contour (myintegrate), 29
integrate.segments (myintegrate), 29
is.primitive (as.primitive), 6

J, 23

K.fun, 24, *35*

lambda (J), 23
latplot, 25
lattice, 26
lemniscatic (equianharmonic), 15
limit, 26
liouville (divisor), 10

massage, 27
misc, 28
mn (fpp), 18
mob, 28
mobius (divisor), 10
myintegrate, 29

nc (sn), 40
nd (sn), 40
near.match, 32
Newton_Raphson (newton_raphson), 33
Newton_raphson (newton_raphson), 33
newton_Raphson (newton_raphson), 33
newton_raphson, 33
nn (sn), 40
nome, 34
ns (sn), 40

P (WeierstrassP), 51
P.laurent, *7*, 35
P.pari (pari), 38
p1.tau, 36
parameters, *16*, 37
PARI (pari), 38
pari, 38
Pdash (WeierstrassP), 51
Pdash.laurent (P.laurent), 35
primes (divisor), 10
pseudolemniscatic (equianharmonic), 15

Re<- (misc), 28
residue (myintegrate), 29

sc (sn), 40
sd (sn), 40
sigma (WeierstrassP), 51
sigma.laurent (P.laurent), 35
sigmadash.laurent (P.laurent), 35
sn, 40
sqrti, 42
ss (sn), 40

Theta (theta), 42
theta, *40*, 42, *47*
theta.c (theta.neville), 44
theta.d (theta.neville), 44
theta.n (theta.neville), 44
theta.neville, *43*, 44
theta.s (theta.neville), 44
Theta1 (theta), 42

theta1 (theta), 42
theta1.dash.zero, 46
theta1dash, 46
theta1dashdash (theta1dash), 46
theta1dashdashdash (theta1dash), 46
theta2 (theta), 42
theta3 (theta), 42
theta4 (theta), 42
totient (divisor), 10

unimodular, *9*, *18*, *29*, 48
unimodularity (unimodular), 48

view, 49

WeierstrassP, 51

zeta (WeierstrassP), 51
zeta.laurent (P.laurent), 35