A Computational Basis for Higher-dimensional Computational Geometry *

Kurt Mehlhorn Stefan Näher Stefan Schirra Michael Seel
Christian Uhrig

May 21, 1996

Abstract

We specify and implement a kernel for computational geometry in arbitrary finite dimensional space. The kernel provides points, vectors, directions, hyperplanes, segments, rays, lines, affine transformations, and operations connecting these types. Points have rational coordinates, hyperplanes have rational coefficients, and analogous statements hold for the other types. We therefore call our types *rat_point*, *rat_vector*, *rat_direction*, *rat_hyperplane*, *rat_segment*, *rat_ray* and *rat_line*. All geometric primitives are *exact*, i.e., they do not incur rounding error (because they are implemented using rational arithmetic) and always produce the correct result. To this end we provide types *integer_vector* and *integer_matrix* which realize exact linear algebra over the integers.

The kernel is submitted to the CGAL-Consortium as a proposal for its higher-dimensional geometry kernel and will become part of the LEDA platform for combinatorial and geometric computing.

*This work was carried out as part of EU LTR CGAL.

Contents

1 Introduction

We define and implement the basic data types of higher-dimensional computational geometry: points, vectors, directions, hyperplanes, segments, rays, lines, affine transformations, and operations connecting these types. Points have rational coordinates, hyperplanes have rational coefficients, and analogous statements hold for the other types. We therefore call our types *rat_point*, *rat_vector*, *rat_direction*, *rat_hyperplane*, *rat_segment*, *rat_ray* and *rat_line*. All geometric primitives are *exact*, i.e., they do not incur rounding error (because they are implemented using rational arithmetic) and always produce the correct result. To this end we provide types *integer_vector* and *integer_matrix* which realize exact linear algebra over the integers.

We need to briefly review the basics of analytical geometry. We use d to denote the dimension of the ambient space and assume that our space is equipped with a standard Cartesian coordinate system. We identify any point p with its cartesian coordinate vector $p = (p_0, \ldots, p_{d-1})$, where the p_i, $0 \leq i < d$, are rational numbers. We store a point by homogenous coordinates $(h_0, \ldots, h_d)$ where $p_i = h_i/h_d$ for all i, $0 \leq i < d$, and the h_i's are integer (LEDA type integer). The homogenizing coordinate h_d is always positive.

Points, vectors, and directions are closely related but nevertheless clearly distinct types. In order to work out the relationship it is usefull to identify a point with an arrow extending from the origin (= an arbitrary but fixed point) to the point. In this view a point is an arrow attached to the origin. A vector is an arrow that is allowed to float freely in space, more precisely, a vector is an equivalence class of arrows where two arrows are equivalent if one can be moved into the other by a translation of space. Points and vectors can be combined by some arithmetical operations. For two points p and q the difference $p - q$ is a vector (= the equivalence class of arrows containing the arrow extending from q to p) and for a point p and a vector v, $p + v$ is a point. All operations of linear algebra apply to vectors, i.e., vectors can be stretched and shrunk (by multiplication with a scalar) and inner and cross product applies to them. On the other hand, geometric tests like collinearity, only apply to points.

A direction is also an equivalence class of arrows, where two arrows are equivalent if one can be moved into the other by a translation of space followed by stretching or shrinking. Alternatively, we may view a direction as a point on the unit sphere. In two-dimensional space directions correspond to angles.

The default dimension of all objects is 2. This means that a call to the standard constructor *rat_point*() delivers an instance of type rat_point for planar geometry. A point in d-dimensional space is constructed by *p*(*integer_vector c*, *integer D*) or *p*(*integer_vector c*). In addition there are standard static initialization operations *d2* and *d3* which allow comfortable creation of objects for the dimensions 2 and 3.

With respect to the user interface we can group together points, vectors, directions and on the other hand segments, rays and lines. For the first group there are common operations like [] and *coord*() to access cartesian coordinates and *hcoord*() to access homogenous coordinates. Conversions within the first group can be made by operations with prefix *to_*. Thus to interpret a point p as its coordinate vector with respect to the origin one can use *p.to_rat_vector*() or *to_rat_vector*(*p*). For the second group there are similar operations to access the coordinates of the determining pair of points.

The input and output operators of group one and hyperplane objects work on *integer* tuples $(i_0, \ldots, i_{d-1}, i_d)$ with $d+1$ components (homogenous representation). This extends

to a pair of points with syntax $[(i_0, \ldots, i_d) === (j_0, \ldots, j_d)]$ for the objects of the second group.

There are many operations that are only available for two-dimensional objects. Some of these make only sense for two-dimensional objects, e.g., the *right_turn* predicate for points, and others come with a special syntax for d-dimensional predicates, e.g., the orientation predicate for $(d+1)$-tuples of points. We have three reasons for this verbosity: It allows a more natural mode of expression, it (essentially) guarantees upward compatibility with the previous $2d$ rational geometry module of LEDA, and it allows more efficient implementation of the two-dimensional operations.

For all of our basic geometric types we have affine transformations which can be used by a call to the common member operation *transform*().

The geometry kernel presented in this paper is submitted to the CGAL-consortium as a proposal for its higher-dimensional geometry kernel and will also become part of LEDA [MN89, MN95, NU95]. The CGAL-consortium is also developing a kernel for two- and three-dimensional geometry [FGK+96, Ove96]. This kernel is parametrized with the underlying arithmetic; the rational arithmetic version offers similar functionality to the kernel described here. The design of our kernel was mainly influenced by three sources: the experiences with the two-dimensional LEDA geometry kernel, our experiences with an experimental higher-dimensional kernel [MZ94], and discussions with the group developing the low-dimensional CGAL-kernel.

This document has two main parts. The first part contains the manual pages of the implemented data types grouped into the linear algebra module and the geometric objects module (the actual geokernel). In the second part we append the complete implementation of each data type in form of CWEB-projects.

2 The Manual Pages

2.1 The Linear Algebra Module

2.1.1 Matrices with Integer Entries (integer_matrix)

1. Definition

An instance of data type *integer_matrix* is a matrix of integer variables. The types integer_matrix and integer_vector together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

2. Creation

integer_matrix $M(int\ n = 0, int\ m = 0)$;

creates an instance M of type integer_matrix. M is initialized to the $n \times m$ - zero matrix.

integer_matrix $M(array\langle integer\_vector\rangle A)$;

creates an instance M of type integer_matrix. Let A be an array of m column-vectors of common dimension n. M is initialized to an $n \times m$ matrix with the columns as specified by A.

3. Operations

int M.dim1( ) returns n, the number of rows of M.

int M.dim2( ) returns m, the number of columns of M.

integer_vector& M.row(*int* i)

returns the i-th row of M (an m-vector).
Precondition: $0 \leq i \leq n-1$.

integer_vector M.col(*int* i) returns the i-th column of M (an n-vector).
Precondition: $0 \leq i \leq m-1$.

integer& $M(int\ i, int\ j)$

returns $M_{i,j}$.
Precondition: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.

integer_matrix $M + M1$ Addition.
Precondition: M.dim1() = $M1$.dim1() and M.dim2() = $M1$.dim2().

integer_matrix $M - M1$ Subtraction.
Precondition: M.dim1() = $M1$.dim1() and M.dim2() = $M1$.dim2().

integer_matrix $M * M1$ Multiplication.
Precondition: M.dim2() = $M1$.dim1().

integer_vector $M *$ *integer_vector* *vec*
Multiplication with vector.
Precondition: M.dim2() = *vec*.dim().

integer_matrix $M *$ *integer* x Multiplication of every entry with integer x.

integer_matrix *integer* $x * M$ Multiplication of every entry with integer x.

integer_matrix identity(*int* n)
returns an n by n identity matrix.

integer_matrix transpose(*integer_matrix* M)
returns M^T ($m \times n$ - matrix).

integer_matrix inverse(*integer_matrix* M, *integer*& D)
returns the inverse matrix of M. More precisely, $1/D$ times the matrix returned is the inverse of M.
Precondition: determinant(M) $\neq 0$.

bool inverse(*integer_matrix* M, *integer_matrix*& *inverse*, *integer*& D, *integer_vector*& c)
determines whether M has an inverse. It also computes either the inverse as $(1/D) \cdot$ *inverse* or a vector c such that $c^T \cdot M = 0$.

integer determinant(*integer_matrix* M, *integer_matrix*& L, *integer_matrix*& U, *array*<*int*>& q, *integer_vector*& c)
returns the determinant D of M and sufficient information to verify that the value of the determinant is correct. If the determinant is zero then c is a vector such that $c^T \cdot M = 0$. If the determinant is non-zero then L and U are lower and upper diagonal matrices respectively and q encodes a permutation matrix Q with $Q(i,j) = 1$ iff $i = q(j)$ such that $L \cdot M \cdot Q = U$, $L(0,0) = 1$, $L(i,i) = U(i-1,i-1)$ for all i, $1 \leq i < n$, and $D = s \cdot U(n-1,n-1)$ where s is the determinant of Q.
Precondition: M is quadratic.

bool verify_determinant(*integer_matrix* M, *integer* D, *integer_matrix*& L, *integer_matrix*& U, *array<int>* q, *integer_vector*& c)

verifies the conditions stated above.

integer determinant(*integer_matrix* M)

returns the determinant of M.
Precondition: M is quadratic.

int sign_of_determinant(*integer_matrix* M)

returns the sign of the determinant of M.
Precondition: M is quadratic.

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_matrix*& *spanning_vectors*, *integer_vector*& c)

determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system.
Precondition: M.dim1() = b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_vector*& c)

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$.
Precondition: M.dim1() = b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D)

as above, but without the witness c
Precondition: M.dim1() = b.dim().

bool solvable(*integer_matrix* M, *integer_vector* b)

determines whether the system $M \cdot x = b$ is solvable
Precondition: M.dim1() = b.dim().

bool homogeneous_linear_solver(*integer_matrix* M, *integer_vector*& x)

determines whether the homogeneous linear system $M \cdot x = 0$ has a non-trivial solution. If yes, then x is such a solution.

void independent_columns(*integer_matrix* M, *array<int>*& *columns*)

returns the indices of a maximal subset of independent columns of M. The index range of *columns* starts at 0.

int rank(*integer_matrix* M)

returns the rank of matrix M

ostream& *ostream*& $O \ll M$

writes matrix M row by row to the output stream O.

istream& *istream*& I $\gg$ *integer_matrix*& M

reads matrix M row by row from the input stream I.

4. Implementation

The datatype integer_matrix is implemented by two-dimensional arrays of integers. Operations *determinant*, *inverse*, *linear_solver*, and *rank* take time $O(n^3)$, *col* takes time $O(n)$, *row*, *dim1*, *dim2*, take constant time, and all other operations take time $O(nm)$. The space requirement is $O(nm)$.

All functions on integer matrices compute the exact result, i.e., there is no rounding error. The implemenation follows a proposal of J. Edmonds (J. Edmonds, Systems of distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241 -245). Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

2.1.2 Vectors with Integer Entries (integer_vector)

1. Definition

An instance of the data type *integer_vector* is a vector of variables of type integer.

2. Creation

integer_vector v; creates an instance v of type *integer_vector*; v is initialized to the zero-dimensional vector.

integer_vector v(*int* d); creates an instance v of type *integer_vector*; v is initialized to the zero vector of dimension d.

integer_vector v(*integer* a, *integer* b);
creates an instance v of type *integer_vector*; v is initialized to the two-dimensional vector (a, b).

integer_vector v(*integer* a, *integer* b, *integer* c);
creates an instance v of type *integer_vector*; v is initialized to the three-dimensional vector (a, b, c).

3. Operations

int v.dim( ) returns the dimension of v.

integer&	$v[int\ i]$	returns i-th component of v. *Precondition*: $0 \leq i \leq v.dim(\ ) - 1$. This check can be turned off by the flag `LEDA_CHECKING_OFF`.
integer_vector&	$v+= v1$	Addition plus assignment. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector&	$v-= v1$	Subtraction plus assignment. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector	$v + v1$	Addition. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector	$v - v1$	Subtraction. *Precondition*: $v.dim(\ ) = v1.dim(\ )$.
integer	$v * v1$	Inner Product. *Precondition*: $v.dim(\ ) = v1.dim(\ )$.
integer_vector	$integer\ r * v$	Componentwise multiplication with number r.
integer_vector	$v * integer\ r$	Componentwise multiplication with number r.
bool	$v \equiv w$	Test for equality.
bool	$v\ != w$	Test for inequality.
ostream&	$ostream\&\ O \ll v$	writes v componentwise to the output stream O.
istream&	$istream\&\ I \gg integer\_vector\&\ v$	reads v componentwise from the input stream I.

4. Implementation

Vectors are implemented by arrays of integers. All operations on a vector v take time $O(v.dim(\ ))$, except for *dim* and [ ] which take constant time. The space requirement is $O(v.dim(\ ))$.

2.2 The Geometric Objects

2.2.1 Rational Points (rat_point)

1. Definition

An instance of data type *rat_point* is a point with rational coordinates in an arbitrary dimensional space. A point $p = (p_0, \ldots, p_{d-1})$ in d-dimensional space is represented by homogeneous coordinates $(h_0, h_1, \ldots, h_d)$ of arbitrary length integers such that $p_i = h_i/h_d$. The homogenizing coordinate h_d is positive.

We call p_i, $0 \leq i < d$ the i-th cartesian coordinate and h_i, $0 \leq i \leq d$, the i-th homogeneous coordinate. We call d the dimension of the point.

The default ordering is the lexicographic ordering of the cartesian coordinate tuples.

rat_point is an item type.

2. Creation

rat_point p(*int* $d = 2$); introduces a variable p of type *rat_point* in d-dimensional space.

rat_point p(*integer* a, *integer* b, *integer* $D = 1$);
introduces a variable p of type *rat_point* initialized to the two-dimensional point with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_point p(*integer_vector* c, *integer* D);
introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_point p(*integer_vector* c); introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_point *rat_point*::d2(*integer a*, *integer b*, *integer D* = 1)

returns a *rat_point* of dimension 2 initialized to a point with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_point *rat_point*::d3(*integer a*, *integer b*, *integer c*, *integer D* = 1)

returns a *rat_point* of dimension 3 initialized to a point with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_point *rat_point*::origin(*int d* = 2)

returns the origin in d-dimensional space.

int *p*.dim( ) returns the dimension of p.

rational *p*.coord(*int i*) returns the i-th cartesian coordinate of p.

rational *p*[*int i*] returns the i-th cartesian coordinate of p.

integer *p*.hcoord(*int i*) returns the i-th homogeneous coordinate of p.

rat_point *p*.transform(*aff_transformation t*)

returns $t(p)$.

rat_vector *p*.to_rat_vector( ) converts p to an vector.

rat_vector to_rat_vector(*rat_point p*)

converts p to an vector.

rat_direction *p*.to_rat_direction( )

converts p to a direction.
Precondition: p is different form the origin.

rat_direction to_rat_direction(*rat_point p*)

converts p to a direction.
Precondition: p is different form the origin.

Additional Operations for points in two-dimensional space

rational *p*.xcoord( ) returns the zeroth cartesian coordinate of p.

rational *p*.ycoord( ) returns the first cartesian coordinate of p.

integer	p.X( )	returns the zeroth homogeneous coordinate of p.
integer	p.Y( )	returns the first homogeneous coordinate of p.
integer	p.W( )	returns the homogenizing coordinate of p.
rat_point	p.rotate90(*rat_point* q)	returns p rotated counterclockwise by 90 degrees about q.
rat_point	p.rotate90( )	returns p rotated counterclockwise by 90 degrees about the origin.

3.2 Tests

bool	p.is_origin( )	returns true if p is equal to the origin.
bool	identical(*rat_point* p, *rat_point* q)	test for identity
bool	$p \equiv q$	test for equality.
bool	p != q	test for inequality.

3.3 Arithmetical Operators

rat_vector	$p - q$	returns $p - q$. *Precondition*: $p.dim( ) \equiv q.dim( )$.
rat_point	p + *rat_vector* v	returns $p + v$. *Precondition*: $p.dim( ) \equiv v.dim( )$.
rat_point	p.translate(*rat_vector* v)	returns returns $p + v$. *Precondition*: $p.dim( ) \equiv v.dim( )$.
rat_point&	$p+=$ *rat_vector* v	adds v to p. *Precondition*: $p.dim( ) \equiv v.dim( )$.
rat_point	$p -$ *rat_vector* v	returns $p - v$. *Precondition*: $p.dim( ) \equiv v.dim( )$.
rat_point&	$p-=$ *rat_vector* v	subtracts v from p. *Precondition*: $p.dim( ) \equiv v.dim( )$.

3.4 Input and Output

ostream& *ostream& O* $\ll$ *p*

writes the homogeneous coordinates of p to output stream O.

istream& *istream& I* $\gg$ *rat_point& p*

reads the homogeneous coordinates of p from input stream I. This operator uses the current dimension of p.

3.5 Position Tests

int orientation(*array<rat_point>A*)

determines the orientation of the points in A, where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \dots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the i-th point in A.

int side_of_oriented_sphere(*array<rat_point>A*, *rat_point x*)

determines whether the point x lies inside ($=-1$), on ($=0$), or outside ($=+1$) the *oriented* sphere defined by the points in A, where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ \mathit{lift}(A[0]) & \mathit{lift}(A[1]) & \dots & \mathit{lift}(A[d]) & \mathit{lift}(x) \end{vmatrix}$$

where for a point p with cartesian coordinates p_i we use $\mathit{lift}(p)$ to denote the $d+1$-dimensional point with cartesian coordinate vector $(p_0, \dots, p_{d-1}, \sum_{0 \le i < d} p_i^2)$.

int side_of_sphere(*array<rat_point>A*, *rat_point x*)

determines whether the point x lies inside ($=-1$), on ($=0$), or outside ($=+1$) the sphere defined by the points in A, where A consists of $d+1$ points in d-space. (equivalent to $\mathit{orientation}(A) * \mathit{side\_of\_oriented\_sphere}(A, x)$)
Precondition: $\mathit{orientation}(A) \neq 0$

bool contained_in_simplex(*array<rat_point>A*, *rat_point x*)

determines whether x is contained in the simplex spanned by the points in A. A may consists of up to $d+1$ points.
Precondition: The points in A are affinely independent.

3.6 Affine Hull, Dependence and Rank

bool contained_in_affine_hull(*array<rat_point>A*, *rat_point x*)

determines whether x is contained in the affine hull of the points in A.

int affine_rank(*array<rat_point>A*)

computes the affine rank of the points in A.

bool affinely_independent(*array<rat_point>A*)

decides whether the points in A are affinely independent.

Additional Operations for points in two-dimensional space

rational area(*rat_point a*, *rat_point b*, *rat_point c*)

computes the signed area of the triangle determined by a,b,c, positive if $orientation(a, b, c) > 0$ and negative otherwise.

int orientation(*rat_point a*, *rat_point b*, *rat_point c*)

computes the orientation of points a, b, c, i.e., the sign of the determinant

$$\begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ c_w & c_x & c_y \end{vmatrix}$$

bool collinear(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and false otherwise.

bool right_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a right turn, i.e., $orientation(a, b, c) > 0$, and false otherwise.

bool left_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a left turn, i.e., $orientation(a, b, c) < 0$, and false otherwise.

int side_of_oriented_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $+1$ if point d lies left of the directed circle through points a, b, and c, 0 if a,b,c,and d are cocircular, and -1 otherwise. If a, b, and c are collinear the directed circle is a line oriented from a to b if c is not part of the connecting segment $\overline{ab}$, or else oriented from b to c.

int side_of_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $+1$ if point d lies inside of, 0 if on and -1 if outside of the circle through points a, b, and c,
Precondition: a, b, c are not collinear

bool cocircular(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if points a, b, c, d are cocircular, i.e., $sidc_of_orientcd_circle(a,b,c) = 0$, and false otherwise.

bool incircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if point d lies in the interior of the circle through the points a, b, and c, and false otherwise.

bool outcircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if point d lies outside the circle through the points a, b, and c, and false otherwise.

4. Implementation

Points are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, point-vector arithmetic, input and output on a point p take time $O(p.dim(\ ))$. $dim(\ )$, coordinate access and conversions take constant time. The operations for affine calculation and determinant evaluation have the cubic costs of the used matrix operations. The space requirement is $O(p.dim(\ ))$.

2.2.2 Rational Vectors (rat_vector)

1. Definition

An instance of data type *rat_vector* is a vector of rational numbers. A d-dimensional vector $r = (r_0, \ldots, r_{d-1})$ is represented in homogeneous coordinates $(h_0, \ldots, h_d)$, where $r_i = h_i/h_d$ and the h_i's are of type *integer*. We call the r_i's the cartesian coordinates of the vector. The homogenizing coordinate h_d is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type *rat_point*). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

rat_vector is an item type.

2. Creation

rat_vector v(*int d* = 2); introduces a variable v of type *rat_vector* initialized to the zero vector of dimension d.

rat_vector v(*integer a*, *integer b*, *integer D* = 1);

introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector *v*(*integer_vector* *c*, *integer* *D*):

introduces a variable v of type *rat_vector* initialized to the vector with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_vector *v*(*integer_vector* *c*); introduces a variable v of type *rat_vector* initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_vector *rat_vector*::d2(*integer* *a*, *integer* *b*, *integer* *D* = 1)

returns a *rat_vector* of dimension 2 initialized to a vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector *rat_vector*::d3(*integer* *a*, *integer* *b*, *integer* *c*, *integer* *D* = 1)

returns a *rat_vector* of dimension 3 initialized to a vector with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector *rat_vector*::unit(*int* *i*, *int* *d* = 2)

returns a *rat_vector* of dimension d initialized to the i-th unit vector.
Precondition: $0 \leq i < d$.

rat_vector *rat_vector*::zero(*int* *d* = 2)

returns the zero vector in d-dimensional space.

int *v*.dim( ) returns the dimension of v.

rational *v*.coord(*int* *i*) returns the i-th cartesian coordinate of v.

rational *v*[*int* *i*] returns the i-th cartesian coordinate of v.

integer *v*.hcoord(*int* *i*) returns the i-th homogeneous coordinate of v.

rat_direction	v.to_rat_direction( )	converts to a direction.
rat_direction	to_rat_direction(*rat_vector* v)	converts to a direction.
rat_point	v.to_rat_point( )	converts to a point.
rat_point	to_rat_point(*rat_vector* v)	converts to a point.
rat_vector	v.transform(*aff_transformation* t)	returns $t(v)$.

Additional Operations for vectors in two-dimensional space

rational	v.xcoord( )	returns the zeroth cartesian coordinate of v.
rational	v.ycoord( )	returns the first cartesian coordinate of v.
integer	v.X( )	returns the zeroth homogeneous coordinate of v.
integer	v.Y( )	returns the first homogeneous coordinate of v.
integer	v.W( )	returns the homogenizing coordinate of v.

3.2 Tests

bool	identical(*rat_vector* v, *rat_vector* w)	Test for identity.
bool	$v \equiv w$	Test for equality.
bool	v != w	Test for inequality.

3.3 Arithmetical Operators

rat_vector	*integer* $n * v$	multiplies all cartesian coordinates by n.
rat_vector	*rational* $r * v$	multiplies all cartesian coordinates by r.
rat_vector&	$v *=$ *integer* n	multiplies all cartesian coordinates by n.
rat_vector&	$v *=$ *rational* r	multiplies all cartesian coordinates by r.
rat_vector	v / *integer* n	divides all cartesian coordinates by n.
rat_vector	v / *rational* r	divides all cartesian coordinates by n.

rat_vector&	v /= *integer* n	divides all cartesian coordinates by n.
rat_vector&	v /= *rational* r	divides all cartesian coordinates by r.
rational	$v * w$	scalar product, i.e., $\sum_{0 \le i < d} v_i w_i$, where v_i and w_i are the cartesian coordinates of v and w respectively.
rat_vector	$v + w$	adds cartesian coordinates.
rat_vector&	v+= w	addition plus assignment.
rat_vector	$v - w$	subtracts cartesian coordinates
rat_vector&	v−= w	subtraction plus assignment.
rat_vector	$-v$	returns -v.

3.4 Input and Output

ostream&	*ostream&* $O \ll v$	writes v's homogeneous coordinates componentwise to the output stream O.
istream&	*istream&* $I \gg$ *rat_vector&* v	reads v's homogeneous coordinates componentwise from the input stream I. The operator uses the current dimension of v.

3.5 Linear Hull, Dependence and Rank

bool	contained_in_linear_hull(*array<rat_vector>* A, *rat_vector* x)	determines whether x is contained in the linear hull of the vectors in A.
int	linear_rank(*array<rat_vector>* A)	computes the linear rank of the vectors in A.
bool	linearly_independent(*array<rat_vector>* A)	decides whether the vectors in A are linearly independent.
array<rat_vector>	linear_base(*array<rat_vector>* A)	computes a basis of the linear space spanned by the vectors in A.

4. Implementation

Vectors are implemented by arrays of integers as an item type. All operations like cre-

ation, initialization, tests, vector arithmetic, input and output on an vector v take time $O(v.dim(\ ))$. $dim(\ )$, coordinate access and conversions take constant time. The operations for linear hull, rank and independence have the cubic costs of the used matrix operations. The space requirement is $O(v.dim(\ ))$.

2.2.3 Rational Directions (rat_direction)

1. Definition

A *rat_direction* is any non-zero vector. We represent directions in d-dimensional space as a tuple $(h_0, \ldots, h_d)$ of integers which we call the homogeneous coordinates of the direction. The coordinate h_d must be positive. The cartesian coordinates of a direction are $c_i = h_i/h_d$ for $0 \leq i < d$. Two directions are equal if their cartesian coordinates are positive multiples of each other. Directions are in one-to-one correspondence to points on the unit sphere.

In two-dimensional space directions are also in one-to-one correspondance to angles. More precisely, a direction $dir = (h_0, h_1, h_2)$ corresponds to the angle α with $\sin\alpha = c_0/L$ and $\cos\alpha = c_1/L$ and $L = \sqrt{(h_0^2 + h_1^2)/h_2^2}$ the length of dir. For a direction dir we use $angle(dir)$ to denote this angle.

rat_direction is an item type.

2. Creation

rat_direction $dir(int\ d\ =\ 2)$; introduces a variable *dir* of type *rat_direction* initialized to some direction in d-dimensional space.

rat_direction $dir(integer\ a, integer\ b, integer\ D\ =\ 1)$;
introduces a variable *dir* of type *rat_direction* initialized to the two-dimensional direction with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction $dir(integer\_vector\ c, integer\ D)$;
introduces a variable *dir* of type *rat_direction* initialized to the two-dimensional direction with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_direction *dir*(*integer_vector* *c*):

introduces a variable *dir* of type *rat_direction* initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_direction *rat_direction*::d2(*integer* *a*, *integer* *b*, *integer* *D* = 1)

returns a *rat_direction* of dimension 2 initialized to a direction with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction *rat_direction*::d3(*integer* *a*, *integer* *b*, *integer* *c*, *integer* *D* = 1)

returns a *rat_direction* of dimension 3 initialized to a direction with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction *rat_direction*::unit(*int* *i*, *int* *d* = 2)

returns a *rat_direction* of dimension d initialized to the i-th unit direction.
Precondition: $0 \leq i < d$.

int *dir*.dim( ) returns the dimension of *dir*.

rational *dir*.coord(*int* *i*) returns the i-th cartesian coordinate of *dir*.

rational *dir*[*int* *i*] returns the i-th cartesian coordinate of *dir*.

integer *dir*.hcoord(*int* *i*) returns the i-th homogeneous coordinate of *dir*.

rat_direction *dir*.transform(*aff_transformation* *t*)

returns $t(p)$.

rat_direction *dir*.opposite( ) returns the direction opposite to *dir*.

rat_direction $-dir$ returns *dir*.opposite( ).

rat_vector *dir*.to_rat_vector( ) returns a vector pointing in direction *dir*.

rat_vector *dir*.to_rat_vector(*rat_direction d*)

returns a vector pointing in direction *dir*.

Additional Operations for directions in two-dimensional space

integer *dir*.X( ) returns the zeroth homogeneous coordinate of *dir*.

integer *dir*.Y( ) returns the first homogeneous coordinate of *dir*.

3.2 Tests

bool identical(*rat_direction v*, *rat_direction w*)

Test for identity.

bool $v \equiv w$ Test for equality.

bool v != w Test for inequality.

3.3 Input and Output

ostream& *ostream&* $O \ll d$

writes the homogeneous coordinates of d to output stream O.

istream& *istream&* $I \gg$ *rat_direction&* d

reads the homogeneous coordinates of d from input stream I. This operator uses the current dimension of d.

4. Implementation

Directions are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, inversion, input and output on a direction d take time $O(d.dim(\,))$. $dim(\,)$, coordinate access and conversion take constant time. The space requirement is $O(d.dim(\,))$.

2.2.4 Rational Hyperplanes (rat_hyperplane)

1. Definition

An instance of data type *rat_hyperplane* is a hyperplane with rational coefficients in an arbitrary dimensional space. A hyperplane h is represented by coefficients $(c_0, c_1, \ldots, c_d)$ of arbitrary length integers. At least one of c_0 to c_{d-1} must be non-zero. The plane equation is $\sum_{0 \le i \le d} c_i x_i = 0$, where x_0 to x_d are homogeneous point coordinates. The sign

of the left hand side of this expression determines the position of a point x with respect to the hyperplane (on the hyperplane, on the negative side, or on the positive side).

There are two equality predicates for hyperplanes. The (weak) equality predicate (*operator* $\equiv$) declares two hyperplanes equal if they consist of the same set of points, the strong equality predicate (*strong_eq*) requires in addition that the negative halfspaces agree. In other words, two hyperplanes are strongly equal if their coefficient vectors are positive multiples of each other and they are (weakly) equal if their coefficient vectors are multiples of each other. Corresponding to the two equality predicates we have two linear orders: *compare* corresponds to weak equality and *strong_compare* corresponds to strong equality.

rat_hyperplane is an item type.

2. Creation

rat_hyperplane *h*(*int d* = 2);

introduces a variable h of type *rat_hyperplane* initialized to some hyperplane in d-dimensional space.

rat_hyperplane *h*(*integer_vector c*);

introduces a variable h of type *rat_hyperplane* initialized to the hyperplane with coefficients c.

rat_hyperplane *h*(*integer_vector c*, *integer D*);

introduces a variable h of type *rat_hyperplane* initialized to the hyperplane with coefficients (c, D).

rat_hyperplane *h*(*array<rat_point>P*, *rat_point o*, *int k* = 0);

constructs some hyperplane that passes through the points in P. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane.
Precondition: There must be a hyperplane passing through the points in P and if $k \neq 0$ then o must not lie on the constructed hyperplane

rat_hyperplane *h*(*rat_point p*, *rat_direction dir*, *rat_point o*, *int side* = 0);

constructs some hyperplane with normal direction *dir* that passes through p. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane.
Precondition: If $k \neq 0$ then o must not lie on the constructed hyperplane

3. Operations

3.1 Initialization, Access and Evaluation

rat_hyperplane *rat_hyperplane*::d2(*rat_point p1*, *rat_point p2*, *rat_point o* = *rat_point*::*origin*( ), *int k* = 0)

returns a *rat_hyperplane* of dimension 2 that passes through the points *p1* and *p2*. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane. *Precondition*: If $k \neq 0$ then o must not lie on the constructed hyperplane

rat_hyperplane *rat_hyperplane*::d3(*rat_point p1*, *rat_point p2*, *rat_point p3*, *rat_point o* = *rat_point*::*origin*(3), *int k* = 0)

returns a *rat_hyperplane* of dimension 3 that passes through the points *p1*, *p2*, *p3*. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane. *Precondition*: If $k \neq 0$ then o must not lie on the constructed hyperplane

int h.dim( ) returns the dimension of h.

integer h[*int i*] returns the i-th coefficient of h.

integer_vector h.coefficent_vector( ) returns the coefficient vector $(c_0, \ldots, c_d)$ of h.

rat_vector h.normal_vector( ) returns the normal vector of h. It points from the negative halfspace into the positive halfspace and its homogeneous coordinates are $(c_0, \ldots, c_{d-1}, 1)$.

rat_direction h.normal_direction( ) returns the normal direction of h. It points from the negative halfspace into the positive halfspace.

integer h.value_at(*rat_point p*)

returns the value of h at the point p, i.e., $\sum_{0 \leq i \leq d} h_i p_i$.
Warning: this value depends on the particular representation of h and p.

int h.which_side(*rat_point p*)

returns the side of the hyperplane h containing p.

bool h.contains(*rat_point p*)

return if the point p lies on the hyperplane h.

rat_hyperplane h.transform(*aff_transformation t*)

returns $t(h)$.

3.2 Tests

int	strong_compare(*rat_hyperplane h1*, *rat_hyperplane h2*)	strong compare.
bool	identical(*rat_hyperplane h1*, *rat_hyperplane h2*)	test for identity.
bool	*h1* $\equiv$ *h2*	test for equality.
bool	*h1* != *h2*	test for inequality.
bool	strong_eq(*rat_hyperplane h1*, *rat_hyperplane h2*)	test for strong equality.

3.3 Input and Output

ostream&	*ostream& O* $\ll$ *h*	writes the coefficients of h to output stream O.
istream&	*istream& I* $\gg$ *rat_hyperplane& h*	reads the coefficients of h from input stream I. This operator uses the current dimension of h.

4. Implementation

Hyperplanes are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on a hyperplane h take time $O(h.dim(\ ))$. $dim(\ )$ and coordinate access take constant time. The space requirement is $O(h.dim(\ ))$.

2.2.5 Rational Segments (rat_segment)

1. Definition

An instance s of the data type *rat_segment* is a directed straight line segment connecting two rational points p and q. p is called the start or source point and q is called the target point of s, both points are called endpoints of s. A segment whose endpoints are equal is called *trivial*.

2. Creation

rat_segment *s*(*int d* = 2);

introduces a variable s of type *rat_segment* and initializes it to some segment in d-dimensional space.

rat_segment *s*(*rat_point p*, *rat_point q*);

introduces a variable s of type *rat_segment*. s is initialized to the segment (p, q).

rat_segment *s*(*integer x1*, *integer y1*, *integer x2*, *integer y2*);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1), (x2, y2)]$ in two-dimensional space.

3. Operations

3.1 Initialization, Access and Conversions

rat_segment *rat_segment*::d2(*integer x1*, *integer y1*, *integer D1*, *integer x2*, *integer y2*, *integer D2*)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_segment *rat_segment*::d2(*integer x1*, *integer y1*, *integer x2*, *integer y2*)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_segment *rat_segment*::d3(*integer x1*, *integer y1*, *integer z1*, *integer x2*, *integer y2*, *integer z2*)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_segment *rat_segment*::d3(*integer x1*, *integer y1*, *integer z1*, *integer D1*, *integer x2*, *integer y2*, *integer z2*, *integer D2*)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int	*s*.dim( )	returns the dimension of the underlying space.
rat_point	*s*.source( )	returns the source point of segment s.
rat_point	*s*.point1( )	returns the source point of segment s.
rat_point	*s*.target( )	returns the target point of segment s.
rat_point	*s*.point2( )	returns the target point of segment s.
rational	*s*.coord1(*int i*)	returns the i-th cartesian coordinate of the source of s.

rational	*s*.coord2(*int i*)	returns the i-th cartesian coordinate of the target of s.
integer	*s*.hcoord1(*int i*)	returns the i-th homogeneous coordinate of the source of s.
integer	*s*.hcoord2(*int i*)	returns the i-th homogeneous coordinate of the target of s.
rat_segment	*s*.reverse( )	returns the segment (*target*( ), *source*( )).
rat_direction	*s*.direction( )	returns the direction of s. *Precondition*: s is non-trivial.
rat_vector	*s*.source_target_vector( )	returns the vector from source to target. *Precondition*: s is non-trivial.
rat_line	*s*.supporting_line( )	returns the supporting line of s. *Precondition*: s is non-trivial.
rat_segment	*s*.transform(*aff_transformation t*)	returns $t(s)$.

3.2 Tests and Calculations

bool	*s*.is_trivial( )	returns true if s is trivial.
bool	identical(*rat_segment s1*, *rat_segment s2*)	Test for identity.
bool	strong_eq(*rat_segment s1*, *rat_segment s2*)	Test for equality as oriented segments.
bool	$s \equiv t$	Test for equality as unoriented segments.
bool	s != t	Test for inequality.
bool	parallel(*rat_segment s1*, *rat_segment s2*)	Test if the supporting lines are parallel. *Precondition*: : *s1* and *s2* are not trivial.
bool	*s*.contains(*rat_point p*)	returns true if p lies on s and false otherwise.
bool	common_endpoint(*rat_segment s1*, *rat_segment s2*, *rat_point*& *common*)	if *s1* and *s2* touch in a common end point, this point is assigned to common and the result is true, otherwise the result is false.

int s.intersection(*rat_line t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ by the following means. The return value is one of the constants {*NO_I*, *PNT_I*, *SEG_I*, *RAY_I*, *LIN_I*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int s.intersection(*rat_ray t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ as above.

int s.intersection(*rat_segment t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ as above.

3.3 Input and Output

ostream& *ostream& O* $\ll$ *s*

writes the homogeneous coordinates of s to output stream O (in order source-target).

istream& *istream& I* $\gg$ *rat_segment& s*

reads the homogeneous coordinates of s from input stream I (in order source-target).

Additional Operations for segments in two-dimensional space

rational s.xcoord1( ) returns the x-coordinate of the source point of segment s.

rational s.xcoord2( ) returns the x-coordinate of the target point of segment s.

rational s.ycoord1( ) returns the y-coordinate of the source point of segment s.

rational s.ycoord2( ) returns the y-coordinate of the target point of segment s.

integer s.X1( ) returns the zeroth homogeneous coordinate of the start point of segment s.

integer s.X2( ) returns the zeroth homogeneous coordinate of the end point of segment s.

integer	*s*.Y1( )	returns the first homogeneous coordinate of the start point of segment s.
integer	*s*.Y2( )	returns the first homogeneous coordinate of the end point of segment s.
integer	*s*.W1( )	returns the homogenizing coordinate of the start point of segment s.
integer	*s*.W2( )	returns the homogenizing coordinate of the end point of segment s.
integer	*s*.dx( )	returns the normalized x-difference $X1 \cdot W2 - X2 \cdot W1$ of the segment.
integer	*s*.dy( )	returns the normalized y-difference $Y1 \cdot W2 - Y2 \cdot W1$ of the segment.
bool	*s*.vertical( )	returns true if s is vertical (or trivial) and false otherwise.
bool	*s*.horizontal( )	returns true if s is horizontal (or trivial) and false otherwise.
bool	*s*.intersection(*rat_segment t*, *rat_point*& *p*)	if s and t intersect in a single point the point of intersection is assigned to p and the result is true, otherwise the result is false. *Precondition*: The supporting lines are not parallel.
bool	*s*.intersection_of_lines(*rat_segment t*, *rat_point*& *p*)	if the lines supporting s and t are not parallel their point of intersection is assigned to p and the result is true, otherwise the result is false.
int	orientation(*rat_segment s*, *rat_point p*)	computes orientation(a, b, p), where a and b are the source and target of s respectively.
int	cmp_slopes(*rat_segment s1*, *rat_segment s2*)	returns *compare*(*slope*(*s1*), *slope*(*s2*)).

int cmp_at_line_defined_by(*rat_segment s1*, *rat_segment s2*, *rat_point r*)

Let L be the directed curve consisting of a vertical upward ray ending in $(r.xcoord(\,) + \epsilon^2, r.ycoord(\,) + \epsilon)$ followed by a horizontal segment ending in $(r.xcoord(\,) - \epsilon^2, r.ycoord(\,) + \epsilon)$,
followed by an upward vertical ray; here ϵ is a positive infinitesimal. If both segments are non-trivial then the result is the order of the two intersections along the line L. If at least one of the segments is trivial then the result is zero if one of the segments is contained in the other segment. If exactly one of the segments is trivial then the result is the position (above, on, or below) of this segment with respect to the other segment.

Precondition: For $i = 1, 2$ we have: if si is trivial then both endpoints are equal to r and if si is non-trivial then its smaller endpoint is less than or equal to r and its larger endpoint is larger than r.

bool intersection(*rat_segment s1*, *rat_segment s2*)

decides whether $s1$ and $s2$ intersect in one point when the supporting lines are not equal.

4. Implementation

Segments are implemented by a pair of points as an item type. All operations like creation, initialization, tests, the calculation of the direction and source-target vector, input and output on a segment s take time $O(s.dim(\,))$. $dim(\,)$, coordinate and end point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dim(\,))$. The space requirement is $O(s.dim(\,))$.

2.2.6 Rational Rays (rat_ray)

1. Definition

An instance of data type *rat_ray* is a ray in d-dimensional Euclidian space. *rat_ray* is an item type.

2. Creation

rat_ray r(*int d* = 2);

introduces a ray in d-dimensional space

rat_ray r(*rat_point p*, *rat_point q*);

introduces a ray through p and q and starting at p.
Precondition: p and q are distinct and have the same dimension.

rat_ray *r*(*rat_point p*, *rat_direction dir*);

introduces a ray starting in p with direction *dir*.
Precondition: p and *dir* have the same dimension.

rat_ray *r*(*rat_segment s*):

introduces a ray through *s.source*( ) and *s.target*( ) and starting at *s.source*( ).
Precondition: s is not trivial.

3. Operations

3.1 Initialization, Access and Conversions

rat_ray *rat_ray*::d2(*integer x1*, *integer y1*, *integer D1*, *integer x2*, *integer y2*, *integer D2*)

introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_ray *rat_ray*::d2(*integer x1*, *integer y1*, *integer x2*, *integer y2*)

introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_ray *rat_ray*::d3(*integer x1*, *integer y1*, *integer z1*, *integer x2*, *integer y2*, *integer z2*)

introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_ray *rat_ray*::d3(*integer x1*, *integer y1*, *integer z1*, *integer D1*, *integer x2*, *integer y2*, *integer z2*, *integer D2*)

introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int r.dim( ) returns the dimension of the underlying space.

rat_point r.source( ) returns the source point of r.

rat_point r.point1( ) returns the source point of r.

rat_point r.point2( ) returns a point on r distinct from r.*source*( ).

rat_direction r.direction( ) returns the direction of r.

rat_line r.supporting_line( )

returns the supporting line of r.

rat_ray r.transform(*aff_transformation t*)

returns $t(l)$.

3.2 Tests and Calculations

bool identical(*rat_ray r1*, *rat_ray r2*)

test for identity.

bool *r1* $\equiv$ *r2* test for equality

bool *r1* != *r2* test for inequality.

int parallel(*rat_ray r1*, *rat_ray r2*)

returns true if *r1* and *r2* are parallel and false otherwise.

bool r.contains(*rat_point p*)

returns true if p lies on r.

bool r.contains(*rat_segment s*)

returns true if s is part of r and false otherwise.

3.3 Intersection Calculations

bool r.intersection(*rat_hyperplane h*, *rat_point& p*)

returns true if h and r intersect in a single point and false otherwise. In the first case the point of intersection is assigned to p.

rat_point r.intersection(*rat_hyperplane h*)

returns the intersection of the hyperplane h with the ray r

Precondition: h and r intersect in a single point.

int r.intersection(*rat_line t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $r \cap t$ by the following means. The return value is one of the constants {*NO_I*, *PNT_I*, *SEG_I*, *RAY_I*, *LIN_I*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int r.intersection(*rat_ray t*, *rat_point& i1*, *rat_point& i2*)
returns the intersection set $r \cap t$ as above.

int r.intersection(*rat_segment t*, *rat_point& i1*, *rat_point& i2*)
returns the intersection set $r \cap t$ as above.

3.4 Input and Output

ostream& *ostream& O* $\ll$ *r*
writes the coefficients of r to output stream O.

istream& *istream& I* $\gg$ *rat_ray& r*
reads the coefficients of r from input stream I. This operator uses the current dimension of r.

Additional Operations for rays in two-dimensional space

bool r.vertical( ) returns true if r is vertical or trivial and false otherwise.

bool r.horizontal( ) returns true if r is horizontal or trivial and false otherwise.

bool r.intersection(*rat_ray l1*, *rat_point& p*)
if r and l_1 are not parallel the point of intersection is assigned to p and the result is true, otherwise the result is false.

int orientation(*rat_ray r*, *rat_point p*)
computes orientation(a, b, p), where a and b are *source* and *another_point* of r respectively.

4. Implementation

Rays are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a ray r take time $O(r.dim(\ ))$. $dim(\ )$, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dim(\ ))$. The space requirement is $O(v.dim(\ ))$.

2.2.7 Rational Lines (rat_line)

1. Definition

An instance of data type *rat_line* is an oriented line in d-dimensional Euclidian space. *rat_line* is an item type.

2. Creation

rat_line l(*int* $d = 2$);

introduces a variable l of type *rat_line* and initializes it to some line in d-dimensional space

rat_line l(*rat_point* p, *rat_point* q);

introduces a line through p and q and oriented from p to q.
Precondition: p and q are distinct and have the same dimension.

rat_line l(*rat_point* p, *rat_direction* *dir*);

introduces a line through p with direction *dir*.
Precondition: p and *dir* have the same dimension.

rat_line l(*rat_segment* s);

introduces a variable l of type *rat_line* and initializes it to the line through s.*source*( ) and s.*target*( ).
Precondition: s is not trivial.

3. Operations

3.1 Initialization, Access and Conversions

rat_line *rat_line*::d2(*integer x1*, *integer y1*, *integer D1*, *integer x2*, *integer y2*, *integer D2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_line *rat_line*::d2(*integer x1*, *integer y1*, *integer x2*, *integer y2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_line *rat_line*::d3(*integer x1*, *integer y1*, *integer z1*, *integer x2*, *integer y2*, *integer z2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_line *rat_line*::d3(*integer x1*, *integer y1*, *integer z1*, *integer D1*, *integer x2*, *integer y2*, *integer z2*, *integer D2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int l.dim( ) returns the dimension of the underlying space.

rat_point	*l*.point1( )	returns a point on l.
rat_point	*l*.point2( )	returns a point on l distinct from *l.point1*( ). The line is directed from *point1* to *point2*.
void	*l*.two_points(*rat_point*& *p1*, *rat_point*& *p2*)	after the call *p1* and *p2* are two different points on l. The line is directed from *p1* to *p2*.
rat_direction	*l*.direction( )	returns the direction of l.
rat_line	*l*.transform(*aff_transformation t*)	returns $t(l)$.

3.2 Tests

bool	identical(*rat_line l1*, *rat_line l2*)	test for identity.
bool	*l1* $\equiv$ *l2*	equality as unoriented lines.
bool	*l1* != *l2*	inequality as unoriented lines.
bool	strong_eq(*rat_line l1*, *rat_line l2*)	equality as oriented lines.
bool	*l*.contains(*rat_point p*)	returns true if p lies on l and false otherwise.
bool	*l*.contains(*rat_segment s*)	returns true if s is part of l and false otherwise.
int	parallel(*rat_line l1*, *rat_line l2*)	returns true if *l1* and *l2* are parallel and false otherwise.

3.3 Intersection Calculations

bool	*l*.intersection(*rat_hyperplane h*, *rat_point*& *p*)	returns true if h and l intersect in a single point and false otherwise. In the first case the point of intersection is assigned to p.
rat_point	*l*.intersection(*rat_hyperplane h*)	returns the intersection of hyperplane h with line l *Precondition:* h and l intersect in a single point.

int *l*.intersection(*rat_line t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ by the following means. The return value is one of the constants {*NO_I*, *PNT_I*, *SEG_I*, *RAY_I*, *LIN_I*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int *l*.intersection(*rat_ray t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ as above.

int *l*.intersection(*rat_segment t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ as above.

3.4 Input and Output

ostream& *ostream*& *O* $\ll$ *l* writes the coefficients of *l* to output stream *O*.

istream& *istream*& *I* $\gg$ *rat_line*& *l*

reads the coefficients of *l* from input stream *I*. This operator uses the current dimension of *l*.

Additional Operations for segments in two-dimensional space

bool *l*.vertical( ) returns true if *l* is vertical and false otherwise.

bool *l*.horizontal( ) returns true if *l* is horizontal and false otherwise.

bool *l*.intersection(*rat_line l1*, *rat_point*& *p*)

if *l* and l_1 are not parallel the point of intersection is assigned to *p* and the result is true, otherwise the result is false.

rat_line *l*.perpendicular(*rat_point p*)

computes the perpendicular line of *l* through *p*.

int orientation(*rat_line l*, *rat_point p*)

computes orientation(*a*, *b*, *p*), where *a* and *b* are point1 and point2 of *l* respectively.

int cmp_slopes(*rat_line l1*, *rat_line l2*)

returns *compare*(*slope*(*l1*), *slope*(*l2*)).

4. Implementation

Lines are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a line l take time $O(l.dim(\,))$. $dim(\,)$, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(l.dim(\,))$. The space requirement is $O(l.dim(\,))$.

2.2.8 Affine Transformations (aff_transformation)

1. Definition

An instance of the data type *aff_transformation* is an affine transformation of d-dimensional space. It is specified by a square integer matrix M of dimension $d+1$. All entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero. A point p with homogeneous coordinates $(p[0], \ldots, p[d])$ is transformed into the point $M * p$.

aff_transformation is an item type.

2. Creation

aff_transformation $t(int\ d\ =\ 2)$;
introduces the identity transformation in d-dimensional space.

aff_transformation $t(integer\_matrix\ M)$;
the transformation of d-space specified by matrix M.
Precondition: M is a square matrix of dimension $d+1$.

aff_transformation $t(rat\_vector\ v)$;
translation by vector v.

aff_transformation $t(integer\_vector\ v)$;
the transformation of d-space specified by a diagonal matrix with vector v on the diagonal (a scaling of the space).
Precondition: v is a vector of dimension $d+1$.

3. Operations

aff_transformation *aff_transformation*::d2_scale(*integer num, integer den*)
returns a scaling by a scale factor *num/den*.

aff_transformation *aff_transformation*::d2_transl(*rat_vector vec*)
returns a translation by a vector *vec*.

aff_transformation *aff_transformation*::d2_rot(*integer sin_num*, *integer cos_num*, *integer den*)

returns a rotation with sine and cosine values sin_num/den and cos_num/den.
Precondition: $sin\_num^2 + cos\_num^2 = den^2$.

aff_transformation *aff_transformation*::d2_rot_approx(*rat_direction dir*, *integer num*, *integer den*)

returns a rotation of 2-space. Approximates the rotation given by direction *dir*, such that the difference between the sindes and soines of the rotation given by *dir* and the approximation rotation are at most num/den each.

aff_transformation *aff_transformation*::d2_trafo(*integer m11*, *integer m12*, *integer m13*, *integer m21*, *integer m22*, *integer m23*, *integer m33*)

returns a general affine transformation in the 3×3 matrix form. The sub matrix $((m_{11}, m_{21})^t, (m_{21}, m_{22})^t)$ contains the scaling and rotation information, the vector $(m_{13}, m_{23})^t$ contains the translational part of the transformation.

aff_transformation *aff_transformation*::d2_trafo(*integer m11*, *integer m12*, *integer m21*, *integer m22*, *integer m33*)

returns a general linear transformation in the 2×2 matrix form. The sub matrix $((m_{11}, m_{21})^t, (m_{21}, m_{22})^t)$ contains the scaling and rotation information. There's no translational part.

int	*t*.dim( )	the dimension of the underlying space
integer_matrix	*t*.matrix( )	returns the transformation matrix
aff_transformation	*t*.inverse( )	returns inverse transfomation
aff_transformation	*t1* * *t2*	composition of transformations
ostream&	*ostream& O* $\ll$ *a*	writes the transformation matrix of a to output stream O.
istream&	*istream& I* $\gg$ *aff_transformation& a*	reads the coordinates of the transformation matrix from input stream I. This operator uses the current dimension of a.

4. Implementation

Affine Transformations are implemented by matrices of integers as an item type. All operations like creation, initialization, input and output on a transformation t take time $O(t.dim(\ )^2)$. $dim(\ )$ takes constant time. The operations for inversion and composition have the cubic costs of the used matrix operations. The space requirement is $O(t.dim(\ )^2)$.

3 The CWEB Projects

References

[FGK+96] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. *First ACM Workshop on Applied Computational Geometry*, 1996.

[MN89] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Technical Report TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989.

[MN95] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MZ94] M. Müller and J. Ziegler. An implementation of a convex hull algorithm. Technical Report MPI-I-94-105, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[NU95] S. Näher and C. Uhrig. *The LEDA User Manual*, 1995. Version 3.3.

[Ove96] M. H. Overmars. Designing the computational geometry algorithms library CGAL. *In Proceedings First ACM Workshop on Applied Computational Geometry*, 1996.

Matrices with Integer Entries (class integer_matrix)

Geokernel

May 21, 1996

Abstract

An instance of data type *integer_matrix* is a matrix of integer variables. The types integer_matrix and integer_vector together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

Contents

1. The Manual Page of class integer_matrix

1. Definition

An instance of data type *integer_matrix* is a matrix of integer variables. The types integer_matrix and integer_vector together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

2. Creation

integer_matrix M(*int* $n = 0$, *int* $m = 0$);

creates an instance M of type integer_matrix, M is initialized to the $n \times m$ - zero matrix.

integer_matrix M(*array*<*integer_vector*>A);

creates an instance M of type integer_matrix. Let A be an array of m column-vectors of common dimension n. M is initialized to an $n \times m$ matrix with the columns as specified by A.

3. Operations

int M.dim1( ) returns n, the number of rows of M.

int M.dim2( ) returns m, the number of columns of M.

integer_vector& M.row(*int* i)

returns the i-th row of M (an m-vector).
Precondition: $0 \leq i \leq n-1$.

integer_vector M.col(*int* i) returns the i-th column of M (an n-vector).
Precondition: $0 \leq i \leq m-1$.

integer& M(*int* i, *int* j)

returns $M_{i,j}$.
Precondition: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.

integer_matrix $M + M1$ Addition.
Precondition: M.dim1() = $M1$.dim1() and M.dim2() = $M1$.dim2().

integer_matrix $M - M1$ Subtraction.
Precondition: M.dim1() = $M1$.dim1() and M.dim2() = $M1$.dim2().

integer_matrix $M * M1$ Multiplication.
Precondition: M.dim2() = $M1$.dim1().

integer_vector $M *$ *integer_vector* *vec*

Multiplication with vector.
Precondition: M.dim2() = *vec*.dim().

integer_matrix $M *$ *integer* x Multiplication of every entry with integer x.

integer_matrix *integer* $x * M$ Multiplication of every entry with integer x.

integer_matrix identity(*int* n)

returns an n by n identity matrix.

integer_matrix transpose(*integer_matrix* M)

returns M^T ($m \times n$ - matrix).

integer_matrix inverse(*integer_matrix* M, *integer*& D)

returns the inverse matrix of M. More precisely, $1/D$ times the matrix returned is the inverse of M.
Precondition: determinant(M) $\neq 0$.

bool inverse(*integer_matrix* M, *integer_matrix*& *inverse*, *integer*& D, *integer_vector*& c)

determines whether M has an inverse. It also computes either the inverse as $(1/D) \cdot$ *inverse* or a vector c such that $c^T \cdot M = 0$.

integer determinant(*integer_matrix* M, *integer_matrix*& L, *integer_matrix*& U, *array<int>*& q, *integer_vector*& c)

returns the determinant D of M and sufficient information to verify that the value of the determinant is correct. If the determinant is zero then c is a vector such that $c^T \cdot M = 0$. If the determinant is non-zero then L and U are lower and upper diagonal matrices respectively and q encodes a permutation matrix Q with $Q(i,j) = 1$ iff $i = q(j)$ such that $L \cdot M \cdot Q = U$, $L(0,0) = 1$, $L(i,i) = U(i-1, i-1)$ for all i, $1 \leq i < n$, and $D = s \cdot U(n-1, n-1)$ where s is the determinant of Q.
Precondition: M is quadratic.

bool verify_determinant(*integer_matrix* M, *integer* D, *integer_matrix*& L, *integer_matrix*& U, *array<int>*q, *integer_vector*& c)

verifies the conditions stated above.

integer determinant(*integer_matrix* M)

returns the determinant of M.
Precondition: M is quadratic.

int sign_of_determinant(*integer_matrix* M)

returns the sign of the determinant of M.
Precondition: M is quadratic.

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_matrix*& *spanning_vectors*, *integer_vector*& c)

determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system.
Precondition: M.dim1() = b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D, *integer_vector*& c)

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$.
Precondition: M.dim1() = b.dim().

bool linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector*& x, *integer*& D)

as above, but without the witness c
Precondition: M.dim1() = b.dim().

bool solvable(*integer_matrix* M, *integer_vector* b)

determines whether the system $M \cdot x = b$ is solvable
Precondition: M.dim1() = b.dim().

bool homogeneous_linear_solver(*integer_matrix* M, *integer_vector*& x)

determines whether the homogeneous linear system $M \cdot x = 0$ has a non-trivial solution. If yes, then x is such a solution.

void independent_columns(*integer_matrix* M, *array<int>*& *columns*)

returns the indices of a maximal subset of independent columns of M. The index range of *columns* starts at 0.

int rank(*integer_matrix* M)

returns the rank of matrix M

ostream& *ostream*& $O \ll M$

writes matrix M row by row to the output stream O.

istream& *istream*& $I \gg$ *integer_matrix*& M

reads matrix M row by row from the input stream I.

4. Implementation

The datatype integer_matrix is implemented by two-dimensional arrays of integers. Operations *determinant*, *inverse*, *linear_solver*, and *rank* take time $O(n^3)$, *col* takes time $O(n)$, *row*, *dim1*, *dim2*, take constant time, and all other operations take time $O(nm)$. The space requirement is $O(nm)$.

All functions on integer matrices compute the exact result, i.e., there is no rounding error. The implemenation follows a proposal of J. Edmonds (J. Edmonds, Systems of distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241

-245). Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

2. The Header File of class integer_matrix

```
⟨ integer_matrix.h 2 ⟩ ≡
#ifndef LEDA_INTEGER_MATRIX_H
#define LEDA_INTEGER_MATRIX_H
#include <math.h>
#include <LEDA/basic.h>
#include <LEDA/array.h>
#include <LEDA/integer.h>
#include <LEDA/rational.h>
#include "integer_vector.h"
#define MAX_SIZE_OF_SMALL_OBJECT 256
  typedef integer integer_t;
  typedef rational rational_t;
  class integer_matrix {
    integer_vector **v;
    int d1;
    int d2;
    void check_dimensions(const integer_matrix &) const;
    integer_t &elem(int i, int j) const { return v[i]¬v[j]; }
    void allocate_small(integer_vector **&v, int d);
    void deallocate_small(integer_vector **v, int d);
    static int LEDA_SMALL;
  public:
    integer_matrix(int n = 0, int m = 0);
    integer_matrix(const array⟨integer_vector⟩ &A);
    integer_matrix(const integer_matrix &);
    integer_matrix(const integer_vector &);
    integer_matrix(int, int, integer_t **);
    integer_matrix &operator=(const integer_matrix &);
    ~integer_matrix( );
    int dim1( ) const { return d1; }
    int dim2( ) const { return d2; }
    integer_vector &row(int i) const
    {
#ifndef LEDA_CHECKING_OFF
      if (i < 0 ∨ i ≥ d1) error_handler(1, "integer_matrix::row: index out of range.");
#endif
      return *v[i];
    }
    integer_vector col(int i) const;
    integer_vector to_integer_vector( ) const;
    friend integer_vector to_integer_vector(const integer_matrix &);
    integer_vector &operator [](int i) const { return row(i); }
    integer_t &operator ( )(int i, int j)
    {
#ifndef LEDA_CHECKING_OFF
```

```
      if (i < 0 ∨ i ≥ d1)
        error_handler(1, "integer_matrix::operator(): row index out of range.");
      if (j < 0 ∨ j ≥ d2)
        error_handler(1, "integer_matrix::operator(): col index out of range.");
#endif
      return elem(i, j);
    }
    integer_t operator ()(int i, int j) const
    {
#ifndef LEDA_CHECKING_OFF
      if (i < 0 ∨ i ≥ d1)
        error_handler(1, "integer_matrix::operator(): row index out of range.");
      if (j < 0 ∨ j ≥ d2)
        error_handler(1, "integer_matrix::operator(): col index out of range.");
#endif
      return elem(i, j);
    }
    int operator≡(const integer_matrix &) const;
    int operator≠(const integer_matrix &x) const
    { return ¬(*this ≡ x); }
    integer_matrix operator+(const integer_matrix &M1);
    integer_matrix operator−(const integer_matrix &M1);
    integer_matrix operator−(); // unary
    integer_matrix &operator−=(const integer_matrix &);
    integer_matrix &operator+=(const integer_matrix &);
    integer_matrix operator ∗ (const integer_matrix &M1) const;
    integer_vector operator ∗ (const integer_vector &vec) const
    { return ((*this) ∗ integer_matrix(vec)).to_integer_vector(); }
    integer_matrix compmul(integer_t x) const;
    friend integer_matrix operator ∗ (const integer_matrix &M, integer_t x)
    { return M.compmul(x); }
    friend integer_matrix operator ∗ (integer_t x, const integer_matrix &M)
    { return M.compmul(x); }
    friend integer_matrix identity(int n);
    friend integer_matrix transpose(const integer_matrix &M);
    friend integer_matrix inverse(const integer_matrix &M, integer_t &D)
    {
      integer_matrix result;
      integer_vector c;
      if (inverse(M, result, D, c)) return result;
      error_handler(1, "integer_matrix::inverse: matrix is singular.");
    }
    friend bool inverse(const integer_matrix &M, integer_matrix &inverse, integer_t
        &D, integer_vector &c);
    friend integer_t determinant(const integer_matrix &M, integer_matrix
        &L, integer_matrix &U, array⟨int⟩ &q, integer_vector &c);
    friend bool verify_determinant(const integer_matrix &M, integer_t
        D, integer_matrix &L, integer_matrix &U, array⟨int⟩ q, integer_vector &c);
```

```
  friend integer_t determinant(const integer_matrix &M);
  friend int sign_of_determinant(const integer_matrix &M);
  friend bool linear_solver(const integer_matrix &M,
      const integer_vector &b, integer_vector &x, integer_t &D, integer_matrix
      &spanning_vectors, integer_vector &c);
  friend bool linear_solver(const integer_matrix &M, const integer_vector
        &b, integer_vector &x, integer_t &D, integer_vector &c)
  {
    integer_matrix spanning_vectors;
    return linear_solver(M, b, x, D, spanning_vectors, c);
  }
  friend bool linear_solver(const integer_matrix &M, const integer_vector
        &b, integer_vector &x, integer_t &D)
  {
    integer_matrix spanning_vectors;
    integer_vector c;
    return linear_solver(M, b, x, D, spanning_vectors, c);
  }
  friend bool solvable(const integer_matrix &M, const integer_vector &b)
  {
    integer_vector x;
    integer_t D;
    integer_matrix spanning_vectors;
    integer_vector c;
    return linear_solver(M, b, x, D, spanning_vectors, c);
  }
  friend bool homogeneous_linear_solver(const integer_matrix &M, integer_vector
      &x);
  friend void independent_columns(const integer_matrix &M, array⟨int⟩ &columns);
  friend int rank(const integer_matrix &M);
  friend ostream &operator≪(ostream &O, const integer_matrix &M);
  friend istream &operator≫(istream &I, integer_matrix &M);
  LEDA_MEMORY(integer_matrix);
};
inline void Print(const integer_matrix &m, ostream &out)
{ out ≪ m; }
inline void Read(integer_matrix &m, istream &in)
{ in ≫ m; }
inline int compare(const integer_matrix &, const integer_matrix &)
{
  error_handler(1, "compare: not defined for type `integer_matrix`.");
  return 0;
}
LEDA_TYPE_PARAMETER(integer_matrix)
#endif
```

3. The Implementation of class integer_matrix

We discuss the implementation of the class **integer_matrix** in two parts. In the first part we give the implementation of the constructors, the destructor, assignment, equality, all access functions, and all arithmetic operators, and in the second part we deal with the functions of linear algebra.

```
⟨ integer_matrix.c 3 ⟩ ≡
#include "integer_matrix.h"
  int integer_matrix::LEDA_SMALL = MAX_SIZE_OF_SMALL_OBJECT/sizeof(integer_vector);
  ⟨ basic functions 4 ⟩;
  ⟨ linear algebra 5 ⟩;
```

4. Construction and Basic Operations

A matrix is represented as a C++-array of pointers to row-vectors with integer entries (type **integer_vector**). Thus $v[i]$ is a pointer to a vector and hence $v[i]\neg v[j]$ is the entry (i,j) of the matrix.

```
⟨ basic functions 4 ⟩ ≡
  void integer_matrix::allocate_small(integer_vector **&v, int d)
  /* we use this procedure to allocate memory for small arrays. We first get an appropriate
  piece of memory from the LEDA memory manager and then initialize each cell by an inplace
  new. */
  {
    v = (integer_vector **) allocate_bytes(d * sizeof(integer_vector *));
    integer_vector **p = v + d - 1;
    while (p ≥ v) {
      new (p, 0) integer_vector*;
      p--;
    }
  }
  void integer_matrix::deallocate_small(integer_vector **v, int d)
  /* we use this procedure to deallocate memory for small arrays. The components are already
  deleted at this point and we only have to return the memory. */
  {
    deallocate_bytes(v, d * sizeof(integer_vector *));
  }
  integer_matrix::integer_matrix(int dim1, int dim2)
  {
    if (dim1 < 0 ∨ dim2 < 0)
      error_handler(1, "integer_matrix::constructor: negative dimension.");
    d1 = dim1;
    d2 = dim2;
    if (d1 > 0) {
      if (d1 < LEDA_SMALL) allocate_small(v, d1);
      else v = new integer_vector * [d1];
      for (int i = 0; i < d1; i++) v[i] = new integer_vector (d2);
    }
    else v = nil;
  }
```

```
integer_matrix::integer_matrix(const integer_matrix &p)
{
  d1 = p.d1;
  d2 = p.d2;
  if (d1 > 0) {
    if (d1 < LEDA_SMALL) allocate_small(v, d1);
    else v = new integer_vector * [d1];
    for (int i = 0; i < d1; i++) v[i] = new integer_vector (*p.v[i]);
  }
  else v = nil;
}
integer_matrix::integer_matrix(const array<integer_vector> &A)
{
  int al = A.low( );
  d2 = A.high( ) - al + 1;
  d1 = A[al].dim( );
  if (d1 > 0) {
    if (d1 < LEDA_SMALL) allocate_small(v, d1);
    else v = new integer_vector * [d1];
    for (int i = 0; i < d1; i++) {
      v[i] = new integer_vector (d2);
      for (int j = 0; j < d2; j++) v[i]->v[j] = A[al + j][i];
    }
  }
  else v = nil;
}
integer_matrix::integer_matrix(int dim1, int dim2, integer_t **p)
{
  d1 = dim1;
  d2 = dim2;
  if (d1 < LEDA_SMALL) allocate_small(v, d1);
  else v = new integer_vector * [d1];
  for (int i = 0; i < d1; i++) {
    v[i] = new integer_vector (d2);
    for (int j = 0; j < d2; j++) elem(i, j) = p[i][j];
  }
}
integer_matrix::~integer_matrix( )
{
  if (v) {
    int d = d1;
    while (d1--) delete v[d1];
    if (d < LEDA_SMALL) deallocate_small(v, d);
    else delete []v;
  }
}
void integer_matrix::check_dimensions(const integer_matrix &mat) const
{
  if (d1 ≠ mat.d1 ∨ d2 ≠ mat.d2)
```

```
    error_handler(1,
        "integer_matrix::check_dimensions: incompatible matrix types.");
}
integer_matrix::integer_matrix(const integer_vector &vec)
{
  d1 = vec.d;
  d2 = 1;
  if (d1 < LEDA_SMALL) allocate_small(v, d1);
  else v = new integer_vector * [d1];
  for (int i = 0; i < d1; i++) {
    v[i] = new integer_vector (1);
    elem(i, 0) = vec[i];
  }
}
integer_matrix &integer_matrix::operator=(const integer_matrix &mat)
{
  register int i, j;
  if (d1 ≠ mat.d1 ∨ d2 ≠ mat.d2) {
    for (i = 0; i < d1; i++) delete v[i];
    if (v) {
      if (d1 < LEDA_SMALL) deallocate_small(v, d1);
      else delete []v;
    }
    d1 = mat.d1;
    d2 = mat.d2;
    if (d1 < LEDA_SMALL) allocate_small(v, d1);
    else v = new integer_vector * [d1];
    for (i = 0; i < d1; i++) v[i] = new integer_vector (d2);
  }
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) elem(i, j) = mat.elem(i, j);
  return *this;
}
int integer_matrix::operator≡(const integer_matrix &x) const
{
  register int i, j;
  if (d1 ≠ x.d1 ∨ d2 ≠ x.d2) return false;
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++)
      if (elem(i, j) ≠ x.elem(i, j)) return false;
  return true;
}
integer_vector integer_matrix::col(int i) const
{
  if (i < 0 ∨ i ≥ d2) error_handler(1, "integer_matrix::col: index out of range.");
  integer_vector result(d1);
  int j = d1;
  while (j--) result.v[j] = elem(j, i);
  return result;
```

```
}
integer_vector integer_matrix::to_integer_vector() const
{
  if (d2 ≠ 1) error_handler(1,"integer_matrix::to_integer_vector:␣cann\
      ot␣make␣integer_vector␣from␣matrix.");
  return col(0);
}
integer_vector to_integer_vector(const integer_matrix &M)
{ return M.to_integer_vector(); }
integer_matrix integer_matrix::operator+(const integer_matrix &mat)
{
  register int i, j;
  check_dimensions(mat);
  integer_matrix result(d1, d2);
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) result.elem(i, j) = elem(i, j) + mat.elem(i, j);
  return result;
}
integer_matrix integer_matrix::operator-(const integer_matrix &mat)
{
  register int i, j;
  check_dimensions(mat);
  integer_matrix result(d1, d2);
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) result.elem(i, j) = elem(i, j) - mat.elem(i, j);
  return result;
}
integer_matrix &integer_matrix::operator+=(const integer_matrix &mat)
{
  register int i, j;
  check_dimensions(mat);
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) elem(i, j) += mat.elem(i, j);
  return *this;
}
integer_matrix &integer_matrix::operator-=(const integer_matrix &mat)
{
  register int i, j;
  check_dimensions(mat);
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) elem(i, j) -= mat.elem(i, j);
  return *this;
}
integer_matrix integer_matrix::operator-()  // unary
{
  register int i, j;
  integer_matrix result(d1, d2);
```

```
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) result.elem(i, j) = -elem(i, j);
  return result;
}
integer_matrix integer_matrix::compmul(integer_t f) const
{
  register int i, j;
  integer_matrix result(d1, d2);
  for (i = 0; i < d1; i++)
    for (j = 0; j < d2; j++) result.elem(i, j) = elem(i, j) * f;
  return result;
}
integer_matrix integer_matrix::operator *(const integer_matrix &mat) const
{
  if (d2 ≠ mat.d1)
    error_handler(1, "integer_matrix::operator*: incompatible matrix types.");
  integer_matrix result(d1, mat.d2);
  register int i, j;
  for (i = 0; i < mat.d2; i++)
    for (j = 0; j < d1; j++) result.elem(j, i) = *v[j] * mat.col(i);
  return result;
}
ostream &operator≪(ostream &s, const integer_matrix &M)
{
  int i;
  s ≪ "\n";
  for (i = 0; i < M.d1; i++) s ≪ M[i] ≪ "\n";
  return s;
}
istream &operator≫(istream &s, integer_matrix &M)
{
  int i = 0;
  while (i < M.d1 ∧ s ≫ M[i++]) ;
  return s;
}
integer_matrix identity(int n)
{
  integer_matrix result(n, n);  // sets all entries zero
  for (int i = 0; i < n; i++) result(i, i) = 1;  // sets diag entries to one
  return result;
}
integer_matrix transpose(const integer_matrix &M)
{
  int d1 = M.dim1();
  int d2 = M.dim2();
  integer_matrix result(d2, d1);
  for (int i = 0; i < d2; i++)
    for (int j = 0; j < d1; j++) result.elem(i, j) = M(j, i);
```

return *result*;
}

This code is used in chunk 3.

5. Linear Algebra

We now turn to the functions of linear algebra. We frequently need a function that swaps the values of its arguments.

⟨ linear algebra 5 ⟩ ≡
template⟨**class T**⟩ **void** *swap*(**T** &x, **T** &y)
{ **T** $h = x$; $x = y$; $y = h$; }

See also chunks 6, 15, 16, 17, 18, and 20.

This code is used in chunk 3.

6. The most important operation is the solution of a non-homogeneous linear system. We use Gaussian elimination as described by J. Edmonds (J. Edmonds, Systems of distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241 -245) .

Consider a linear system $A \cdot x = b$. Gaussian elimination operates in two phases. In the first phase it transforms A by a sequence of row and column operations into an upper diagonal matrix and in the second phase it solves the resulting upper diagonal system.

Let us have a closer look at the first phase. It operates in subphases, numbered 0 to $m-1$ where m is the number of rows of A. Before the k-th subphase we have transformed A into a matrix $C_k = \begin{bmatrix} D & E \\ 0 & F \end{bmatrix}$, where D is a non-singular upper triangular matrix of order k. For $k = 0$ we have $C_0 = A$. In the k-th subphase we first determine a non-zero element in F (called the *pivot-element*) and move it into the left upper corner of F by interchanging rows and columns if necessary and then subtract suitable multiples of the top row of F from the other rows of F to zero out all below diagonal entries in F's first column. We assume for ease of exposition that no interchanging of rows and columns is ever necessary.

Lemma 1 *All entries of F can be written as rational numbers with denominator* $\det A_k$ *where A_k is the submatrix formed by the first k rows and columns of A.*

Proof: For a matrix M and row indices $i_1, \ldots, i_k$ and column indices $j_1, \ldots, j_l$ use $M_{i_1,\ldots,i_l}^{j_1,\ldots,j_l}$ to denote the $l \times l$ submatrix formed by the elements in the selected rows and columns. Consider any entry f_{ij} of F. We have

$$f_{ij} = \frac{\det((C_k)_{0,\ldots,k-1,i}^{0,\ldots,k-1,j})}{\det((C_k)_{0,\ldots,k-1}^{0,\ldots,k-1})}$$

since the matrix in the numerator has f_{ij} in the right lower corner, zeroes in all other entries of the last row, and the matrix in the denominator in its other row and columns. The numerator is therefore f_{ij} times the denominator. Observe next that both determinants do not change if we write C_0 instead of C_k since C_k is obtained from C_0 by subtracting multiples of the first k rows from the other rows. Thus,

$$f_{ij} = \frac{\det((C_0)_{0,\ldots,k-1,i}^{0,\ldots,k-1,j})}{\det((C_0)_{0,\ldots,k-1}^{0,\ldots,k-1})}$$

∎

The Lemma above suggests to take $\det A_k$ as the denominator of all entries of F, to store the denominator separately, say in a variable *denom*,and to keep only the numerators in the matrix F. Thus, we maintain the invariants

$$\begin{array}{ll} (1) & denom = \det A_k \\ (2) & F_{ij} = \det A^{0,\ldots,k-1,j}_{0,\ldots,k-1,i}, \text{ for } i \geq k, j \geq k \\ (3) & f_{ij} = F_{ij}/denom. \end{array}$$

The effect of the k-th subphase is to replace f_{ij} by

$$\begin{aligned} f'_{ij} &= f_{ij} - f_{ik} \cdot f_{kj}/f_{kk} \\ &= (f_{ij} \cdot f_{kk} - f_{ik} \cdot f_{kj})/f_{kk} \\ &= ((F_{ij} \cdot F_{kk} - F_{ik} \cdot F_{kj})/\det A_k)/F_{kk} \end{aligned}$$

for $i > k$ and $j \geq k$.

Lemma 2 *$F_{kk} = \det A_{k+1}$ and the division $(F_{ij}F_{kk} - F_{ik}F_{kj})/\det A_k$ is without remainder.*

Proof: Invariant (2) shows

$$F_{kk} = \det(A^{0,\ldots,k-1,k}_{0,\ldots,k-1,k}) = \det A_k$$

and Lemma 1 tells us that f'_{ij} can be written as a rational number with denominator F_{kk}. Thus, $(F_{ij}F_{kk} - F_{ik}F_{kj})/\det A_k$ must be an integer. ∎

We summarize. We take a matrix C and initialize it with A. Before the k-th subphase we have

$$\begin{array}{ll} (1) & C_{ii} = \det A_{i+1} \text{ for } 0 \leq i \leq k \\ (2) & C^{0,\ldots,k-1}_{0,\ldots,k-1} \text{ is non-singular upper triangular} \\ (3) & C_{ij} = 0 \text{ for } i \geq k \text{ and } j < k \\ (4) & C_{ij} = \det A^{0,\ldots,k-1,j}_{0,\ldots,k-1,i} \text{ for } i \geq k, j \geq k \\ (5) & C_{ij} = \det A^{0,\ldots,i-1,j}_{0,\ldots,i-1,i} \text{ for } i < k, j > i \end{array}$$

In the k-th subphase we set

$$C_{ij} = (C_{ij}C_{kk} - C_{ik}C_{kj})/\det A_k$$

For $i > k$ and $j \geq k$. The division is without remainder.

So far we ignored pivoting and the right-hand side b. We handle b by adjoining it to A as an additional column. We handle pivoting by storing all column interchanges (there is no need to keep track of the row interchanges since we adjoined b to A and hence handle both sides of the equation in the same way). We store the column interchanges in an array *var*: What is now column j was originally column $var[j]$. In other words, column j of C represents variable $var[j]$.

It is useful to keep track of all row operations performed. We do so in a matrix L which we initialize to an $m \times m$ identity matrix and subject to the same row operations as C, i.e., in the k-th subphase we set

$$L'_{ij} = (L_{ij} \cdot C_{kk} - C_{ik} \cdot L_{kj})/\det A_k$$

for $i > k$ and all j. This maintains the invariant

$$L \cdot (A|b) \cdot P = C$$

where $(A|b)$ denotes the matrix obtained by adjoining b to A, P is the permutation matrix corresponding to *var* and C is the matrix we are working in. We initialize C with $(A|b)$ and we intialize P with the identity matrix.

We still need to fill in two details in our treatment of phase 1: Why do the entries of L stay integral and how do we discover unsolvability?

Lemma 3 *The entries of L stay integral.*

Proof: Let us again ignore pivoting. Then

$$L = \begin{bmatrix} U & 0 \\ V & W \end{bmatrix}$$

before subphase k where U is a lower diagonal matrix of order k and W is a diagonal matrix. It is readily proved by induction on k that all diagonal entries of W are equal to $\det A_k$, i.e., $W = (\det A_k) \cdot I$, and that the diagonal entries of U are $\det A_0$, $\det A_1$, ..., $\det A_{k-1}$. Let

$$A = \begin{bmatrix} A_k & \cdots \\ A'_k & \cdots \end{bmatrix}.$$

Then $U \cdot A_k = D$ and $V \cdot A_k + W \cdot A'_k = 0$. Thus, $U = DA_k^{-1}$ and $V = -WA'_kA_k^{-1} = -(\det A_k) \cdot A'_kA_k^{-1}$. This implies already the integrality of V since all entries of A_k^{-1} are rational numbers whose denominator divides $\det A_k$ (by Cramers's rule). For matrix U we at least know that it is uniquely defined. It therefore suffices to show that there is an integral matrix U satisfying $U \cdot A_k = D$. This is easy to see. We have $D_{ij} = \det A^{0,\ldots,i-1,j}_{0,\ldots,i-1,i}$. Expansion according to the last column yields

$$D_{ij} = \sum (-1)^{l+j} \det A^{0,\ldots,i-1}_{0,\ldots,l-1,l+1,\ldots,i} A_{lj}.$$

Thus, $U_{ij} = (-1)^{l+j} \det A^{0,\ldots,i-1}_{0,\ldots,l-1,l+1,\ldots,i}$. ∎

Unsolvability is easy to detect. We discover unsolvability in subphase k when the search for a non-zero pivot is unsuccessful but the current right hand side has a non-zero entry in row k or below.

We are now ready for the program.

```
⟨ linear algebra 5 ⟩ +≡
  bool linear_solver(const integer_matrix &A, const integer_vector &b,
          integer_vector &x, integer_t &D,
          integer_matrix &spanning_vectors, integer_vector &c)
  {
    bool solvable = true;
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    ⟨ test whether a solution exists and compute c if there is no solution 13 ⟩
    if (solvable) {
      ⟨ compute solution space 14 ⟩
    }
    return solvable;
  }
```

7. Initialization.

⟨ initialize C and L from A and b 7 ⟩ ≡

```
int i, j, k;  // indices to step through the matrix
int rows = A.dim1();
int cols = A.dim2();
/* at this point one might want to check whether the computation can be carried out with
doubles, see section 21. */
if (b.dim() ≠ rows) error_handler(1,"linear_solver: b has wrong dimension");
integer_matrix C(rows, cols + 1);  // the matrix in which we will calculate (C = (A|b))
/* copy A and b into C */
for (i = 0; i < rows; i++) {
  for (j = 0; j < cols; j++) C(i, j) = A(i, j);
  C(i, cols) = b[i];
}
integer_matrix L(rows, rows);  // is initialized with zeros
for (i = 0; i < rows; i++) L(i, i) = 1;  // sets diagonal elements to one
```

This code is used in chunks 6, 15, 16, 17, 18, and 20.

8. Phase 1 of Gaussian elimination.

⟨ phase 1 8 ⟩ ≡

```
array⟨int⟩ var(0, cols − 1);  // column j of C represents the var[j]-th variable
// the array is indexed between 0 and cols − 1
for (j = 0; j < cols; j++) var[j] = j;  // at the beginning, variable x_j stands in column j
integer_t denom = 1;  // the determinant of an empty matrix is 1
int sign = 1;  // no interchanges yet
int rank = 0;  // we have not seen any non-zero row yet
/* here comes the main loop */
for (k = 0; k < rows; k++) {
  ⟨ search for a non-zero element; if found it is in (i, j) 9 ⟩
  if (non_zero_found) {
    rank++;  // increase the rank
    ⟨ interchange rows k and i and columns k and j 10 ⟩
    ⟨ one subphase of phase 1 11 ⟩
  }
  else break;
}
```

This code is used in chunks 6, 15, 16, 17, 18, and 20.

9. We search for a non-zero element.

⟨ search for a non-zero element; if found it is in (i, j) 9 ⟩ ≡

```
bool non_zero_found = false;
for (i = k; i < rows; i++)  // step through rows k to rows − 1
{
  for (j = k; j < cols ∧ C(i, j) ≡ 0; j++) ;  // step through columns k to cols − 1
  if (j < cols) { non_zero_found = true; break; }
}
```

This code is used in chunk 8.

10. We interchange rows and columns. Any exchange changes the sign of the determinant.

```
⟨ interchange rows k and i and columns k and j 10 ⟩ ≡
  if (i ≠ k) {
    sign = −sign;
    /* we interchange rows k and i of L and C */
    swap(L[i], L[k]);
    swap(C[i], C[k]);
  }
  if (j ≠ k) {
    sign = −sign;
    /* We interchange columns k and j */
    for (int ih = 0; ih < rows; ih++) swap(C(ih, k), C(ih, j));
    /* We store the exchange of variables in var */
    swap(var[k], var[j]);
  }
```

This code is used in chunk 8.

11. Now we are ready to do the pivot-step with the element $C_{k,k}$. We do the L's first since we want to work with the old values of C. Note that the division by *denom* is allowed as this factor is contained in the nominator term.

```
⟨ one subphase of phase 1 11 ⟩ ≡
  for (i = k + 1; i < rows; i++)
    for (j = 0; j < rows; j++)  // and all columns of L
      L(i, j) = (L(i, j) * C(k, k) − C(i, k) * L(k, j))/denom;
  for (i = k + 1; i < rows; i++) {
    /* the following iteration uses and changes C(i, k) */
    integer_t temp = C(i, k);
    for (j = k; j ≤ cols; j++) C(i, j) = (C(i, j) * C(k, k) − temp * C(k, j))/denom;
  }
  denom = C(k, k);
  ⟨ check invariant L · A · P = C 12 ⟩
```

This code is used in chunk 8.

12. It is good custom to check the state of a computation. The matrix L multiplied by A permuted as given by *var* should be equal to the current C. The permutation *var* moves column *var*$[j]$ of A to column j of C.

```
⟨ check invariant L · A · P = C 12 ⟩ ≡
#ifdef LEDA_TEST
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      integer_t Sum = 0;
      for (int l = 0; l < rows; l++) Sum += L(i, l) * A(l, var[j]);
      if (Sum ≠ C(i, j)) error_handler(1, "linear_solver: L*A*P different from C");
    }
    integer_t Sum = 0;
```

```
    for (int l = 0; l < rows; l++) Sum += L(i,l) * A(l,cols);
    if (Sum ≠ C(i,cols)) error_handler(1,"linear_solver: L*A*P different from C");
  }
#endif
```

This code is used in chunk 11.

13. We are done with Gaussian elimination. At this point C has a $rank \times rank$ upper triangular matrix in its left upper corner and the remaining rows of C are zero. The system is solvable if the current right hand side has no non-zero entry in row *rank* and below. Assume otherwise, say $C(i, cols) \neq 0$. Then the i-th row of L proves the unsolvability of the system.

```
⟨ test whether a solution exists and compute c if there is no solution 13 ⟩ ≡
  for (i = rank; i < rows ∧ C(i,cols) ≡ 0; i++) ; // no body
  if (i < rows) {
    solvable = false;
    c = integer_vector(rows);
    for (j = 0; j < rows; j++) c[j] = L(i,j);
  }
```

This code is used in chunk 6.

14. We compute the solution space. It is determined by a solution x of the non-homogeneous system plus $cols - rank$ linearly independent solutions to the homogeneous system.

Recall that C has a $rank \times rank$ upper triangular matrix in its upper left corner. We view the variables corresponding to the first *rank* columns as dependent and the others as independent. The vector *var* connects columns with variables: column j represents variable $var[j]$.

The components of x are rational numbers with common denominator *denom* (by Cramer's rule and since *denom* is (up to sign) equal to the determinant of the submatrix formed by the dependent variables). We set the components corresponding to the independent variables to zero and compute the components corresponding to the dependent variables by back substitution. During back substitution we compute x_i (again ignoring pivoting) by $x_i = (b_i - \sum_{j>i} c_{i,j}x_j)/c_{i,i}$. Let $x[i]$ be the numerator of x_i. Then $x[i] = (b_i * denom - \sum_{j>i} c_{i,j} * x[j])/c_{i,i}$.

We have a spanning vector for each independent variable. We set the value of the independent variable to $1 = denom/denom$ and then compute the values of all dependent variables.

```
⟨ compute solution space 14 ⟩ ≡
  x = integer_vector(cols);
  D = denom;
  for (i = rank - 1; i ≥ 0; i--) {
    integer_t h = C(i,cols) * D;
    for (j = i + 1; j < rank; j++) {
      h -= C(i,j) * x[var[j]];
    }
    x[var[i]] = h/C(i,i);
  }
#ifdef LEDA_TEST
  /* we check whether x is a solution */
  {
    for (i = 0; i < rows; i++) {
      integer_t sum = 0;
```

```
      for (j = 0; j < cols; j++) sum += A(i, j) * x[j];
      if (sum ≠ D * b[i]) error_handler(1, "linear_solver: base is not a solution");
    }
  }
#endif
  int dimension = cols − rank;  // dimension of solution
  spanning_vectors = integer_matrix(cols, dimension);
  if (dimension > 0) {
    /* In the l-th spanning vector, 0 ≤ l < dimension we set variable var[rank + l] to 1 =
    denom/denom and then the dependent variables as dictated by the rank + l-th column of
    C. */
    for (int l = 0; l < dimension; l++) {
      spanning_vectors(var[rank + l], l) = D;
      for (i = rank − 1; i ≥ 0; i−−) {
        integer_t h = −C(i, rank + l) * D;
        for (j = i + 1; j < rank; j++) h −= C(i, j) * spanning_vectors(var[j], l);
        spanning_vectors(var[i], l) = h/C(i, i);
      }
#ifdef TEST
      /* we check whether the l-th spanning vector is a solution of the homogeneous system
      */
      {
        integer_vector zero(rows);
        if (A * spanning_vectors.col(l) ≠ zero)
          error_handler(1, "linear_solver: spanning_vector is not a solution.");
      }
#endif
    }
  }
```

This code is used in chunks 6 and 20.

15. This completes the implementation of the linear solver. We next turn to the functions *determinant*, and *inverse*, and *rank*.

The determinant function is simple. The determinant is either zero (if the matrix does not have full rank) or is *sign* * *denom* where *sign* and *denom* are computed during phase 1.

```
⟨ linear algebra 5 ⟩ +≡
  integer_t determinant(const integer_matrix &A)
  {
    if (A.dim1( ) ≠ A.dim2( ))
      error_handler(1, "determinant: only square matrices are legal inputs.");
    integer_vector b(A.dim1( ));  // zero-vector
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    if (rank < rows) return 0;
    else return integer_t(sign) * denom;
  }
```

16. The certified version of the determinant is slightly more interesting. If the matrix is singular then the last row of L proves that fact, i.e., is a vector c with $c^T \cdot A = 0$. If the matrix is non-singular then we return the LU-decomposition. Note that no row interchanges took place during the first phase of Gaussian elimination when A is non-singular. Thus L is really a lower diagonal matrix.

We need to check whether L and U are really lower and upper diagonal matrices, whether the diagonal elements are as prescribed, whether $L \cdot A \cdot Q = U$ and whether the claimed value of the determinant is equal to the sign of Q times $U(n-1, n-1)$. Since Q is given implicitely by a **array**⟨**int**⟩ q the handling of Q requires a little bit of work.

We first check whether q is indeed a permutation by checking whether it is surjective and then compute the sign of q. To do so we trace the cycles of q. A cycle of length l contributes $(-1)^{l-1}$ to the sign. We trace each cycle starting at its smallest element.

```
⟨ linear algebra 5 ⟩ +≡
  integer_t determinant(const integer_matrix &A, integer_matrix &LD,
          integer_matrix &UD, array⟨int⟩ &q, integer_vector &c)
  {
    if (A.dim1( ) ≠ A.dim2( ))
      error_handler(1, "determinant: only square matrices are legal inputs.");
    integer_vector b(A.dim1( )); // zero-vector
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    if (rank < rows) {
      c = L.row(rows − 1);
      return 0;
    }
    else {
      LD = L;
      UD = integer_matrix(rows, rows);
      for (i = 0; i < rows; i++)
        for (j = 0; j < rows; j++) UD(i, j) = C(i, j);
      q = var;
      return integer_t(sign) ∗ denom;
    }
  }
  int sign_of_determinant(const integer_matrix &M)
  {
    return sign(determinant(M));
  }
  bool verify_determinant(const integer_matrix &A, integer_t D,
          integer_matrix &L, integer_matrix &U, array⟨int⟩ q, integer_vector &c)
  {
    if (q.low( ) ≠ 0 ∨ q.high( ) ≠ A.dim2( ) − 1) error_handler(1, "verify_determinant:\
          q should be a permutation array with index range [0,A.dim2()-1].");
    int n = A.dim1( );
    int i, j;
    if (D ≡ 0) {
      /∗ we have c^T · A = 0 ∗/
      integer_vector zero(n);
      return (transpose(A) ∗ c ≡ zero);
```

```
  }
  else {
    /* we check the conditions on L and U */
    if (L(0,0) ≠ 1) return false;
    for (i = 0; i < n; i++) {
      for (j = 0; j < i; j++)
        if (U(i,j) ≠ 0) return false;
      if (i > 0 ∧ L(i,i) ≠ U(i-1,i-1)) return false;
      for (j = i+1; j < n; j++)
        if (L(i,j) ≠ 0) return false;
    }
    /* check whether L · A · Q = U */
    integer_matrix LA = L * A;
    for (j = 0; j < n; j++)
      if (LA.col(q[j]) ≠ U.col(j)) return false;
      /* compute sign s of Q */
    int sign = 1;
    /* we chase the cycles of q. An even length cycle contributes -1 and vice versa */
    array⟨bool⟩ already_considered(0, n-1);
    for (i = 0; i < n; i++) already_considered[i] = false;
    for (i = 0; i < n; i++) already_considered[q[i]] = true;
    for (i = 0; i < n; i++)
      if (¬already_considered[i])
        error_handler(1, "verify_determinant:q is not a permutation.");
      else already_considered[i] = false;
    for (i = 0; i < n; i++) {
      if (already_considered[i]) continue;
      /* we have found a new cycle with minimal element i. */
      int k = q[i];
      already_considered[i] = true;
      while (k ≠ i) {
        sign = -sign;
        already_considered[k] = true;
        k = q[k];
      }
    }
    return (D ≡ integer_t(sign) * U(n-1, n-1));
  }
}
```

17. To determine a maximal subset of independent columns of a matrix we use the main modules of our linear solver and collect the columns which are represented by the independent variables after the diagonalization of the matrix.

```
⟨ linear algebra 5 ⟩ +≡
  void independent_columns(const integer_matrix &A, array⟨int⟩ &columns)
  {
  integer_vector b(A.dim1( )); // zero-vector
```

```
⟨ initialize C and L from A and b 7 ⟩
⟨ phase 1 8 ⟩
/* at this point we have: C has an rank × rank upper triangular matrix in its left upper
corner; var tells us the columns of A corresponding to the dependent variables; */
columns = array⟨int⟩ (rank);
for (i = 0; i < rank; i++) columns[i] = var[i];
}
```

18. *rank* and *inverse* are easy to implement. The rank is already computed during phase 1 and the i-th column of the inverse is a solution to the linear system $A \cdot x = e_i$ where e_i is the i-th unit-vector.

```
⟨ linear algebra 5 ⟩ +≡
  int rank(const integer_matrix &A)
  {
    integer_vector b(A.dim1( )); // zero-vector
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    return rank;
  }
  bool inverse(const integer_matrix &A, integer_matrix &inverse,
          integer_t &D, integer_vector &c)
  {
    if (A.dim1( ) ≠ A.dim2( ))
      error_handler(1, "inverse: only square matrices are legal inputs.");
    integer_vector b(A.dim1( )); // zero-vector
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    if (rank < rows) {
      /* matrix is singular; we return a vector c with c^T · A = 0. */
      c = integer_vector(rows);
      for (j = 0; j < rows; j++) c[j] = L(rows − 1, j);
      return false;
    }
    ⟨ complete computation of inverse 19 ⟩
    return true;
  }
```

19. In order to compute the i-th column of the inverse we need to solve the linear system $A \cdot x = e_i$, where e_i is the i-th unit vector. We have already subjected e_i to the first phase of Gaussian elimination as the i-th column of L. Thus we only have to perform back substitution with this column in order to get the i-th column of the inverse.

```
⟨ complete computation of inverse 19 ⟩ ≡
  D = denom;
  inverse = integer_matrix(rows, rows);
  integer_t h;
```

```
  for (i = 0; i < rows; i++) {  // i-th column of inverse
    for (j = rows − 1; j ≥ 0; j−−) {
      h = L(j, i) ∗ D;
      for (int l = j + 1; l < rows; l++) h −= C(j, l) ∗ inverse(var[l], i);
      inverse(var[j], i) = h/C(j, j);
    }
  }
#ifdef TEST
  if (¬(A ∗ inverse ≡ identity(rows) ∗ D))
    error_handler(1, "inverse: matrix inverse computed incorrectly.");
#endif
```

This code is used in chunk 18.

20. Let's next solve a homogeneous system.

```
⟨ linear algebra 5 ⟩ +≡
  bool homogeneous_linear_solver(const integer_matrix &A, integer_vector &x)
  /∗ returns true if the homogeneous system Ax = 0 has a non-trivial solution and false otherwise. ∗/
  {
    integer_vector b(A.dim1( ));  // zero-vector
    integer_t D;
    ⟨ initialize C and L from A and b 7 ⟩
    ⟨ phase 1 8 ⟩
    integer_matrix spanning_vectors;
    ⟨ compute solution space 14 ⟩;
    if (dimension ≡ 0) return false;
    /∗ return first column of spanning_vectors ∗/
    for (i = 0; i < cols; i++) x[i] = spanning_vectors(i, 0);
    return true;
  }
```

21. Optimization.

When can we carry out the computation with double arithmetic? Double arithmetic works as long as all determinants computed are at most 2^{52} in absolute value. The determinant of a matrix is bounded by the product of the norms of the column vectors. The norm of a vector is bounded by $\sqrt{n}M$ where M is the maximal absolute value of any matrix entry. The log of the determinant is therefore bounded by $n(\log M + (1/2) \cdot \log n)$.

We need to square this number since we compute terms of the form $D_1 D/D$ and hence before the division our numbers may be as large as the square of the determinant.

```
⟨ unused code: does double arithmetic suffice? 21 ⟩ ≡
  {
    double max_entry = 1;
    for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++) {
        if (fabs(A(i, j)) > max_entry) max_entry = fabs(A(i, j));
      }
    if (rows ∗ (log(max_entry) + log(rows)/2) ≤ 26)
      ⟨ convert the input to double and use double arithmetic 0 ⟩;
```

```
}
```

22. A Test of class integer_matrix

And finally we test our vector type in a little program. We first test construction and access operations. We test our equality operators. Then we do some basic calculations with matrix-matrix and matrix-vector operations. We also use *transpose*, *rank* to characterize example matrices. Then we test the modules *inverse* and *linear_solver* of the package.

```
⟨ integer_matrix-test.c 22 ⟩ ≡
#include "integer_matrix.h"
#define MAT_DIM 10
  main( )
  { /* some construction and access ops */
    integer_matrix A(MAT_DIM, MAT_DIM), B(MAT_DIM, MAT_DIM);
    for (int i = 0; i < MAT_DIM; i++)
      for (int j = 0; j < MAT_DIM; j++) {
        A(i, j) = i;
        B(i, j) = j;
      }
    integer_matrix C(A);
    /* some dimension and equality testing */
    cout ≪ "Our test input with dimensions: ";
    cout ≪ A.dim1( ) ≪ "x" ≪ A.dim2( ) ≪ "\n\n";
    cout ≪ "Matrix A:" ≪ A ≪ "\n";
    cout ≪ "Matrix B:" ≪ B ≪ "\n";
    cout ≪ "Matrix C:" ≪ C ≪ "\n";
    cout ≪ "A " ≪ (A ≡ B ? "==" : "!=") ≪ " B\n";
    cout ≪ "A " ≪ (A ≡ C ? "==" : "!=") ≪ " C\n";
    /* some basic arithmetic testing */
    C += A;
    C −= A ∗ 3;
    C = −C;
    cout ≪ "C " ≪ (C ≡ A ? "==" : "!=") ≪ " -(C +A -3A)\n\n";
    /* some row sums: */
    integer_vector ones(MAT_DIM);
    for (int i = 0; i < MAT_DIM; i++) ones[i] = 1;
    cout ≪ "A *1-vec= " ≪ A ∗ ones ≪ "\n";
    cout ≪ "B *1-vec= " ≪ B ∗ ones ≪ "\n\n";
    /* matrix operations +,* and transpose, rank */
    cout ≪ "A +B= " ≪ (C = A + B) ≪ "is ";
    cout ≪ (C ≡ transpose(C) ? "symmetric" : "not symmetric");
    cout ≪ " and has rank " ≪ rank(C) ≪ "\n\n";
    cout ≪ "A *B= " ≪ (C = A ∗ B) ≪ "is ";
    cout ≪ (C ≡ transpose(C) ? "symmetric" : "not symmetric");
    cout ≪ " and has rank " ≪ rank(C) ≪ "\n\n";
    /* matrix operations 2* and identity, determinant */
    C = 2 ∗ identity(MAT_DIM);
    cout ≪ "Matrix 2 *I:" ≪ C ≪ "has determinant ";
    integer_matrix L, U;
    integer_vector c;
    array⟨int⟩ q;
```

```
  integer det = determinant(C, L, U, q, c);
  bool ok = verify_determinant(C, det, L, U, q, c);
  cout << det << " which is " << (ok ? "ok" : "not ok") << "\n\n";
  /* matrix operation inverse */
  C += (A + B);
  integer_matrix D;
  integer denom;
  bool invertible = inverse(C, D, denom, c);
  cout << "if we add this matrix and A+B we get" << C;
  cout << "\nthis matrix has rank " << rank(C) << " and is ";
  cout << (invertible ? "" : "not ") << "invertible:\n";
  if (invertible) cout << "Inverse = 1/" << denom << " * " << D;
  else cout << "Proofvector = " << c;
  cout << "\nif we multiply both we get:" << C * D << "\n";
  /* a random linear solver task: */
  integer_matrix E(MAT_DIM, MAT_DIM);
  integer_vector b(MAT_DIM), c(MAT_DIM);
  integer_vector x(MAT_DIM);
  random_source ranso;
  for (int i = 0; i < MAT_DIM; i++) {
    for (int j = 0; j < MAT_DIM; j++)
      E(i, j) = ranso(-MAT_DIM, MAT_DIM);
    b[i] = ranso(-MAT_DIM, MAT_DIM);
  }
  cout << "random linear system (E,b) with dimensions ";
  cout << E.dim1() << "x" << E.dim2() + 1;
  cout << E << "\n" << b << "\n\n";
  if (linear_solver(E, b, x, denom, A, e)) {
    cout << "solvable with solution x:\n" << x << "\n";
    cout << "E * x " << (E * x == denom * b ? "==" : "!=") << " b\n";
  }
  else {
    cout << "not solvable with proof e: " << e << "\n";
    cout << "e * E = " << (transpose(E) * e) << "\n";
    cout << "b * E = " << (b * e) << "\n";
  }
}
```

Index

List of Refinements

⟨ basic functions 4 ⟩ Used in chunk 3.
⟨ check invariant $L \cdot A \cdot P = C$ 12 ⟩ Used in chunk 11.
⟨ complete computation of inverse 19 ⟩ Used in chunk 18.
⟨ compute solution space 14 ⟩ Used in chunks 6 and 20.
⟨ convert the input to double and use double arithmetic 0 ⟩ Used in chunk 21.
⟨ initialize C and L from A and b 7 ⟩ Used in chunks 6, 15, 16, 17, 18, and 20.
⟨ `integer_matrix-test.c` 22 ⟩
⟨ `integer_matrix.c` 3 ⟩
⟨ `integer_matrix.h` 2 ⟩
⟨ interchange rows k and i and columns k and j 10 ⟩ Used in chunk 8.
⟨ linear algebra 5, 6, 15, 16, 17, 18, 20 ⟩ Used in chunk 3.
⟨ one subphase of phase 1 11 ⟩ Used in chunk 8.
⟨ phase 1 8 ⟩ Used in chunks 6, 15, 16, 17, 18, and 20.
⟨ search for a non-zero element; if found it is in (i, j) 9 ⟩ Used in chunk 8.
⟨ test whether a solution exists and compute c if there is no solution 13 ⟩ Used in chunk 6.
⟨ unused code: does double arithmetic suffice? 21 ⟩

Vectors with Integer Entries (class integer_vector)

Geokernel

May 21, 1996

Abstract

An instance of data type integer_vector is a vector of variables of type integer. Together with the type integer_matrix it realizes the basic operations of linear algebra.

Contents

1. The Manual Page of class integer_vector

1. Definition

An instance of the data type *integer_vector* is a vector of variables of type integer.

2. Creation

integer_vector v; creates an instance v of type *integer_vector*; v is initialized to the zero-dimensional vector.

integer_vector $v(int\ d)$; creates an instance v of type *integer_vector*; v is initialized to the zero vector of dimension d.

integer_vector $v(integer\ a, integer\ b)$;
creates an instance v of type *integer_vector*; v is initialized to the two-dimensional vector (a, b).

integer_vector $v(integer\ a, integer\ b, integer\ c)$;
creates an instance v of type *integer_vector*; v is initialized to the three-dimensional vector (a, b, c).

3. Operations

int	v.dim( )	returns the dimension of v.
integer&	$v[int\ i]$	returns i-th component of v. *Precondition*: $0 \leq i \leq v.dim(\ ) - 1$. This check can be turned off by the flag `LEDA_CHECKING_OFF`.
integer_vector&	$v+= v1$	Addition plus assignment. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector&	$v-= v1$	Subtraction plus assignment. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector	$v + v1$	Addition. *Precondition*: $v.dim(\ ) \equiv v1.dim(\ )$.
integer_vector	$v - v1$	Subtraction. *Precondition*: $v.dim(\ ) = v1.dim(\ )$.
integer	$v * v1$	Inner Product. *Precondition*: $v.dim(\ ) = v1.dim(\ )$.
integer_vector	$integer\ r * v$	Componentwise multiplication with number r.
integer_vector	$v * integer\ r$	Componentwise multiplication with number r.
bool	$v \equiv w$	Test for equality.

bool	v != w	Test for inequality.
ostream&	*ostream&* $O \ll v$	writes v componentwise to the output stream O.
istream&	*istream&* $I \gg$ *integer_vector&* v	reads v componentwise from the input stream I.

4. Implementation

Vectors are implemented by arrays of integers. All operations on a vector v take time $O(v.dim(\,))$, except for *dim* and [ ] which take constant time. The space requirement is $O(v.dim(\,))$.

2. The Header File of class integer_vector

```
⟨ integer_vector.h 2 ⟩ ≡
#ifndef LEDA_INTEGER_VECTOR_H
#define LEDA_INTEGER_VECTOR_H
#include <math.h>
#include <LEDA/basic.h>
#include <LEDA/integer.h>
#include <LEDA/rational.h>
#define MAX_SIZE_OF_SMALL_OBJECT 256
  typedef integer integer_t;
  typedef rational rational_t;
  class integer_vector {
    friend class integer_matrix;
    integer_t *v;
    int d;
    void check_dimensions(const integer_vector &) const;
    void allocate_small(integer_t *&v, int d);
    void deallocate_small(integer_t *v, int d);
    static int LEDA_SMALL;
  public:
    integer_vector( );
    integer_vector(int d);
    integer_vector(integer_t a, integer_t b);
    integer_vector(integer_t a, integer_t b, integer_t c);
    integer_vector(const integer_vector &);
    ~integer_vector( );
    integer_vector &operator=(const integer_vector &);
    int dim( ) const { return d; }
    integer_t &operator [](int i)
    {
#ifndef LEDA_CHECKING_OFF
      if (i < 0 ∨ i ≥ d)
        error_handler(1, "integer_vector::operator[]: index out of range.");
#endif
      return v[i];
    }
    integer_t operator [](int i) const
    {
#ifndef LEDA_CHECKING_OFF
      if (i < 0 ∨ i ≥ d)
        error_handler(1, "integer_vector::operator[]: index out of range.");
#endif
      return v[i];
    }
    integer_vector &operator+=(const integer_vector &v1);
    integer_vector &operator-=(const integer_vector &v1);
    integer_vector operator+(const integer_vector &v1) const;
    integer_vector operator-(const integer_vector &v1) const;
```

```
  integer_t operator*(const integer_vector &v1) const;
  integer_vector compmul(integer_t r) const;
  friend integer_vector operator*(integer_t r, const integer_vector &v);
  friend integer_vector operator*(const integer_vector &v, integer_t r);
  integer_vector operator-() const;
  bool operator≡(const integer_vector &w) const;
  bool operator≠(const integer_vector &w) const { return ¬(*this ≡ w); }
  friend ostream &operator≪(ostream &O, const integer_vector &v);
  friend istream &operator≫(istream &I, integer_vector &v);
  static int cmp(const integer_vector &, const integer_vector &);
  LEDA_MEMORY(integer_vector)
};
inline int compare(const integer_vector &x, const integer_vector &y)
{ return integer_vector::cmp(x, y); }
inline void Print(const integer_vector &v, ostream &out) { out ≪ v; }
inline void Read(integer_vector &v, istream &in) { in ≫ v; }
LEDA_TYPE_PARAMETER(integer_vector)
#endif
```

3. The Implementation of class integer_vector

We use the LEDA constant `MAX_SIZE_OF_SMALL_OBJECT` to implement a two level memory allocation scheme dependent on the integer size.

```
⟨ integer_vector.c 3 ⟩ ≡
#include "integer_vector.h"
  int integer_vector::LEDA_SMALL = MAX_SIZE_OF_SMALL_OBJECT/sizeof(integer_t);
  ⟨ constructors, destructors, and assignment 4 ⟩;
  ⟨ other functions 5 ⟩;
```

4. Constructors, Destructors, and Assignment.

```
⟨ constructors, destructors, and assignment 4 ⟩ ≡
  void integer_vector::allocate_small(integer_t *&v, int d)
  /* we use this procedure to allocate memory for small arrays. We first get an appropriate
  piece of memory from the LEDA memory manager and then initialize each cell by an inplace
  new. */
  {
    v = (integer_t *) allocate_bytes(d * sizeof(integer_t));
    integer_t *p = v + d - 1;
    while (p ≥ v) { new (p) integer_t; p--; }
  }
  void integer_vector::deallocate_small(integer_t *v, int d)
  /* we use this procedure to deallocate memory for small arrays. We first call the destructor
  for type integer_t for each cell of the array and then return the piece of memory to the LEDA
  memory manager. */
  {
    integer_t *p = v + d - 1;
    while (p ≥ v) { p->~integer_t(); p--; }
    deallocate_bytes(v, d * sizeof(integer_t));
  }
  integer_vector::integer_vector()
  { d = 0; v = nil; }
  integer_vector::integer_vector(int n)
  {
    if (n < 0) error_handler(1, "integer_vector::constructor: negative dimension.");
    d = n;
    v = nil;
    if (d > 0) {
      if (d < LEDA_SMALL) integer_vector::allocate_small(v, d);
      else v = new integer_t [d];
      integer_t zero = 0;
      while (n--) v[n] = zero;
    }
  }
  integer_vector::~integer_vector()
  {
    if (v)
      if (d < LEDA_SMALL) integer_vector::deallocate_small(v, d);
```

```
    else delete []v;
  }
  integer_vector::integer_vector(const integer_vector &p)
  {
    d = p.d;
    v = nil;
    if (d > 0) {
      if (d < LEDA_SMALL) integer_vector::allocate_small(v, d);
      else v = new integer_t [d];
      for (int i = 0; i < d; i++) v[i] = p.v[i];
    }
  }
  integer_vector &integer_vector::operator=(const integer_vector &vec)
  {
    register int n = vec.d;
    if (n ≠ d) {
      if (v) {
        if (d < LEDA_SMALL) integer_vector::deallocate_small(v, d);
        else delete []v;
      }
      if (n < LEDA_SMALL) integer_vector::allocate_small(v, n);
      else v = new integer_t [n];
      d = n;
    }
    while (n--) v[n] = vec.v[n];
    return *this;
  }
  integer_vector::integer_vector(integer_t x, integer_t y)
  {
    integer_vector::allocate_small(v, 2);
    d = 2;
    v[0] = x;
    v[1] = y;
  }
  integer_vector::integer_vector(integer_t x, integer_t y, integer_t z)
  {
    integer_vector::allocate_small(v, 3);
    d = 3;
    v[0] = x;
    v[1] = y;
    v[2] = z;
  }
```

This code is used in chunk 3.

5. Other Functions.

⟨ other functions 5 ⟩ ≡

```
  void integer_vector::check_dimensions(const integer_vector &vec) const
  {
```

```
  if (d ≠ vec.d)
    error_handler(1,"integer_vector::check_dimensions: different dimensions.");
}
integer_vector &integer_vector::operator+=(const integer_vector &vec)
{
  check_dimensions(vec);
  register int n = d;
  while (n--) v[n] += vec.v[n];
  return *this;
}
integer_vector &integer_vector::operator-=(const integer_vector &vec)
{
  check_dimensions(vec);
  register int n = d;
  while (n--) v[n] -= vec.v[n];
  return *this;
}
integer_vector integer_vector::operator+(const integer_vector &vec) const
{
  check_dimensions(vec);
  register int n = d;
  integer_vector result(n);
  while (n--) result.v[n] = v[n] + vec.v[n];
  return result;
}
integer_vector integer_vector::operator-(const integer_vector &vec) const
{
  check_dimensions(vec);
  register int n = d;
  integer_vector result(n);
  while (n--) result.v[n] = v[n] - vec.v[n];
  return result;
}
integer_vector integer_vector::operator-() const  // unary minus
{
  register int n = d;
  integer_vector result(n);
  while (n--) result.v[n] = -v[n];
  return result;
}
integer_vector integer_vector::compmul(integer_t x) const
{
  int n = d;
  integer_vector result(n);
  while (n--) result.v[n] = v[n] * x;
  return result;
}
```

```
integer_vector operator ∗ (integer_t f, const integer_vector &v)
{ return v.compmul(f); }
integer_vector operator ∗ (const integer_vector &v, integer_t f)
{ return v.compmul(f); }
integer_t integer_vector::operator ∗ (const integer_vector &vec) const
{
  check_dimensions(vec);
  integer_t result = 0;
  register int n = d;
  while (n−−) result = result + v[n] ∗ vec.v[n];
  return result;
}
bool integer_vector::operator≡(const integer_vector &vec) const
{
  if (vec.d ≠ d) return false;
  int i = 0;
  while ((i < d) ∧ (v[i] ≡ vec.v[i])) i++;
  return (i ≡ d);
}
ostream &operator≪(ostream &s, const integer_vector &v)
{
  for (int i = 0; i < v.d; i++) s ≪ v[i] ≪ "␣";
  return s;
}
istream &operator≫(istream &s, integer_vector &x)
{
  int i = 0;
  while (i < x.d ∧ s ≫ x.v[i++]) ;
  return s;
}
int integer_vector::cmp(const integer_vector &v1, const integer_vector &v2)
{
  register int i;
  v1.check_dimensions(v2);
  for (i = 0; i < v1.dim() ∧ v1[i] ≡ v2[i]; i++) ;
  if (i ≡ v1.dim()) return 0;
  return (v1[i] < v2[i]) ? −1 : 1;
}
```

This code is used in chunk 3.

6. A Test of class integer_vector

And finally we test our vector type in a little program. We first test construction and access operations. Then we do some basic calculations.

```
⟨ integer_vector-test.c 6 ⟩ ≡
#include "integer_vector.h"
#define VEC_DIM 17
  main( )
  { /* some construction and access ops */
    integer_vector v1(VEC_DIM), v2(VEC_DIM);
    for (int i = 0; i < VEC_DIM; i++) {
      v1[i] = i;
      v2[i] = VEC_DIM − i;
    }
    integer_vector v3(v1);
    /* some dimension and equality testing */
    cout ≪ "three␣vectors␣v1,v2,v3:\n";
    cout ≪ v1 ≪ "\n";
    cout ≪ v2 ≪ "\n";
    cout ≪ v3 ≪ "\n";
    cout ≪ "with␣dimension␣" ≪ v1.dim( ) ≪ "\n";
    cout ≪ "v1␣" ≪ (v1 ≡ v2 ? "==" : "!=") ≪ "␣v2\n";
    cout ≪ "v1␣" ≪ (v1 ≡ v3 ? "==" : "!=") ≪ "␣v3\n";
    /* some arithmetic testing */
    v1 += 2 ∗ v2;
    v1 −= v2;
    v1 = −v1;
    cout ≪ "squared␣length␣of␣(v1␣+␣v2)␣=␣" ≪ v1 ∗ v1 ≪ "\n";
    cout ≪ "should␣be␣dim*dim*dim␣=␣" ≪ VEC_DIM ∗ VEC_DIM ∗ VEC_DIM ≪ "\n";
    cout ≪ "v1*v1␣-␣v2*v2:␣" ≪ (v1 ∗ v1 − v2 ∗ v2) ≪ "\n";
    cout ≪ "(v1+v2)(v1-v2):␣" ≪ ((v1 − v2) ∗ (v1 + v2)) ≪ "\n\n";
  }
```

Index

List of Refinements

⟨ constructors, destructors, and assignment 4 ⟩ Used in chunk 3.
⟨ `integer_vector-test.c` 6 ⟩
⟨ `integer_vector.c` 3 ⟩
⟨ `integer_vector.h` 2 ⟩
⟨ other functions 5 ⟩ Used in chunk 3.

Points with Rational Coordinates in d-Space (class rat_point)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_point

1. Definition

An instance of data type *rat_point* is a point with rational coordinates in an arbitrary dimensional space. A point $p = (p_0, \ldots, p_{d-1})$ in d-dimensional space is represented by homogeneous coordinates $(h_0, h_1, \ldots, h_d)$ of arbitrary length integers such that $p_i = h_i/h_d$. The homogenizing coordinate h_d is positive.

We call p_i, $0 \leq i < d$ the i-th cartesian coordinate and h_i, $0 \leq i \leq d$, the i-th homogeneous coordinate. We call d the dimension of the point.

The default ordering is the lexicographic ordering of the cartesian coordinate tuples.

rat_point is an item type.

2. Creation

rat_point *p*(*int d* = 2); introduces a variable p of type *rat_point* in d-dimensional space.

rat_point *p*(*integer a*, *integer b*, *integer D* = 1);
introduces a variable p of type *rat_point* initialized to the two-dimensional point with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_point *p*(*integer_vector c*, *integer D*);
introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_point *p*(*integer_vector c*); introduces a variable p of type *rat_point* initialized to the point with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_point *rat_point*::d2(*integer a*, *integer b*, *integer D* = 1)
returns a *rat_point* of dimension 2 initialized to a point with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_point	*rat_point*::d3(*integer a*, *integer b*, *integer c*, *integer D* = 1)	returns a *rat_point* of dimension 3 initialized to a point with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative. *Precondition*: D is non-zero.
rat_point	*rat_point*::origin(*int d* = 2)	returns the origin in d-dimensional space.
int	*p*.dim( )	returns the dimension of p.
rational	*p*.coord(*int i*)	returns the i-th cartesian coordinate of p.
rational	*p*[*int i*]	returns the i-th cartesian coordinate of p.
integer	*p*.hcoord(*int i*)	returns the i-th homogeneous coordinate of p.
rat_point	*p*.transform(*aff_transformation t*)	returns $t(p)$.
rat_vector	*p*.to_rat_vector( )	converts p to an vector.
rat_vector	to_rat_vector(*rat_point p*)	converts p to an vector.
rat_direction	*p*.to_rat_direction( )	converts p to a direction. *Precondition*: p is different form the origin.
rat_direction	to_rat_direction(*rat_point p*)	converts p to a direction. *Precondition*: p is different form the origin.

Additional Operations for points in two-dimensional space

rational	*p*.xcoord( )	returns the zeroth cartesian coordinate of p.
rational	*p*.ycoord( )	returns the first cartesian coordinate of p.
integer	*p*.X( )	returns the zeroth homogeneous coordinate of p.
integer	*p*.Y( )	returns the first homogeneous coordinate of p.
integer	*p*.W( )	returns the homogenizing coordinate of p.
rat_point	*p*.rotate90(*rat_point q*)	returns p rotated counterclockwise by 90 degrees about q.
rat_point	*p*.rotate90( )	returns p rotated counterclockwise by 90 degrees about the origin.

3.2 Tests

bool p.is_origin( ) returns true if p is equal to the origin.

bool identical(*rat_point* p, *rat_point* q)
test for identity

bool $p \equiv q$ test for equality.

bool p != q test for inequality.

3.3 Arithmetical Operators

rat_vector $p - q$ returns $p - q$.
Precondition: $p.dim( ) \equiv q.dim( )$.

rat_point $p +$ *rat_vector* v returns $p + v$.
Precondition: $p.dim( ) \equiv v.dim( )$.

rat_point p.translate(*rat_vector* v)
returns returns $p + v$.
Precondition: $p.dim( ) \equiv v.dim( )$.

rat_point& $p+=$ *rat_vector* v adds v to p.
Precondition: $p.dim( ) \equiv v.dim( )$.

rat_point $p -$ *rat_vector* v returns $p - v$.
Precondition: $p.dim( ) \equiv v.dim( )$.

rat_point& $p-=$ *rat_vector* v subtracts v from p.
Precondition: $p.dim( ) \equiv v.dim( )$.

3.4 Input and Output

ostream& *ostream&* $O \ll p$ writes the homogeneous coordinates of p to output stream O.

istream& *istream&* $I \gg$ *rat_point&* p
reads the homogeneous coordinates of p from input stream I. This operator uses the current dimension of p.

3.5 Position Tests

int orientation(*array<rat_point>A*)
determines the orientation of the points in A, where A consists of $d + 1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the i-th point in A.

int side_of_oriented_sphere(*array<rat_point>A*, *rat_point x*)

determines whether the point x lies inside ($= -1$), on ($= 0$), or outside ($= +1$) the *oriented* sphere defined by the points in A, where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ lift(A[0]) & lift(A[1]) & \dots & lift(A[d]) & lift(x) \end{vmatrix}$$

where for a point p with cartesian coordinates p_i we use $lift(p)$ to denote the $d+1$-dimensional point with cartesian coordinate vector $(p_0, \dots, p_{d-1}, \sum_{0 \le i < d} p_i^2)$.

int side_of_sphere(*array<rat_point>A*, *rat_point x*)

determines whether the point x lies inside ($= -1$), on ($= 0$), or outside ($= +1$) the sphere defined by the points in A, where A consists of $d+1$ points in d-space. (equivalent to $orientation(A) * side\_of\_oriented\_sphere(A, x)$)
Precondition: $orientation(A) \neq 0$

bool contained_in_simplex(*array<rat_point>A*, *rat_point x*)

determines whether x is contained in the simplex spanned by the points in A. A may consists of up to $d+1$ points.
Precondition: The points in A are affinely independent.

3.6 Affine Hull, Dependence and Rank

bool contained_in_affine_hull(*array<rat_point>A*, *rat_point x*)

determines whether x is contained in the affine hull of the points in A.

int affine_rank(*array<rat_point>A*)

computes the affine rank of the points in A.

bool affinely_independent(*array<rat_point>A*)

decides whether the points in A are affinely independent.

Additional Operations for points in two-dimensional space

rational area(*rat_point a*, *rat_point b*, *rat_point c*)

computes the signed area of the triangle determined by a,b,c, positive if $orientation(a, b, c) > 0$ and negative otherwise.

int orientation(*rat_point a*, *rat_point b*, *rat_point c*)

computes the orientation of points a, b, c, i.e., the sign of the determinant

$$\begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ c_w & c_x & c_y \end{vmatrix}$$

bool collinear(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and false otherwise.

bool right_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a right turn, i.e., $orientation(a, b, c) > 0$, and false otherwise.

bool left_turn(*rat_point a*, *rat_point b*, *rat_point c*)

returns true if points a, b, c form a left turn, i.e., $orientation(a, b, c) < 0$, and false otherwise.

int side_of_oriented_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $+1$ if point d lies left of the directed circle through points a, b, and c, 0 if a,b,c,and d are cocircular, and -1 otherwise. If a, b, and c are collinear the directed circle is a line oriented from a to b if c is not part of the connecting segment $\overline{ab}$, or else oriented from b to c.

int side_of_circle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns $+1$ if point d lies inside of, 0 if on and -1 if outside of the circle through points a, b, and c,
Precondition: a, b, c are not collinear

bool cocircular(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if points a, b, c, d are cocircular, i.e., $side_of_oriented_circle(a, b, c) = 0$, and false otherwise.

bool incircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if point d lies in the interior of the circle through the points a, b, and c, and false otherwise.

bool outcircle(*rat_point a*, *rat_point b*, *rat_point c*, *rat_point d*)

returns true if point d lies outside the circle through the points a, b, and c, and false otherwise.

4. Implementation

Points are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, point-vector arithmetic, input and output on a point p take time $O(p.dim(\,))$. $dim(\,)$, coordinate access and conversions take constant time. The operations for affine calculation and determinant evaluation have the cubic costs of the used matrix operations. The space requirement is $O(p.dim(\,))$.

2. The Header File of class rat_point

The type rat_point is an item class with representation class geo_rep. It shares this representation class with hyperplanes, vectors, and directions. We derive rat_point from handle_base and derive geo_rep from handle_rep. This gives us reference counting for free. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_point.h 2 ⟩ ≡
#ifndef LEDA_RAT_POINT_H
#define LEDA_RAT_POINT_H
#include <iostream.h>
#include <math.h>
#include <ctype.h>
#include <LEDA/array.h>
#include "integer_vector.h"
#include "integer_matrix.h"
#include "geo_rep.h"
#include "rat_direction.h"
#include "rat_vector.h"
  class aff_transformation;
  class rat_point : public handle_base {
  geo_rep *ptr() const { return (geo_rep *) PTR; }
  rat_point(const handle_base &p) : handle_base(p) {}
  friend class rat_vector;
  friend class rat_direction;
public:
  rat_point(int d = 2) { PTR = new geo_rep (d); ptr()->v[d] = 1; }
  rat_point(integer_t a, integer_t b, integer_t D = 1)
  {
    if (D ≡ 0) error_handler(1, "rat_point::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (-a, -b, -D);
    else PTR = new geo_rep (a, b, D);
  }
  rat_point(int a, int b, int D = 1)
  {
    if (D ≡ 0) error_handler(1, "rat_point::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (-a, -b, -D);
    else PTR = new geo_rep (a, b, D);
  }
  rat_point(const integer_vector &c, integer_t D)
  {
    if (D ≡ 0) error_handler(1, "rat_point::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (-c, -D);
    else PTR = new geo_rep (c, D);
  }
  rat_point(const integer_vector &c)
  {
    int d = c.dim();
    integer_t D = c[d - 1];
```

```
  if (D ≡ 0) error_handler(1,"rat_point::constructor:␣D␣must␣be␣nonzero.");
  if (D < 0) PTR = new geo_rep (−c);
  else PTR = new geo_rep (c);
}
rat_point(const rat_point &p) : handle_base(p) { }
~rat_point( ) { }
rat_point &operator=(const rat_point &p)
{ handle_base::operator=(p); return *this; }
static rat_point d2(integer_t a, integer_t b, integer_t D = 1);
static rat_point d3(integer_t a, integer_t b, integer_t c, integer_t D = 1);
static rat_point origin(int d = 2);
int dim( ) const { return ptr( )→dim; }
rational_t coord(int i) const { return rational_t(ptr( )→v[i], ptr( )→v[ptr( )→dim]); }
rational_t operator [](int i) const { return coord(i); }
integer_t hcoord(int i) const { return ptr( )→v[i]; }
rat_point transform(const aff_transformation &t) const;
rat_vector to_rat_vector( ) const;
friend rat_vector to_rat_vector(const rat_point &p);
rat_direction to_rat_direction( ) const;
friend rat_direction to_rat_direction(const rat_point &p);
rational_t xcoord( ) const { return rational_t(hcoord(0), hcoord(ptr( )→dim)); }
rational_t ycoord( ) const { return rational_t(hcoord(1), hcoord(ptr( )→dim)); }
integer_t X( ) const { return hcoord(0); }
integer_t Y( ) const { return hcoord(1); }
integer_t W( ) const { return hcoord(ptr( )→dim); }
rat_point rotate90(const rat_point &q) const;
rat_point rotate90( ) const { return rat_point(−Y( ), X( ), W( )); }
bool is_origin( ) const
{
  for (int i = 0; i < dim( ); i++)
    if (hcoord(i) ≠ 0) return false;
  return true;
}
friend bool identical(const rat_point &p, const rat_point &q)
{ return p.ptr( ) ≡ q.ptr( ); }
int cmp(const rat_point &p, const rat_point &q) const
{ return (identical(p, q) ? 0 :
      p.ptr( )→cmp_rat_coords(p.ptr( ), q.ptr( ))); }
friend int compare(const rat_point &p, const rat_point &q)
{ return p.cmp(p, q); }
friend bool operator≡(const rat_point &p, const rat_point &q)
{ return p.cmp(p, q) ≡ 0; }
friend bool operator≠(const rat_point &p, const rat_point &q)
{ return p.cmp(p, q) ≠ 0; }
rat_vector operator−(const rat_point &q) const;
rat_point operator+(const rat_vector &v) const;
```

```
  rat_point translate(const rat_vector &v) const
  { return operator+(v); }
  rat_point &operator+=(const rat_vector &v);
  rat_point operator-(const rat_vector &v) const;
  rat_point &operator-=(const rat_vector &v);
  friend ostream &operator≪(ostream &O, const rat_point &p);
  friend istream &operator≫(istream &I, rat_point &p); } ;
  inline void Print(const rat_point &p, ostream &out) { out ≪ p; }
  inline void Read(rat_point &p, istream &in) { in ≫ p; }
  int orientation(const array⟨rat_point⟩ &A);
  int side_of_oriented_sphere(const array⟨rat_point⟩ &A, const rat_point &x);
  int side_of_sphere(const array⟨rat_point⟩ &A, const rat_point &x);
  bool contained_in_simplex(const array⟨rat_point⟩ &A, const rat_point &x);
  bool contained_in_affine_hull(const array⟨rat_point⟩ &A, const rat_point &x);
  int affine_rank(const array⟨rat_point⟩ &A);
  bool affinely_independent(const array⟨rat_point⟩ &A);
  rational_t area(const rat_point &a, const rat_point &b, const rat_point &c);
  int orientation(const rat_point &a, const rat_point &b, const rat_point &c);
  inline bool collinear(const rat_point &a, const rat_point &b, const rat_point &c)
  { return orientation(a, b, c) ≡ 0; }
  inline bool right_turn(const rat_point &a, const rat_point &b, const rat_point &c)
  { return orientation(a, b, c) < 0; }
  inline bool left_turn(const rat_point &a, const rat_point &b, const rat_point &c)
  { return orientation(a, b, c) > 0; }
  int side_of_oriented_circle(const rat_point &a, const rat_point &b, const rat_point
      &c, const rat_point &d);
  inline int side_of_circle(const rat_point &a, const rat_point &b, const rat_point
        &c, const rat_point &d)
  { return (orientation(a, b, c) * side_of_oriented_circle(a, b, c, d)); }
  inline bool cocircular(const rat_point &a, const rat_point &b, const rat_point
        &c, const rat_point &d)
  { return (side_of_oriented_circle(a, b, c, d) ≡ 0); }
  inline bool incircle(const rat_point &a, const rat_point &b, const rat_point &c, const
        rat_point &d)
  { return (orientation(a, b, c) * side_of_oriented_circle(a, b, c, d) > 0); }
  inline bool outcircle(const rat_point &a, const rat_point &b, const rat_point
        &c, const rat_point &d)
  { return (orientation(a, b, c) * side_of_oriented_circle(a, b, c, d) < 0); }
#endif
```

3. The Implementation of class rat_point

```
⟨ rat_point.c 3 ⟩ ≡
#include "rat_point.h"
  ⟨ initialization 4 ⟩
  ⟨ conversions 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ arithmetic operations 7 ⟩
  ⟨ position tests 11 ⟩
  ⟨ affine operations 15 ⟩
  ⟨ special operations of 2-space 18 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_point rat_point::d2(integer_t a, integer_t b, integer_t D)
  {
    if (D ≡ 0) error_handler(1, "rat_point::d2: denominator must not be zero.");
    return rat_point(a, b, D);
  }
  rat_point rat_point::d3(integer_t a, integer_t b, integer_t c, integer_t D)
  {
    rat_point d3p(3);
    if (D ≡ 0) error_handler(1, "rat_point::d3: denominator must not be zero.");
    if (D < 0) d3p.ptr()→init4(−a, −b, −c, −D);
    else d3p.ptr()→init4(a, b, c, D);
    return d3p;
  }
  rat_point rat_point::origin(int d = 2) { return rat_point(d); }
```

This code is used in chunk 3.

5. Conversion

Points can be converted to directions and vectors. Since the converted object has the same representation, conversion amounts to a call of the copy constructor of the base class.

```
⟨ conversions 5 ⟩ ≡
  rat_direction rat_point::to_rat_direction() const
  {
    if (is_origin()) error_handler(1,
          "rat_point::to_rat_direction: origin cannot be a direction.");
    return rat_direction(*this);
  }
  rat_vector rat_point::to_rat_vector() const
  { return rat_vector(*this); }
  rat_vector to_rat_vector(const rat_point &p)
  { return p.to_rat_vector(); }
  rat_direction to_rat_direction(const rat_point &p)
  { return p.to_rat_direction(); }
```

This code is used in chunk 3.

6. Input and Output

We define the operators $\ll$ and $\gg$. The output operator reduces to the output operator for geo_reps. The input operator additionally has to take care that is does not overwrite a commonly used representation object. Moreover it has to secure our invariant that the denominator is neither zero nor negative.

⟨ input and output 6 ⟩ ≡

```
ostream &operator≪(ostream &out, const rat_point &p)
{
  out ≪ p.ptr( );
  return out;
}
istream &operator≫(istream &in, rat_point &p)
{
  int d = p.dim( );
  if (p.refs( ) > 1) p = rat_point(d);
  in ≫ p.ptr( );
  if (p.hcoord(d) ≡ 0)
    error_handler(1, "operator>>: denominator of point must be nonzero.");
  if (p.hcoord(d) < 0) p.ptr( )→negate(d + 1);
  return in;
}
```

This code is used in chunk 3.

7. Arithmetical Operators

The operators + and − do the obvious. They construct a new rat_point and fill it with the appropriate values.

⟨ arithmetic operations 7 ⟩ ≡

```
rat_point rat_point::operator+(const rat_vector &v) const
{
  rat_point res(dim( ));
  c_add(res.ptr( ), ptr( ), v.ptr( ));
  return res;
}
rat_point rat_point::operator−(const rat_vector &v) const
{
  rat_point res(dim( ));
  c_sub(res.ptr( ), ptr( ), v.ptr( ));
  return res;
}
rat_vector rat_point::operator−(const rat_point &q) const
{
  rat_vector res(dim( ));
  c_sub(res.ptr( ), ptr( ), q.ptr( ));
  return res;
}
```

See also chunks 8 and 9.

This code is used in chunk 3.

8. The operators += and −= are almost implemented the same. However, they avoid the construction of a new geo_rep if the left hand side was sole owner of its geo_rep. We establish a pointer *old* to the old geo_rep, construct a new geo_rep if necessary, and then fill the new geo_rep. Be warned, the old and the new geo_rep might be identical and hence care is necessary when this code is adopted to operators that are less local than + and −.

```
⟨ arithmetic operations 7 ⟩ +≡
  rat_point &rat_point::operator+=(const rat_vector &v)
  {
    int d = dim();
    rat_point old(*this);
    if (ptr()→count > 2) *this = rat_point(d);
    c_add(ptr(), old.ptr(), v.ptr());
    return *this;
  }
  rat_point &rat_point::operator−=(const rat_vector &v)
  {
    int d = dim();
    rat_point old(*this);
    if (ptr()→count > 2) *this = rat_point(d);
    c_sub(ptr(), old.ptr(), v.ptr());
    return *this;
  }
```

9. The transformation of a point is just implemented by a matrix multiplication of the transformation matrix and the vector which represents the point.

```
⟨ arithmetic operations 7 ⟩ +≡
  rat_point rat_point::transform(const aff_transformation &t) const
  { return rat_point(t.matrix() * ptr()→ivec()); }
```

10. Linear Algebra

11. Orientation. We are given a array A of $d+1$ points in d-space and compute their orientation. This amounts to computing the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the i-th point in A. Multiplying the i-th column by the homogenizing coordinate of $A[i]$ leaves the sign of the determinant unchanged. We set up this matrix and return its determinant. Actually, it is more convenient to transpose it and to make the first row the last. This changes the sign if the number of rows is even, i.e., if d is odd.

```
⟨ position tests 11 ⟩ ≡
  int orientation(const array⟨rat_point⟩ &A)
  {
    int al = A.low(); // the lower index start of A
    int d = A.high() − al; // A contains d+1 points
    if (d ≠ A[al].dim())
      error_handler(1, "orientation: needs A[].dim()+1 many input points.");
```

```
    integer_matrix M(d+1, d+1);
    for (int i = 0; i ≤ d; i++)
      for (int j = 0; j ≤ d; j++) M(i, j) = A[al + i].hcoord(j);
    integer_t det = determinant(M);
    int s = (d % 2 ≡ 0 ? 1 : −1);
    if (det > 0) return s;
    if (det < 0) return −s;
    return 0;
  }
```

See also chunks 12, 13, and 14.

This code is used in chunk 3.

12. Side of sphere. We have tests which test a point in relation to a sphere. The first one tests the location of a point x with respect to the oriented sphere defined by $d+1$ points in d-space, and the second test determines the location with respect to the sphere.

The first test amounts to evaluate the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ lift(A[0]) & lift(A[1]) & \dots & lift(A[d]) & lift(x) \end{vmatrix}$$

where for a point p with cartesian coordinates p_i we use $lift(p)$ to denote the $d+1$-dimensional point with cartesian coordinate vector $(p_0, \dots, p_{d-1}, \sum_{0 \le i < d} p_i^2)$. The lifting map *lift* maps a point p onto the paraboloid of revolution and the position of the point x with respect to the oriented sphere defined by the points in A is the same as the orientation of the lifted points. In order to evaluate the determinant we multiply each column by the square of the homogenizing coordinate. This turns any column into $(h_d^2, h_0 h_d, ..., h_{d-1} h_d, \sum_{0 \le i < d} h_i^2)$, where the h_i's denote homogeneous coordinates. Thus we set up this matrix and return its sign (it is actually simpler to set up the transposed matrix and so that's what we do).

```
⟨ position tests 11 ⟩ +≡
  int side_of_oriented_sphere(const array⟨rat_point⟩ &A, const rat_point &x)
  {
    int al = A.low( );  // the lower index start of A
    int d = A.high( ) − al;  // A contains d + 1 points
    if (d ≠ A[al].dim( )) error_handler(1,
          "side_of_oriented_sphere: needs A[].dim()+1 many input points.");
    integer_matrix M(d + 2, d + 2);
    for (int i = 0; i ≤ d; i++) {
      integer_t Sum = 0;
      integer_t hd = A[al + i].hcoord(d);
      M(i, 0) = hd * hd;
      for (int j = 0; j < d; j++) {
        integer_t hj = A[al + i].hcoord(j);
        M(i, j + 1) = hj * hd;
        Sum += hj * hj;
      }
      M(i, d + 1) = Sum;
    }
```

```
    integer_t Sum = 0;
    integer_t hd = x.hcoord(d);
    M(d+1,0) = hd * hd;
    for (int j = 0; j < d; j++) {
      integer_t hj = x.hcoord(j);
      M(d+1,j+1) = hj * hd;
      Sum += hj * hj;
    }
    M(d+1,d+1) = Sum;
    integer_t det = determinant(M);
    if (det > 0) return 1;
    if (det < 0) return -1;
    return 0;
  }
```

13. The side_of_sphere function first checks its precondition and then returns the value stated in the specification.

```
⟨ position tests 11 ⟩ +≡
  int side_of_sphere(const array⟨rat_point⟩ &A, const rat_point &x)
  {
    int s = orientation(A);
    if (s ≡ 0) error_handler(1, "side_of_sphere: A must be full dimensional.");
    return s * side_of_oriented_sphere(A, x);
  }
```

14. Containment in Simplex.

A point x is contained in the convex hull of a set A of points if x is a convex combination of the points in A, i.e., if the system $\sum \lambda_i A_i = x$ has a solution with $\sum \lambda_i = 1$ and $\lambda_i \geq 0$. If the points in A are linearly independent then the solution is unique (if it exists at all). We therefore proceed as above and then check whether the solution vector is non-negative.

```
⟨ position tests 11 ⟩ +≡
  bool contained_in_simplex(const array⟨rat_point⟩ &A, const rat_point &x)
  {
    int al = A.low( );  // the lower index start of A
    int k = A.high( ) - al + 1;  // A contains k points
    int d = A[al].dim( );
    integer_matrix M(d+1,k);
    integer_vector b(d+1);
    for (int i = 0; i ≤ d; i++) {
      b[i] = x.hcoord(i);
      for (int j = 0; j < k; j++) M(i,j) = A[al+j].hcoord(i);
    }
    integer_t D;
    integer_vector lambda;
    if (linear_solver(M, b, lambda, D)) {
      int s = sign(D);
```

```
    for (int j = 0; j < k; j++) {
      int t = sign(lambda[j]);
      if (s * t < 0) return false;
    }
  }
  return true;
}
```

15. Containment in Affine Hull.

A point x is contained in the affine hull of a set A of points if x is an affine combination of the points in A, i.e., if the system $\sum \lambda_i A_i = x$ has a solution with $\sum \lambda_i = 1$. Set $\lambda_i = A_{i,d}\beta_i/x_d$ with $A_{i,d}$ being the homogenizing coordinate of A_i and x_d being the homogenizing coordinate of x. The i-th column of the system for the β_i's is simply the homogeneous vector of A_i and the right hand side is simply the homogeneous vector for x. Thus we proceed as above but let i run up to d.

⟨ affine operations 15 ⟩ ≡

```
bool contained_in_affine_hull(const array⟨rat_point⟩ &A, const rat_point &x)
{
  int al = A.low( ); // the lower index start of A
  int k = A.high( ) − al + 1; // A contains k points
  int d = A[al].dim( );
  integer_matrix M(d + 1, k);
  integer_vector b(d + 1);
  for (int i = 0; i ≤ d; i++) {
    b[i] = x.hcoord(i);
    for (int j = 0; j < k; j++) M(i, j) = A[al + j].hcoord(i);
  }
  return solvable(M, b);
}
```

See also chunks 16 and 17.

This code is used in chunk 3.

16. Affine Rank.

The affine rank of points $p_0, p_1, \ldots, p_k$ is the linear rank of points $p_0 - p_k, \ldots, p_{k-1} - p_k$.

⟨ affine operations 15 ⟩ +≡

```
int affine_rank(const array⟨rat_point⟩ &A)
{
  int al = A.low( ); // the lower index start of A
  int k = A.high( ) − al; // A contains k + 1 points
  if (k < 0) return −1;
  if (k ≡ 0) return 0;
  int d = A[al].dim( );
  integer_matrix M(d, k);
  for (int j = 0; j < k; j++) {
    rat_point p = (A[al + j] − A[al + k]).to_rat_point( );
```

```
    for (int i = 0; i < d; i++) M(i, j) = p.hcoord(i);
  }
  return rank(M);
}
```

17. Affine Independence.

A set of points is affinely independent if their affine rank is equal to the number of points minus 1.

```
⟨ affine operations 15 ⟩ +≡
  bool affinely_independent(const array⟨rat_point⟩ &A)
  { return (affine_rank(A) ≡ (A.high( ) − A.low( ))); }
```

18. Special Operations in 2-Space

```
⟨ special operations of 2-space 18 ⟩ ≡
  int side_of_oriented_circle(const rat_point &a, const rat_point &b, const rat_point
        &c, const rat_point &d)
  { /* copied from the LEDA 2d module _rat_point.c */
    integer_t AX = a.X( );
    integer_t AY = a.Y( );
    integer_t AW = a.W( );
    integer_t BX = b.X( );
    integer_t BY = b.Y( );
    integer_t BW = b.W( );
    integer_t CX = c.X( );
    integer_t CY = c.Y( );
    integer_t CW = c.W( );
    integer_t DX = d.X( );
    integer_t DY = d.Y( );
    integer_t DW = d.W( );
    integer_t bx, by, bw, cx, cy, cw, dx, dy, dw;
    if (AW ≡ 1 ∧ BW ≡ 1 ∧ CW ≡ 1 ∧ DW ≡ 1) {
      bx = BX − AX;
      by = BY − AY;
      bw = bx ∗ bx + by ∗ by;
      cx = CX − AX;
      cy = CY − AY;
      cw = cx ∗ cx + cy ∗ cy;
      dx = DX − AX;
      dy = DY − AY;
      dw = dx ∗ dx + dy ∗ dy;
    }
    else {
      integer_t b1 = BX ∗ AW − AX ∗ BW;
      integer_t b2 = BY ∗ AW − AY ∗ BW;
      integer_t c1 = CX ∗ AW − AX ∗ CW;
      integer_t c2 = CY ∗ AW − AY ∗ CW;
      integer_t d1 = DX ∗ AW − AX ∗ DW;
      integer_t d2 = DY ∗ AW − AY ∗ DW;
```

```
    bx = b1 * AW * BW;
    by = b2 * AW * BW;
    bw = b1 * b1 + b2 * b2;
    cx = c1 * AW * CW;
    cy = c2 * AW * CW;
    cw = c1 * c1 + c2 * c2;
    dx = d1 * AW * DW;
    dy = d2 * AW * DW;
    dw = d1 * d1 + d2 * d2;
  }
  return sign((bx * cy - by * cx) * dw + (by * cw - bw * cy) * dx + (bw * cx - bx * cw) * dy);
}
int orientation(const rat_point &a, const rat_point &b, const rat_point &c)
{ /* copied from the LEDA 2d module _rat_point.c */
  integer_t AX = a.X();
  integer_t AY = a.Y();
  integer_t AW = a.W();
  integer_t BX = b.X();
  integer_t BY = b.Y();
  integer_t BW = b.W();
  integer_t CX = c.X();
  integer_t CY = c.Y();
  integer_t CW = c.W();
  integer_t D = (AX * BW - BX * AW) * (AY * CW - CY * AW) - (AY * BW - BY *
      AW) * (AX * CW - CX * AW);
  return sign(D);
}
rational_t area(const rat_point &a, const rat_point &b, const rat_point &c)
{
  return ((a.xcoord() - b.xcoord()) * (a.ycoord() - c.ycoord()) -
      (a.ycoord() - b.ycoord()) * (a.xcoord() - c.xcoord()))/2;
}
rat_point rat_point::rotate90(const rat_point &q) const
{
  integer_t x0 = q.X();
  integer_t y0 = q.Y();
  integer_t w0 = q.W();
  integer_t x1 = X();
  integer_t y1 = Y();
  integer_t w1 = W();
  integer_t x = (x0 + y0) * w1 - y1 * w0;
  integer_t y = (y0 - x0) * w1 + x1 * w0;
  return rat_point(x, y, w0 * w1);
}
```

This code is used in chunk 3.

19. A Test of class rat_point

We do the testing in two stages. First we consider the easier and direct implemented 2-space operations. There we test only the special functions. Then we shift our attention to the d-dimensional operations which we test in 3-space.

```
⟨ rat_point-test.c 19 ⟩ ≡
#include "rat_point.h"
  main()
  {
    ⟨ 2d tests 20 ⟩
    ⟨ 3d tests 21 ⟩
  }
```

20. In this chunk we test the special 2d procedures.

```
⟨ 2d tests 20 ⟩ ≡
  { /* some construction test */
    cout ≪ "\n2-SPACE␣MODULE:\n";
    rat_point p0 = rat_point::origin(), p1(1,0), p2 = rat_point::d2(0,1), p3(2);
    cout ≪ "three␣points␣p0,p1,p2:␣" ≪ p0 ≪ p1 ≪ p2 ≪ "\n";
    /* some input and access test */
    cout ≪ "enter␣fourth␣point␣p3:␣";
    cin ≫ p3;
    cout ≪ "access␣operations␣on␣p3:";
    cout ≪ "\ncartesian:␣" ≪ p3.xcoord() ≪ "␣" ≪ p3.ycoord();
    cout ≪ "\nhomogenous:␣" ≪ p3.X() ≪ "␣" ≪ p3.Y() ≪ "␣" ≪ p3.W();
    /* the geometric operations and predicates */
    cout ≪ "\n(p2␣==␣p1.rotate90())␣=␣" ≪ (p2 ≡ p1.rotate90());
    cout ≪ "\np3.rotate90(p2):␣" ≪ p3.rotate90(p2);
    cout ≪ "\narea(p0,p1,p2)␣=␣" ≪ area(p0,p1,p2);
    cout ≪ "\norientation(p0,p1,p3)␣=␣" ≪ orientation(p0,p1,p3);
    array⟨rat_point⟩ tp(3);
    tp[0] = p0; tp[1] = p1; tp[2] = p3;
    cout ≪ "\norientation(<p0,p1,p3>)␣=␣" ≪ orientation(tp);
    cout ≪ "\nleft_turn(p0,p1,p3)␣=␣" ≪ left_turn(p0,p1,p3);
    cout ≪ "\nright_turn(p0,p1,p3)␣=␣" ≪ right_turn(p0,p1,p3);
    tp[2] = p2;
    cout ≪ "\nside_of_oriented_circle(p0,p1,p2,p3)␣=␣" ≪ side_of_oriented_circle(p0,
        p1,p2,p3);
    cout ≪ "\nside_of_oriented_sphere(<p0,p1,p2>,p3)␣=␣" ≪ side_of_oriented_sphere(tp,
        p3);
    cout ≪ "\nside_of_circle(p0,p1,p2,p3)␣=␣" ≪ side_of_circle(p0,p1,p2,p3);
    cout ≪ "\nside_of_sphere(<p0,p1,p2>,p3)␣=␣" ≪ side_of_sphere(tp,p3);
    cout ≪ "\nincircle(p0,p1,p2,p3)␣=␣" ≪ incircle(p0,p1,p2,p3);
    cout ≪ "\noutcircle(p0,p1,p2,p3)␣=␣" ≪ outcircle(p0,p1,p2,p3);
    rat_point p5 = rat_point::d2(1,0,2);
    cout ≪ "\nthe␣points␣p0,p1,p5:␣" ≪ p0 ≪ p1 ≪ p5;
    cout ≪ "\nside_of_oriented_circle(p0,p5,p1,p3)␣=␣" ≪ side_of_oriented_circle(p0,
        p5,p1,p3);
```

```
    cout ≪ "\n";
  }
```

This code is used in chunk 19.

21. Next we do some tests in 3-space. Here we test the d-dimensional components.

```
⟨ 3d tests 21 ⟩ ≡
  { /* some construction test */
    cout ≪ "\nd-SPACE␣MODULE:␣(d=3)\n";
    rat_point p0 = rat_point::origin(3); // the origin
    rat_vector e1 = rat_vector::unit(0,3), e2 = rat_vector::unit(1,3);
    // the first two unit vectors
    rat_point p1(p0 + e1), p2(p0 + e2), p3 = rat_point::d3(0,0,1), p4(3);
    cout ≪ "four␣points␣p0,p1,p2,p3:\n";
    cout ≪ p0 ≪ p1 ≪ p2 ≪ p3;
    cout ≪ "\np0.is_origin()␣=␣" ≪ p0.is_origin();
    cout ≪ "\np1.is_origin()␣=␣" ≪ p1.is_origin();
    /* some input and access test */
    cout ≪ "\nenter␣fifth␣point␣p4:␣";
    cin ≫ p4;
    cout ≪ "access␣operations␣on␣p4:";
    cout ≪ "\ncartesian:␣";
    int i;
    for (i = 0; i < p4.dim(); i++) cout ≪ p4.coord(i) ≪ "␣";
    cout ≪ "\nhomogenous:␣";
    for (i = 0; i ≤ p4.dim(); i++) cout ≪ p4.hcoord(i) ≪ "␣";
    cout ≪ "\ncompare(p1,p4)␣=␣" ≪ compare(p1, p4);
    /* orientation, sphere position, simplex position */
    array⟨rat_point⟩ A(4);
    A[0] = p0; A[1] = p1; A[2] = p2; A[3] = p4;
    cout ≪ "\norientation(<p0,p1,p2,p4>)␣=␣" ≪ orientation(A);
    A[3] = p3;
    cout ≪ "\nside_of_sphere(<p0,p1,p2,p3>,p4)␣=␣" ≪ side_of_sphere(A, p4);
    cout ≪ "\ncontained_in_simplex(<p0,p1,p2,p3>,p4)␣=␣";
    cout ≪ contained_in_simplex(A, p4);
    /* affine hull, independence and rank */
    array⟨rat_point⟩ P12(2);
    P12[0] = p1; P12[1] = p2;
    cout ≪ "\ncontained_in_affine_hull(<p1,p2>,p4)␣=␣";
    cout ≪ contained_in_affine_hull(P12, p4);
    rat_vector a = p4.to_rat_vector();
    cout ≪ "\na␣translation␣vector␣a(p4):␣" ≪ a;
    p0 -= a; p1 = p1 - a; p2 += a; p3 = p3 + a;
    cout ≪ "\nwe␣translate␣the␣points␣p0-=a,␣p1-=a,␣p2+=a␣,p3+=a:\n";
    cout ≪ p0 ≪ p1 ≪ p2 ≪ p3;
    array⟨rat_point⟩ B(3);
```

```
    B[0] = p1; B[1] = p2; B[2] = p3;
    A[0] = p0; A[1] = p1; A[2] = p2; A[3] = p4;
    cout ≪ "\naffinely_independent(<p0,p1,p2,p3>)␣=␣";
    cout ≪ affinely_independent(A);
    cout ≪ "\naffine_rank(<p0,p1,p2,p3>)␣=␣";
    cout ≪ affine_rank(A);
    cout ≪ "\n\n";
  }
```

This code is used in chunk 19.

Index

List of Refinements

⟨ 2d tests 20 ⟩ Used in chunk 19.
⟨ 3d tests 21 ⟩ Used in chunk 19.
⟨ affine operations 15, 16, 17 ⟩ Used in chunk 3.
⟨ arithmetic operations 7, 8, 9 ⟩ Used in chunk 3.
⟨ conversions 5 ⟩ Used in chunk 3.
⟨ initialization 4 ⟩ Used in chunk 3.
⟨ input and output 6 ⟩ Used in chunk 3.
⟨ position tests 11, 12, 13, 14 ⟩ Used in chunk 3.
⟨ `rat_point-test.c` 19 ⟩
⟨ `rat_point.c` 3 ⟩
⟨ `rat_point.h` 2 ⟩
⟨ special operations of 2-space 18 ⟩ Used in chunk 3.

Vectors with Rational Coordinates in d-Space (class rat_vector)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_vector

1. Definition

An instance of data type *rat_vector* is a vector of rational numbers. A d-dimensional vector $r = (r_0, \ldots, r_{d-1})$ is represented in homogeneous coordinates $(h_0, \ldots, h_d)$, where $r_i = h_i/h_d$ and the h_i's are of type *integer*. We call the r_i's the cartesian coordinates of the vector. The homogenizing coordinate h_d is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type *rat_point*). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

rat_vector is an item type.

2. Creation

rat_vector $v(int\ d\ =\ 2)$; introduces a variable v of type *rat_vector* initialized to the zero vector of dimension d.

rat_vector $v(integer\ a, integer\ b, integer\ D\ =\ 1)$;
introduces a variable v of type *rat_vector* initialized to the two-dimensional vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector $v(integer\_vector\ \ c, integer\ D)$;
introduces a variable v of type *rat_vector* initialized to the vector with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_vector $v(integer\_vector\ c)$; introduces a variable v of type *rat_vector* initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_vector *rat_vector*::d2($integer\ a, integer\ b, integer\ D\ =\ 1$)
returns a *rat_vector* of dimension 2 initialized to a vector with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_vector	*rat_vector*::d3(*integer a*, *integer b*, *integer c*, *integer D* = 1)	returns a *rat_vector* of dimension 3 initialized to a vector with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative. *Precondition*: D is non-zero.
rat_vector	*rat_vector*::unit(*int i*, *int d* = 2)	returns a *rat_vector* of dimension d initialized to the i-th unit vector. *Precondition*: $0 \leq i < d$.
rat_vector	*rat_vector*::zero(*int d* = 2)	returns the zero vector in d-dimensional space.
int	v.dim( )	returns the dimension of v.
rational	v.coord(*int i*)	returns the i-th cartesian coordinate of v.
rational	v[*int i*]	returns the i-th cartesian coordinate of v.
integer	v.hcoord(*int i*)	returns the i-th homogeneous coordinate of v.
rat_direction	v.to_rat_direction( )	converts to a direction.
rat_direction	to_rat_direction(*rat_vector v*)	converts to a direction.
rat_point	v.to_rat_point( )	converts to a point.
rat_point	to_rat_point(*rat_vector v*)	converts to a point.
rat_vector	v.transform(*aff_transformation t*)	returns $t(v)$.

Additional Operations for vectors in two-dimensional space

rational	v.xcoord( )	returns the zeroth cartesian coordinate of v.
rational	v.ycoord( )	returns the first cartesian coordinate of v.
integer	v.X( )	returns the zeroth homogeneous coordinate of v.
integer	v.Y( )	returns the first homogeneous coordinate of v.
integer	v.W( )	returns the homogenizing coordinate of v.

3.2 Tests

bool	identical(*rat_vector v*, *rat_vector w*)	Test for identity.

bool	$v \equiv w$	Test for equality.
bool	v != w	Test for inequality.

3.3 Arithmetical Operators

rat_vector	*integer* $n * v$	multiplies all cartesian coordinates by n.
rat_vector	*rational* $r * v$	multiplies all cartesian coordinates by r.
rat_vector&	$v *=$ *integer* n	multiplies all cartesian coordinates by n.
rat_vector&	$v *=$ *rational* r	multiplies all cartesian coordinates by r.
rat_vector	v / *integer* n	divides all cartesian coordinates by n.
rat_vector	v / *rational* r	divides all cartesian coordinates by n.
rat_vector&	$v /=$ *integer* n	divides all cartesian coordinates by n.
rat_vector&	$v /=$ *rational* r	divides all cartesian coordinates by r.
rational	$v * w$	scalar product, i.e., $\sum_{0 \le i < d} v_i w_i$, where v_i and w_i are the cartesian coordinates of v and w respectively.
rat_vector	$v + w$	adds cartesian coordinates.
rat_vector&	$v+= w$	addition plus assignment.
rat_vector	$v - w$	subtracts cartesian coordinates
rat_vector&	$v-= w$	subtraction plus assignment.
rat_vector	$-v$	returns -v.

3.4 Input and Output

ostream&	*ostream&* $O \ll v$	writes v's homogeneous coordinates componentwise to the output stream O.
istream&	*istream&* $I \gg$ *rat_vector&* v	reads v's homogeneous coordinates componentwise from the input stream I. The operator uses the current dimension of v.

3.5 Linear Hull, Dependence and Rank

bool	contained_in_linear_hull(*array<rat_vector>* A, *rat_vector* x)	determines whether x is contained in the linear hull of the vectors in A.

int linear_rank(*array*<*rat_vector*>*A*)

computes the linear rank of the vectors in A.

bool linearly_independent(*array*<*rat_vector*>*A*)

decides whether the vectors in A are linearly independent.

array<*rat_vector*> linear_base(*array*<*rat_vector*>*A*)

computes a basis of the linear space spanned by the vectors in A.

4. Implementation

Vectors are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on an vector v take time $O(v.dim(\,))$. $dim(\,)$, coordinate access and conversions take constant time. The operations for linear hull, rank and independence have the cubic costs of the used matrix operations. The space requirement is $O(v.dim(\,))$.

2. The Header File of class rat_vector

The type rat_vector is an item class with representation class geo_rep. It shares this representation class with hyperplanes, points, and directions. We derive rat_vector from handle_base and derive geo_rep from handle_rep. This gives us reference counting for free. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_vector.h 2 ⟩ ≡
#ifndef LEDA_RAT_VECTOR_H
#define LEDA_RAT_VECTOR_H
#include "rat_point.h"
#include "rat_direction.h"
#include "aff_transform.h"
#include "geo_rep.h"
  class rat_point;
  class rat_direction;
  class aff_transformation;
  class rat_vector : public handle_base {
  geo_rep *ptr() const { return (geo_rep *) PTR; }
  rat_vector(const handle_base &p) : handle_base(p) { }
  friend class rat_point;
  friend class rat_direction;
  friend class rat_hyperplane;
  friend class geo_pair_rep;
public:
  rat_vector(int d = 2) { PTR = new geo_rep (d); ptr()->v[d] = 1; }
  rat_vector(integer_t a, integer_t b, integer_t D = 1)
  {
    if (D == 0) error_handler(1, "rat_vector::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (-a, -b, -D);
    else PTR = new geo_rep (a, b, D);
  }
  rat_vector(const integer_vector &c, integer_t D)
  {
    if (D == 0) error_handler(1, "rat_vector::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (-c, -D);
    else PTR = new geo_rep (c, D);
  }
  rat_vector(const integer_vector &c)
  {
    int d = c.dim();
    integer_t D = c[d - 1];
    if (D == 0) error_handler(1, "rat_point::constructor: D must be nonzero");
    if (D < 0) PTR = new geo_rep (-c);
    else PTR = new geo_rep (c);
  }
  rat_vector(const rat_vector &p) : handle_base(p) { }
  ~rat_vector() { }
```

```
rat_vector &operator=(const rat_vector &p)
{ handle_base::operator=(p); return *this; }
static rat_vector d2(integer_t a, integer_t b, integer_t D = 1);
static rat_vector d3(integer_t a, integer_t b, integer_t c, integer_t D = 1);
static rat_vector unit(int i, int d = 2);
static rat_vector zero(int d = 2);
int dim( ) const { return ptr( )->dim; }
rational_t coord(int i) const { return rational_t(ptr( )->v[i], ptr( )->v[ptr( )->dim]); }
rational_t operator [](int i) const { return coord(i); }
integer_t hcoord(int i) const { return ptr( )->v[i]; }
rat_direction to_rat_direction( ) const;
friend rat_direction to_rat_direction(const rat_vector &v);
rat_point to_rat_point( ) const;
friend rat_point to_rat_point(const rat_vector &v);
rat_vector transform(const aff_transformation &t) const;
rational_t xcoord( ) const { return rational_t(hcoord(0), hcoord(ptr( )->dim)); }
rational_t ycoord( ) const { return rational_t(hcoord(1), hcoord(ptr( )->dim)); }
integer_t X( ) const { return hcoord(0); }
integer_t Y( ) const { return hcoord(1); }
integer_t W( ) const { return hcoord(ptr( )->dim); }
friend bool identical(const rat_vector &v, const rat_vector &w)
{ return v.ptr( ) ≡ w.ptr( ); }
int cmp(const rat_vector &x, const rat_vector &y) const
{ return (identical(x, y) ? 0 :
      x.ptr( )->cmp_rat_coords(x.ptr( ), y.ptr( ))); }
friend int compare(const rat_vector &p, const rat_vector &q)
{ return p.cmp(p, q); }
bool operator≡(const rat_vector &w) const
{ return cmp(*this, w) ≡ 0; }
bool operator≠(const rat_vector &w) const
{ return cmp(*this, w) ≠ 0; }
rat_vector rat_vector::scale(integer_t m, integer_t n) const;
void rat_vector::self_scale(integer_t m, integer_t n);
friend rat_vector operator * (int n, const rat_vector &v);
friend rat_vector operator * (integer_t n, const rat_vector &v);
friend rat_vector operator * (rational_t r, const rat_vector &v);
rat_vector &operator*= (integer_t n);
rat_vector &operator*= (int n);
rat_vector &operator*= (rational_t r);
friend rat_vector operator/(const rat_vector &v, int n);
friend rat_vector operator/(const rat_vector &v, integer_t n);
friend rat_vector operator/(const rat_vector &v, rational_t r);
rat_vector &operator/=(integer_t n);
rat_vector &operator/=(int n);
rat_vector &operator/=(rational_t r);
friend rational_t operator * (const rat_vector v, const rat_vector &w);
```

```
  friend rat_vector operator+(const rat_vector &v, const rat_vector &w);
  rat_vector &operator+=(const rat_vector &w);
  friend rat_vector operator-(const rat_vector &v, const rat_vector &w);
  rat_vector &operator-=(const rat_vector &w);
  rat_vector operator-() const;
  friend ostream &operator<<(ostream &O, const rat_vector &v);
  friend istream &operator>>(istream &I, rat_vector &v);
  };
  inline void Print(const rat_vector &v, ostream &out) { out << v; }
  inline void Read(rat_vector &v, istream &in) { in >> v; }
  bool contained_in_linear_hull(const array<rat_vector> &A, const rat_vector &x);
  int linear_rank(const array<rat_vector> &A);
  bool linearly_independent(const array<rat_vector> &A);
  array<rat_vector> linear_base(const array<rat_vector> &A);
#endif
```

3. The Implementation of class rat_vector

```
⟨ rat_vector.c 3 ⟩ ≡
#include "rat_vector.h"
  ⟨ initialization 4 ⟩
  ⟨ conversions 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ arithmetic operations 7 ⟩
  ⟨ linear operations 11 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_vector rat_vector::d2(integer_t a, integer_t b, integer_t D)
  {
    if (D ≡ 0) error_handler(1, "rat_vector::d2: denominator must not be zero.");
    return rat_vector(a, b, D);
  }
  rat_vector rat_vector::d3(integer_t a, integer_t b, integer_t c, integer_t D)
  {
    rat_vector d3v(3);
    if (D ≡ 0) error_handler(1, "rat_vector::d3: denominator must not be zero.");
    if (D < 0) d3v.ptr()→init4(−a, −b, −c, −D);
    else d3v.ptr()→init4(a, b, c, D);
    return d3v;
  }
  rat_vector rat_vector::unit(int i, int d)
  {
    if (i < 0 ∨ i ≥ d) error_handler(1, "rat_vector::unit: i out of range.");
    rat_vector uv(d);
    uv.ptr()→v[i] = 1;
    return uv;
  }
  rat_vector rat_vector::zero(int d) { return rat_vector(d); }
```

This code is used in chunk 3.

5. Conversions

Vectors can be converted to points and directions. Since the converted object has the same representation, conversion amounts to a call of the copy constructor of the base class.

```
⟨ conversions 5 ⟩ ≡
  rat_direction rat_vector::to_rat_direction() const
  {
    if (to_rat_point().is_origin()) error_handler(1,
          "rat_vector::to_rat_direction: origin cannot be a direction.");
    return rat_direction(*this);
  }
```

```
rat_point rat_vector::to_rat_point( ) const
{ return rat_point(*this); }
rat_direction to_rat_direction(const rat_vector &v)
{ return v.to_rat_direction( ); }
rat_point to_rat_point(const rat_vector &v)
{ return v.to_rat_point( ); }
```

This code is used in chunk 3.

6. Input and Output

We just take care that in case of input we don't overwrite a common representation. Apart from that we use the I/O-operations of geo_rep.

⟨ input and output 6 ⟩ ≡

```
ostream &operator≪(ostream &out, const rat_vector &a)
{
  out ≪ a.ptr( );
  return out;
}
istream &operator≫(istream &in, rat_vector &a)
{
  int d = a.dim( );
  if (a.refs( ) > 1) a = rat_vector(d);
  in ≫ a.ptr( );
  if (a.hcoord(d) ≡ 0)
    error_handler(1,"operator>>: denominator of vector must be nonzero.");
  if (a.hcoord(d) < 0) a.ptr( )→negate(d + 1);
  return in;
}
```

This code is used in chunk 3.

7. Multiplication and Division

We start with the operators $*$ and $/$. We define a function $scale(m, n)$ that multiplies the homogenizing coordinate by n and all other coordinates by m and reduce the operators to this function.

⟨ arithmetic operations 7 ⟩ ≡

```
rat_vector rat_vector::scale(integer_t m, integer_t n) const
{
  int d = dim( );
  rat_vector result(d);
  result.ptr( )→v[d] = ptr( )→v[d] * n;
  integer_t g = gcd(ptr( )→v[d], m);
  result.ptr( )→v[d] /= g;
  m /= g;
  for (int i = 0; i < d; i++) result.ptr( )→v[i] = ptr( )→v[i] * m;
  return result;
}
```

```
void rat_vector::self_scale(integer_t m, integer_t n)
{
  int d = dim( );
  ptr( )→v[d] *= n;
  integer_t g = gcd(ptr( )→v[d], m);
  ptr( )→v[d] /= g;
  m /= g;
  for (int i = 0; i < d; i++) ptr( )→v[i] *= m;
}
rat_vector operator * (int n, const rat_vector &p)
{ return p.scale(n, 1); }
rat_vector operator * (integer_t n, const rat_vector &p)
{ return p.scale(n, 1); }
rat_vector operator * (rational_t r, const rat_vector &p)
{ return p.scale(r.numerator( ), r.denominator( )); }
rat_vector operator/(const rat_vector &p, int n)
{ return p.scale(1, n); }
rat_vector operator/(const rat_vector &p, integer_t n)
{ return p.scale(1, n); }
rat_vector operator/(const rat_vector &p, rational_t r)
{ return p.scale(r.denominator( ), r.numerator( )); }
rat_vector &rat_vector::operator* = (integer_t n)
{ self_scale(n, 1); return *this; }
rat_vector &rat_vector::operator* = (int n)
{ self_scale(n, 1); return *this; }
rat_vector &rat_vector::operator* = (rational_t r)
{ self_scale(r.numerator( ), r.denominator( )); return *this; }
rat_vector &rat_vector::operator/=(integer_t n)
{ self_scale(1, n); return *this; }
rat_vector &rat_vector::operator/=(int n)
{ self_scale(1, n); return *this; }
rat_vector &rat_vector::operator/=(rational_t r)
{ self_scale(r.denominator( ), r.numerator( )); return *this; }
rational_t operator * (const rat_vector v, const rat_vector &w)
{
  int d = v.dim( );
  if (d ≠ w.dim( )) error_handler(1, "inner␣product:␣dimensions␣disagree.");
  integer_t nom = 0;
  for (int i = 0; i < d; i++) nom += v.hcoord(i) * w.hcoord(i);
  integer_t denom = v.hcoord(d) * w.hcoord(d);
  return rational_t(nom, denom);
}
```

See also chunks 8 and 9.

This code is used in chunk 3.

8. Addition and Subtraction

To implement addition and substraction of the cartesien coordinates we use common code in our class geo_rep. As these operations have the same functionality which are required for rat_points.

```
⟨ arithmetic operations 7 ⟩ +≡
  rat_vector operator+(const rat_vector &v, const rat_vector &w)
  {
    rat_vector res(v.dim( ));
    c_add(res.ptr( ), v.ptr( ), w.ptr( ));
    return res;
  }
  rat_vector &rat_vector::operator+=(const rat_vector &w)
  {
    int d = dim( );
    rat_vector old(*this);
    if (ptr( )->count > 2) *this = rat_vector(d);
    c_add(ptr( ), old.ptr( ), w.ptr( ));
    return *this;
  }
  rat_vector operator-(const rat_vector &v, const rat_vector &w)
  {
    rat_vector res(v.dim( ));
    c_sub(res.ptr( ), v.ptr( ), w.ptr( ));
    return res;
  }
  rat_vector &rat_vector::operator-=(const rat_vector &w)
  {
    int d = dim( );
    rat_vector old(*this);
    if (ptr( )->count > 2) *this = rat_vector(d);
    c_sub(ptr( ), old.ptr( ), w.ptr( ));
    return *this;
  }
  rat_vector rat_vector::operator-( ) const
  {
    int d = dim( );
    rat_vector result(d);
    result.ptr( )->copy(ptr( ));
    result.ptr( )->negate(d);
    return result;
  }
```

9. Transformation

To transform a vector we interpret it as a point in the coordinate system with origin *origin*( ) and take the the difference between the transformed point and the transformed origin as the transformed vector. There should be more efficient implementation possible when you take out the translational part of the transformation.

⟨ arithmetic operations 7 ⟩ +≡

```
rat_vector rat_vector::transform(const aff_transformation &t) const
{
  integer_matrix m_at = t.matrix( );
  int d = t.dim( );
  for (int i = 0; i < d; i++) m_at(i, d) = 0;
  return rat_vector(m_at * (ptr( )→ivec( )));
}
```

10. Linear Algebra

11. Containment in Linear Hull. A vector x is contained in the linear hull of a set A of vectors if x is a linear combination of the vectors in A, i.e., if the system $\sum \lambda_i A_i = x$ has a solution. We may scale each vector by its homogenizing coordinate and hence forget about the homogenizing coordinates.

⟨ linear operations 11 ⟩ ≡

```
bool contained_in_linear_hull(const array⟨rat_vector⟩ &A, const rat_vector &x)
{
  int al = A.low( );
  int k = A.high( ) − al + 1;  // A contains k vectors
  int d = A[al].dim( );
  integer_matrix M(d, k);
  integer_vector b(d);
  for (int i = 0; i < d; i++) {
    b[i] = x.hcoord(i);
    for (int j = 0; j < k; j++) M(i, j) = A[al + j].hcoord(i);
  }
  return solvable(M, b);
}
```

See also chunks 12, 13, and 14.

This code is used in chunk 3.

12. Linear Rank.

We set up a matrix having the cartesian coordinates of the vectors in A as its columns. The linear rank is the rank of this matrix. Since the rank of a matrix does not change under multiplication of a row with a constant we may also use the homogeneous coordinates 0 to $d-1$ of the vectors in A.

⟨ linear operations 11 ⟩ +≡

```
int linear_rank(const array⟨rat_vector⟩ &A)
{
  int al = A.low( );
  int k = A.high( ) − al + 1;  // A contains k vectors
  int d = A[al].dim( );
  integer_matrix M(d, k);
  for (int i = 0; i < d; i++)
    for (int j = 0; j < k; j++) M(i, j) = A[al + j].hcoord(i);
  return rank(M);
}
```

13. Linear Independence.

A set of vectors is linearly independent if their linear rank is equal to the number of vectors in the set.

```
⟨ linear operations 11 ⟩ +≡
  bool linearly_independent(const array⟨rat_vector⟩ &A)
  { return (linear_rank(A) ≡ A.high( ) − A.low( ) + 1); }
```

14. Linear Base.

```
⟨ linear operations 11 ⟩ +≡
  array⟨rat_vector⟩ linear_base(const array⟨rat_vector⟩ &A)
  {
    int al = A.low( );
    int k = A.high( ) − al + 1;  // A contains k vectors
    int d = A[al].dim( );
    integer_t denom;
    integer_matrix M(d, k);
    for (int j = 0; j < k; j++)
      for (int i = 0; i < d; i++) M(i, j) = A[al + j].hcoord(i);
    array⟨int⟩ indcols;
    independent_columns(M, indcols);
    int indcolsdim = indcols.high( ) + 1;
    array⟨rat_vector⟩ L(indcolsdim);
    for (int i = 0; i < indcolsdim; i++) L[i] = rat_vector(M.col(indcols[i]), 1);
    return L;
  }
```

15. A Test of class rat_vector

We test the construction, access to the components and all the arithmetical operators in 3-Space.

```
⟨ rat_vector-test.c 15 ⟩ ≡
#include "rat_vector.h"
  main( )
  { /* construction and access */
    integer_vector vi1(4), vi2(3);
    vi1[0] = 1; vi1[1] = 2; vi1[2] = 3; vi1[3] = 1;
    vi2[0] = 3; vi2[1] = 2; vi2[2] = 1;
    rat_vector a0(3), a1(vi1), a2(vi2, 1), a3 = rat_vector::d3(4, −4, 0, 1), a4(3);
    cout ≪ "four vectors a0,a1,a2,a3:\n" ≪ a0 ≪ a1 ≪ a2 ≪ a3;
    cout ≪ "\nenter fifth vector a4: ";
    cin ≫ a4;
    cout ≪ "access operations on a4:\n";
    cout ≪ "cartesian: ";
    for (int i = 0; i < a4.dim( ); i++) cout ≪ a4.coord(i) ≪ " ";
    cout ≪ "\nhomogenous: ";
    for (int i = 0; i ≤ a4.dim( ); i++) cout ≪ a4.hcoord(i) ≪ " ";
    /* arithmetical operations and compare */
    cout ≪ "\nquadratic length of vector a4 = " ≪ a4 * a4;
    cout ≪ "\ncompare(a0,a4) = " ≪ compare(a0, a4);
    vi1[0] = 1; vi1[1] = 1; vi1[2] = 1; vi1[3] = 1;
    rat_vector eins(vi1);
    cout ≪ "\na0 = a1 + a2: " ≪ (a0 = a1 + a2);
    cout ≪ "\na0 += eins: " ≪ (a0 += eins);
    cout ≪ "\na0 -= eins: " ≪ (a0 −= eins);
    cout ≪ "\na0 = a0 - a4: " ≪ (a0 = a0 − a4);
    cout ≪ "\na0 = 2 * a0: " ≪ (a0 = 2 * a0);
    cout ≪ "\na0 = 1/2 * a0: " ≪ (a0 = rational_t(1, 2) * a0);
    cout ≪ "\na0 *= 3: " ≪ (a0 *= 3);
    cout ≪ "\na0 *= 1/3: " ≪ (a0 *= rational_t(1, 3));
    cout ≪ "\na0 = a0 / 4: " ≪ (a0 = a0/4);
    cout ≪ "\na0 = a0 / 1/4: " ≪ (a0 = a0/rational_t(1, 4));
    cout ≪ "\na0 /= 5: " ≪ (a0 /= 5);
    cout ≪ "\na0 /= 1/5: " ≪ (a0 /= rational_t(1, 5));
    if ((a1 + a2 − a4) ≡ a0) cout ≪ "\n(a1+a2-a4) == a0\n";
    else cout ≪ "\nsomething wrong: (a1+a2-a4) != a0\n";
    cout ≪ "\na0 = -a4: " ≪ (a0 = −a4);
    /* some linear algebra */
    array⟨rat_vector⟩ P12(2);
    P12[0] = a1; P12[1] = a2;
    cout ≪ "\ncontained_in_linear_hull(<a1,a2>,a4) = ";
    cout ≪ contained_in_linear_hull(P12, a4);
    cout ≪ "\ncontained_in_linear_hull(<a1,a2>,a1+a2) = ";
    cout ≪ contained_in_linear_hull(P12, a1 + a2);
    array⟨rat_vector⟩ B(3), A(4);
    B[0] = a0; B[1] = a1; B[2] = a2;
    A[0] = a0; A[1] = a1; A[2] = a2; A[3] = a1 + a2;
```

```
  cout << "\nlinearly_independent(<a0,a1,a2>) = ";
  cout << linearly_independent(B);
  cout << "\nlinearly_independent(<a0,a1,a2,a1+a2>) = ";
  cout << linearly_independent(A);
  cout << "\nlinear_rank(<a0,a1,a2,a1+a2>) = ";
  cout << linear_rank(A);
  cout << "\nlinear_base(<a0,a1,a2,a1+a2>) = ";
  array<rat_vector> res = linear_base(A);
  for (int i = 0; i < res.high( ) + 1; i++) cout << res[i];
  cout << "\n\n";
}
```

Index

Directions with Rational Coordinates in d-Space (class rat_direction)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_direction

1. Definition

A *rat_direction* is any non-zero vector. We represent directions in d-dimensional space as a tuple $(h_0, \ldots, h_d)$ of integers which we call the homogeneous coordinates of the direction. The coordinate h_d must be positive. The cartesian coordinates of a direction are $c_i = h_i/h_d$ for $0 \leq i < d$. Two directions are equal if their cartesian coordinates are positive multiples of each other. Directions are in one-to-one correspondence to points on the unit sphere.

In two-dimensional space directions are also in one-to-one correspondance to angles. More precisely, a direction $dir = (h_0, h_1, h_2)$ corresponds to the angle α with $\sin\alpha = c_0/L$ and $\cos\alpha = c_1/L$ and $L = \sqrt{(h_0^2 + h_1^2)/h_2^2}$ the length of dir. For a direction dir we use $angle(dir)$ to denote this angle.

rat_direction is an item type.

2. Creation

rat_direction *dir*(*int* $d = 2$); introduces a variable *dir* of type *rat_direction* initialized to some direction in d-dimensional space.

rat_direction *dir*(*integer* a, *integer* b, *integer* $D = 1$);
introduces a variable *dir* of type *rat_direction* initialized to the two-dimensional direction with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction *dir*(*integer_vector* c, *integer* D);
introduces a variable *dir* of type *rat_direction* initialized to the two-dimensional direction with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

rat_direction *dir*(*integer_vector* c);
introduces a variable *dir* of type *rat_direction* initialized to the direction with homogeneous coordinate vector $\pm c$, where the sign chosen is the sign of the last component of c.
Precondition: The last component of c is non-zero.

3. Operations

3.1 Initialization, Access and Conversions

rat_direction *rat_direction*::d2(*integer* a, *integer* b, *integer* $D = 1$)

returns a *rat_direction* of dimension 2 initialized to a direction with homogeneous representation (a, b, D) if D is positive and representation $(-a, -b, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction *rat_direction*::d3(*integer* a, *integer* b, *integer* c, *integer* $D = 1$)

returns a *rat_direction* of dimension 3 initialized to a direction with homogeneous representation (a, b, c, D) if D is positive and representation $(-a, -b, -c, -D)$ if D is negative.
Precondition: D is non-zero.

rat_direction *rat_direction*::unit(*int* i, *int* $d = 2$)

returns a *rat_direction* of dimension d initialized to the i-th unit direction.
Precondition: $0 \leq i < d$.

int *dir*.dim( ) returns the dimension of *dir*.

rational *dir*.coord(*int* i) returns the i-th cartesian coordinate of *dir*.

rational *dir*[*int* i] returns the i-th cartesian coordinate of *dir*.

integer *dir*.hcoord(*int* i) returns the i-th homogeneous coordinate of *dir*.

rat_direction *dir*.transform(*aff_transformation* t)

returns $t(p)$.

rat_direction *dir*.opposite( ) returns the direction opposite to *dir*.

rat_direction $-dir$ returns *dir*.opposite().

rat_vector *dir*.to_rat_vector( ) returns a vector pointing in direction *dir*.

rat_vector *dir*.to_rat_vector(*rat_direction* d)

returns a vector pointing in direction *dir*.

Additional Operations for directions in two-dimensional space

integer *dir*.X( ) returns the zeroth homogeneous coordinate of *dir*.

integer *dir*.Y( ) returns the first homogeneous coordinate of *dir*.

3.2 Tests

bool identical(*rat_direction* v, *rat_direction* w)

Test for identity.

bool	$v \equiv w$	Test for equality.
bool	v != w	Test for inequality.

3.3 Input and Output

ostream&	*ostream&* $O \ll d$	writes the homogeneous coordinates of d to output stream O.
istream&	*istream&* $I \gg$ *rat_direction&* d	reads the homogeneous coordinates of d from input stream I. This operator uses the current dimension of d.

4. Implementation

Directions are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, inversion, input and output on a direction d take time $O(d.dim(\,))$. $dim(\,)$, coordinate access and conversion take constant time. The space requirement is $O(d.dim(\,))$.

2. The Header File of class rat_direction

The type rat_direction is an item class with representation class geo_rep. It shares this representation class with rat_hyperplanes, rat_points, and rat_directions. We derive rat_direction from handle_base and derive geo_rep from handle_rep. This gives us reference counting for free. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_direction.h 2 ⟩ ≡
#ifndef LEDA_RAT_DIRECTION_H
#define LEDA_RAT_DIRECTION_H
#include "rat_vector.h"
#include "rat_point.h"
#include "aff_transform.h"
#include "geo_rep.h"
  class aff_transformation;
  class rat_vector;
  class rat_point; class rat_direction : public handle_base {
  geo_rep *ptr( ) const { return (geo_rep *) PTR; }
  rat_direction(const handle_base &p) : handle_base(p) { }
  friend class rat_vector;
  friend class rat_point;
public:
  rat_direction(int d = 2) { PTR = new geo_rep (d); ptr( )->v[d] = 1; }
  rat_direction(integer_t a, integer_t b, integer_t D = 1)
  {
    if (D ≡ 0) error_handler(1, "rat_direction::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (−a, −b, −D);
    else PTR = new geo_rep (a, b, D);
  }
  rat_direction(const integer_vector &c, integer_t D)
  {
    if (D ≡ 0) error_handler(1, "rat_direction::constructor: D must be nonzero.");
    if (D < 0) PTR = new geo_rep (−c, −D);
    else PTR = new geo_rep (c, D);
  }
  rat_direction(const integer_vector &c)
  {
    int d = c.dim( );
    integer_t D = c[d − 1];
    if (D ≡ 0) error_handler(1, "d_rat_point::constructor: D must be nonzero");
    if (D < 0) PTR = new geo_rep (−c);
    else PTR = new geo_rep (c);
  }
  rat_direction(const rat_direction &p) : handle_base(p) { }
  ∼rat_direction( ) { }
  rat_direction &operator=(const rat_direction &p)
  { handle_base::operator=(p); return *this; }
```

```
  static rat_direction d2(integer_t a, integer_t b, integer_t D = 1);
  static rat_direction d3(integer_t a, integer_t b, integer_t c, integer_t D = 1);
  static rat_direction unit(int i, int d = 2);
  int dim( ) const { return ptr( )→dim; }
  rational_t coord(int i) const { return rational_t(ptr( )→v[i], ptr( )→v[ptr( )→dim]); }
  rational_t operator [](int i) const { return coord(i); }
  integer_t hcoord(int i) const { return ptr( )→v[i]; }
  rat_direction transform(const aff_transformation &t) const;
  rat_direction opposite( ) const;
  rat_direction operator−( ) const { return opposite( ); }
  rat_vector to_rat_vector( ) const;
  friend rat_vector to_rat_vector(rat_direction d);
  integer_t X( ) const { return hcoord(0); }
  integer_t Y( ) const { return hcoord(1); }
  friend bool identical(const rat_direction &v, const rat_direction &w)
  { return v.ptr( ) ≡ w.ptr( ); }
  int cmp(const rat_direction &h1, const rat_direction &h2) const;
  friend int compare(const rat_direction &p, const rat_direction &q)
  { return p.cmp(p, q); }
  friend bool operator≡(const rat_direction &v, const rat_direction &w)
  { return compare(v, w) ≡ 0; }
  friend bool operator≠(const rat_direction &v, const rat_direction &w)
  { return compare(v, w) ≠ 0; }
  friend ostream &operator≪(ostream &O, const rat_direction &d);
  friend istream &operator≫(istream &I, rat_direction &d);
  } ;
  inline void Print(const rat_direction &v, ostream &out) { out ≪ v; }
  inline void Read(rat_direction &v, istream &in) { in ≫ v; }
#if (0)
  inline int compare(const rat_direction &p, const rat_direction &q)
  { return p.cmp(p, q); }
#endif
#endif
```

3. The Implementation of class rat_direction

```
⟨ rat_direction.c 3 ⟩ ≡
#include "rat_direction.h"
  ⟨ initialization 4 ⟩
  ⟨ conversion 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ other operations 7 ⟩
  ⟨ compares 8 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_direction rat_direction::d2(integer_t a, integer_t b, integer_t D)
  {
    if (D ≡ 0) error_handler(1, "rat_direction::d2:␣denominator␣must␣not␣be␣zero.");
    return rat_direction(a, b, D);
  }
  rat_direction rat_direction::d3(integer_t a, integer_t b, integer_t c, integer_t D)
  {
    rat_direction d3d(3);
    if (D ≡ 0) error_handler(1, "rat_direction::d3:␣denominator␣must␣not␣be␣zero.");
    if (D < 0) d3d.ptr()→init4(−a, −b, −c, −D);
    else d3d.ptr()→init4(a, b, c, D);
    return d3d;
  }
  rat_direction rat_direction::unit(int i, int d)
  {
    if (i < 0 ∨ i ≥ d) error_handler(1, "_direction::unit:␣i␣out␣of␣range.");
    rat_direction ud(d);
    ud.ptr()→v[i] = 1;
    return ud;
  }
```

This code is used in chunk 3.

5. Conversion

Directions can be converted to vectors. Since the converted object has the same representation, conversion amounts to a call of the copy constructor of the base class.

```
⟨ conversion 5 ⟩ ≡
  rat_vector rat_direction::to_rat_vector() const
  { return rat_vector(*this); }
  rat_vector to_rat_vector(rat_direction d)
  { return d.to_rat_vector(); }
```

This code is used in chunk 3.

6. Input and Output

We just take care that in case of input we don't overwrite a common representation. Apart from that we use the I/O-operations of geo_rep.

⟨ input and output 6 ⟩ ≡

```
ostream &operator≪(ostream &out, const rat_direction &d)
{
  out ≪ d.ptr( );
  return out;
}
istream &operator≫(istream &in, rat_direction &d)
{
  int dim = d.dim( );
  if (d.refs( ) > 1) d = rat_direction((int) dim);
  in ≫ d.ptr( );
  if (d.hcoord(dim) ≡ 0)
    error_handler(1, "operator>>: denominator of direction must be nonzero.");
  if (d.hcoord(dim) < 0) d.ptr( )→negate(dim + 1);
  return in;
}
```

This code is used in chunk 3.

7. Other Operations. We can easily create the opposite direction by sign inversion. The transformation operation is the same as for the rat_vector type as we anyway ignore the length of directions. We could also implement a special transform operation which ignores the scaling and translation component of the transformation.

⟨ other operations 7 ⟩ ≡

```
rat_direction rat_direction::opposite( ) const
{
  int d = dim( );
  rat_direction result(d);
  result.ptr( )→copy(ptr( ));
  result.ptr( )→negate(d);
  return result;
}
rat_direction rat_direction::transform(const aff_transformation &t) const
{ return to_rat_vector( ).transform(t).to_rat_direction( ); }
```

This code is used in chunk 3.

8. The Compare Function. Two directions are equal if their cartesian coordinates are positive multiples of each other. This is equivalent to saying that their first $d-1$ homogeneous coordinates are positive multiples of each other. Thus we take the lexicographic order of the first $d-1$ homogeneous coordinates under multiplication of positive multiples. We gave the details of this already in **rat_hyperplane**::*strong_compare*.

⟨ compares 8 ⟩ ≡

```
int rat_direction::cmp(const rat_direction &h1, const rat_direction &h2) const
{
  if (identical(h1, h2)) return 0;
```

```
    int i;
    int d = h1.dim( );
    for (i = 0; i < d ∧ h1.hcoord(i) ≡ 0 ∧ h2.hcoord(i) ≡ 0; i++) ;   // no body
    int c1 = sign(h1.hcoord(i));
    int c2 = sign(h2.hcoord(i));
    if (c1 ≠ c2) return compare(c1, c2);
    integer_t s1 = (integer_t) sign(h2.hcoord(i)) * h2.hcoord(i);
    integer_t s2 = (integer_t) sign(h1.hcoord(i)) * h1.hcoord(i);
    i++;
    int c;
    while (i < d) {
      c = compare(s1 * h1.hcoord(i), s2 * h2.hcoord(i));
      if (c ≠ 0) return (c > 0 ? 1 : −1);
      i++;
    }
    return 0;
  }
```

This code is used in chunk 3.

9. A Test of class rat_direction

```
⟨ rat_direction-test.c 9 ⟩ ≡
#include "rat_direction.h"
  main( )
  { /* construction and access */
    integer_vector vi(4);
    vi[0] = 1; vi[1] = 5; vi[2] = −3; vi[3] = 1;
    rat_direction d0(vi), d1 = rat_direction::unit(2, 3), d2 = rat_direction::d3(1, 1, 0),
        d3(3);
    cout ≪ "three␣directions␣d0,d1,d2:\n" ≪ d0 ≪ d1 ≪ d2;
    cout ≪ "\nenter␣third␣direction␣d3:␣";
    cin ≫ d3;
    cout ≪ "access␣operations␣on␣d3:\n";
    cout ≪ "cartesian:␣";
    for (int i = 0; i < d3.dim( ); i++) cout ≪ d3.coord(i) ≪ "␣";
    cout ≪ "\nhomogenous:␣";
    for (int i = 0; i ≤ d3.dim( ); i++) cout ≪ d3.hcoord(i) ≪ "␣";
    /* inversion and compare */
    cout ≪ "\nd0.opposite()␣=␣" ≪ d0.opposite( );
    cout ≪ "\ncompare(d0,d3)␣=␣" ≪ compare(d0, d3);
    cout ≪ "\n\n";
  }
```

Index

List of Refinements

Hyperplanes with Rational Coordinates in d-Space (class rat_hyperplane)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_hyperplane

1. Definition

An instance of data type *rat_hyperplane* is a hyperplane with rational coefficients in an arbitrary dimensional space. A hyperplane h is represented by coefficients $(c_0, c_1, \ldots, c_d)$ of arbitrary length integers. At least one of c_0 to c_{d-1} must be non-zero. The plane equation is $\sum_{0 \le i \le d} c_i x_i = 0$, where x_0 to x_d are homogeneous point coordinates. The sign of the left hand side of this expression determines the position of a point x with respect to the hyperplane (on the hyperplane, on the negative side, or on the positive side).

There are two equality predicates for hyperplanes. The (weak) equality predicate (*operator* $\equiv$) declares two hyperplanes equal if they consist of the same set of points, the strong equality predicate (*strong_eq*) requires in addition that the negative halfspaces agree. In other words, two hyperplanes are strongly equal if their coefficient vectors are positive multiples of each other and they are (weakly) equal if their coefficient vectors are multiples of each other. Corresponding to the two equality predicates we have two linear orders: *compare* corresponds to weak equality and *strong_compare* corresponds to strong equality.

rat_hyperplane is an item type.

2. Creation

rat_hyperplane *h*(*int d* = 2);

introduces a variable h of type *rat_hyperplane* initialized to some hyperplane in d-dimensional space.

rat_hyperplane *h*(*integer_vector c*);

introduces a variable h of type *rat_hyperplane* initialized to the hyperplane with coefficients c.

rat_hyperplane *h*(*integer_vector c*, *integer D*);

introduces a variable h of type *rat_hyperplane* initialized to the hyperplane with coefficients (c, D).

rat_hyperplane *h*(*array*<*rat_point*>*P*, *rat_point o*, *int k* = 0);

constructs some hyperplane that passes through the points in P. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane.
Precondition: There must be a hyperplane passing through the points in P and if $k \neq 0$ then o must not lie on the constructed hyperplane

rat_hyperplane *h*(*rat_point p*, *rat_direction dir*, *rat_point o*, *int side* = 0);

constructs some hyperplane with normal direction *dir* that passes through p. If $k \in \{-1, +1\}$ then o is on k-side of the constructed hyperplane.
Precondition: If $k \neq 0$ then o must not lie on the constructed hyperplane

3. Operations

3.1 Initialization, Access and Evaluation

rat_hyperplane *rat_hyperplane*::d2(*rat_point p1*, *rat_point p2*, *rat_point o* = *rat_point*::*origin*( ), *int k* = 0)
returns a *rat_hyperplane* of dimension 2 that passes through the points *p1* and *p2*. If $k \in \{-1, +1\}$ then *o* is on *k*-side of the constructed hyperplane.
Precondition: If $k \neq 0$ then *o* must not lie on the constructed hyperplane

rat_hyperplane *rat_hyperplane*::d3(*rat_point p1*, *rat_point p2*, *rat_point p3*, *rat_point o* = *rat_point*::*origin*(3), *int k* = 0)
returns a *rat_hyperplane* of dimension 3 that passes through the points *p1*, *p2*, *p3*. If $k \in \{-1, +1\}$ then *o* is on *k*-side of the constructed hyperplane.
Precondition: If $k \neq 0$ then *o* must not lie on the constructed hyperplane

int *h*.dim( ) returns the dimension of *h*.

integer *h*[*int i*] returns the *i*-th coefficient of *h*.

integer_vector *h*.coefficent_vector( ) returns the coefficient vector $(c_0, \ldots, c_d)$ of *h*.

rat_vector *h*.normal_vector( ) returns the normal vector of *h*. It points from the negative halfspace into the positive halfspace and its homogeneous coordinates are $(c_0, \ldots, c_{d-1}, 1)$.

rat_direction *h*.normal_direction( ) returns the normal direction of *h*. It points from the negative halfspace into the positive halfspace.

integer *h*.value_at(*rat_point p*) returns the value of *h* at the point *p*, i.e., $\sum_{0 \leq i \leq d} h_i p_i$. Warning: this value depends on the particular representation of *h* and *p*.

int *h*.which_side(*rat_point p*)
returns the side of the hyperplane *h* containing *p*.

bool *h*.contains(*rat_point p*) return if the point p lies on the hyperplane *h*.

rat_hyperplane *h*.transform(*aff_transformation t*)
returns $t(h)$.

3.2 Tests

int strong_compare(*rat_hyperplane h1*, *rat_hyperplane h2*)
strong compare.

bool identical(*rat_hyperplane h1*, *rat_hyperplane h2*)
test for identity.

bool	$h1 \equiv h2$	test for equality.
bool	$h1$!= $h2$	test for inequality.
bool	strong_eq(*rat_hyperplane* $h1$, *rat_hyperplane* $h2$)	test for strong equality.

3.3 Input and Output

ostream&	*ostream&* $O \ll h$	writes the coefficients of h to output stream O.
istream&	*istream&* $I \gg$ *rat_hyperplane&* h	reads the coefficients of h from input stream I. This operator uses the current dimension of h.

4. Implementation

Hyperplanes are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on a hyperplane h take time $O(h.dim(\,))$. $dim(\,)$ and coordinate access take constant time. The space requirement is $O(h.dim(\,))$.

2. The Header File of class rat_hyperplane

The type rat_hyperplane is an item class with representation class geo_rep. It shares this representation class with points, vectors, and directions. We derive rat_hyperplane from handle_base and derive geo_rep from handle_rep. This gives us reference counting for free. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_hyperplane.h 2 ⟩ ≡
#ifndef LEDA_RAT_HYPERPLANE_H
#define LEDA_RAT_HYPERPLANE_H
#include "rat_point.h"
#include "geo_rep.h"
  class rat_hyperplane : public handle_base {
  geo_rep *ptr() const { return (geo_rep *) PTR; }
public:
  rat_hyperplane(int d = 2) { PTR = new geo_rep (d); }
  rat_hyperplane(const integer_vector &c) { PTR = new geo_rep (c); }
  rat_hyperplane(const integer_vector &c, integer_t D)
  { PTR = new geo_rep (c, D); }
  rat_hyperplane(const array⟨rat_point⟩ &P, const rat_point &o, int k = 0);
  rat_hyperplane(const rat_point &p, const rat_direction &dir, const rat_point
      &o, int side = 0);
  rat_hyperplane(const rat_hyperplane &p) : handle_base(p) { }
  rat_hyperplane(const handle_base &p) : handle_base(p) { }
  ~rat_hyperplane() { }
  rat_hyperplane &operator=(const rat_hyperplane &p)
  { handle_base::operator=(p); return *this; }
  static rat_hyperplane d2(const rat_point &p1, const rat_point &p2, const rat_point
      &o = rat_point::origin(), int k = 0);
  static rat_hyperplane d3(const rat_point &p1, const rat_point &p2, const rat_point
      &p3, const rat_point &o = rat_point::origin(3), int k = 0);
  int dim() const { return ptr()→dim; }
  integer_t operator [](int i) const { return ptr()→v[i]; }
  integer_t hcoord(int i) const { return ptr()→v[i]; }
  integer_vector coefficent_vector() const;
  rat_vector normal_vector() const;
  rat_direction normal_direction() const;
  integer_t value_at(const rat_point &p) const;
  int which_side(const rat_point &p) const;
  bool contains(const rat_point &p) const { return (value_at(p) ≡ 0); }
  rat_hyperplane transform(const aff_transformation &t) const
  { return rat_hyperplane(transpose(t.inverse().matrix()) * ptr()→ivec()); }
  int cmp(const rat_hyperplane &, const rat_hyperplane &) const;
  int strong_cmp(const rat_hyperplane &h1, const rat_hyperplane &h2) const;
  friend int strong_compare(const rat_hyperplane &h1, const rat_hyperplane &h2);
  friend bool identical(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.ptr() ≡ h2.ptr(); }
```

```
  friend bool operator≡(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.cmp(h1, h2) ≡ 0; }
  friend bool operator≠(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.cmp(h1, h2) ≠ 0; }
  friend bool strong_eq(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.strong_cmp(h1, h2) ≡ 0; }
  friend ostream &operator≪(ostream &O, const rat_hyperplane &h);
  friend istream &operator≫(istream &I, rat_hyperplane &h);
  };
  inline void Print(const rat_hyperplane &h, ostream &out) { out ≪ h; }
  inline void Read(rat_hyperplane &h, istream &in) { in ≫ h; }
  inline int compare(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.cmp(h1, h2); }
  inline int strong_compare(const rat_hyperplane &h1, const rat_hyperplane &h2)
  { return h1.strong_cmp(h1, h2); }
#endif
```

3. The Implementation of class rat_hyperplane

```
⟨ rat_hyperplane.c 3 ⟩ ≡
#include "rat_hyperplane.h"
  ⟨ initialization 4 ⟩
  ⟨ constructors 5 ⟩;
  ⟨ input and output 7 ⟩⟨ vectors 8 ⟩;
  ⟨ functions 12 ⟩;
  ⟨ compares 13 ⟩;
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_hyperplane rat_hyperplane::d2(const rat_point &p1, const rat_point
        &p2, const rat_point &o, int k)
  {
    array⟨rat_point⟩ P12(2);
    P12[0] = p1;
    P12[1] = p2;
    return rat_hyperplane(P12, o, k);
  }
  rat_hyperplane rat_hyperplane::d3(const rat_point &p1, const rat_point
        &p2, const rat_point &p3, const rat_point &o, int k)
  {
    array⟨rat_point⟩ P123(3);
    P123[0] = p1;
    P123[1] = p2;
    P123[2] = p3;
    return rat_hyperplane(P123, o, k);
  }
```

This code is used in chunk 3.

5. Constructors

We want to construct a hyperplane that passes through a set P of points in d-dimensional space and has a specified point o on a specified side. I simply have to find a vector n such that $p^T \cdot n = 0$ for every point in P. Afterwards I use the point o to normalize.

```
⟨ constructors 5 ⟩ ≡
  rat_hyperplane::rat_hyperplane(const array⟨rat_point⟩ &P, const rat_point &o, int
        side)
  {
    int pl = P.low( );
    int d = P[pl].dim( );  // we are in d-dimensional space
    int m = P.high( ) − pl + 1;  // P has m points
    integer_matrix A(d, d + 1);
    int i, j, k;
```

```
    for (i = 0; i < m; i++) {
      /* define i-th equation */
      for (j = 0; j ≤ d; j++) A(i, j) = P[pl + i].hcoord(j);   // j-th coord of i-th point
    }
    integer_vector non_triv_sol;
    if (¬homogeneous_linear_solver(A, non_triv_sol))
      error_handler(1, "rat_hyperplane::constructor: set P is full dimensional.");
    if (side ≡ 0) {
      PTR = new geo_rep (non_triv_sol);
      return;
    }
    integer_t sum = 0;
    for (i = 0; i ≤ d; i++) sum += non_triv_sol[i] * o.hcoord(i);
    if (sum ≡ 0) error_handler(0,
          "rat_hyperplane::constructor:cannot use o to determine side.");
    if (sign(sum) ≠ side) PTR = new geo_rep (−non_triv_sol);
    else PTR = new geo_rep (non_triv_sol);
  }
```

See also chunk 6.

This code is used in chunk 3.

6. Given a point p and a direction dir we want to construct a hyperplane with normal direction dir and passing through p. We set the coefficent vector equal to $(dir_0, \ldots, dir_{d-1}, D)$ for some unknown D and then use p to determine D. Note that D will be rational in general.

```
⟨ constructors 5 ⟩ +≡
  rat_hyperplane::rat_hyperplane(const rat_point &p, const rat_direction &dir, const
        rat_point &o, int side)
  {
    int d = p.dim( );
    integer_vector c(d + 1);
    integer_t sum = 0;
    for (int i = 0; i < d; i++) {
      sum += dir.hcoord(i) * p.hcoord(i);   // D is −sum/p.hcoord(d)
      c[i] = dir.hcoord(i) * p.hcoord(d);
    }
    c[d] = −sum;
    int s = 1;
    if (side ≠ 0) {
      integer_t sum = 0;
      for (int i = 0; i ≤ d; i++) sum += c[i] * o.hcoord(i);
      if (sign(sum) ≠ side) c = −c;
    }
    PTR = new geo_rep (c);
  }
```

7. Input and Output

We just take care that in case of input we don't overwrite a common representation. Apart from that we use the I/O-operations of geo_rep.

```
⟨ input and output 7 ⟩ ≡
  ostream &operator≪(ostream &out, const rat_hyperplane &h)
  {
    out ≪ h.ptr( );
    return out;
  }
  istream &operator≫(istream &in, rat_hyperplane &h)
  {
    int d = h.dim( );
    if (h.refs( ) > 1) h = rat_hyperplane(d);
    in ≫ h.ptr( );
    return in;
  }
```

This code is used in chunk 3.

8. Vectors

To deliver the coefficent vector is a trivial task.

```
⟨ vectors 8 ⟩ ≡
  integer_vector rat_hyperplane::coefficent_vector( ) const
  {
    integer_vector result(ptr( )→dim + 1);
    for (int i = 0; i ≤ ptr( )→dim; i++) result[i] = ptr( )→v[i];
    return result;
  }
```

See also chunks 9, 10, and 11.

This code is used in chunk 3.

9. Normal Vector.

Any multiple of $(c_0, \ldots, c_{d-1})$ is a normal vector. We want the vector to point from the negative to the positive halfspace. Recall that our hyperplane has the equation $c_d + \sum(c_i z_i) = 0$, where the $z[i]$ are Euclidian point coordinates. The point $z = -c_d \cdot c/\| c \|$ is on the hyperplane, the point $z_n = (-1 - c_d) \cdot c/\| c \|$ is in the negative halfspace and the point $z_p = (1 - c_d) \cdot c/\| c \|$ is in the positive halfspace. Thus any positive multiple of c is the desired normal vector. We take $(c_0, \ldots, c_{d-1}, 1)$.

```
⟨ vectors 8 ⟩ +≡
  rat_vector rat_hyperplane::normal_vector( ) const
  {
    int d = dim( );
    rat_vector res(d);
    res.ptr( )→copy(ptr( ));
    res.ptr( )→v[d] = 1;
    return res;
  }
```

10. Normal direction.

We take the normal vector and convert it to a direction.

```
⟨ vectors 8 ⟩ +≡
  rat_direction rat_hyperplane::normal_direction( ) const
  { return normal_vector( ).to_rat_direction( ); }
```

11. Functions

For *value_at* we just calculate the inner product $\sum_{0\le i\le d} h_i p_i$ of both representations.

```
⟨ vectors 8 ⟩ +≡
  integer_t rat_hyperplane::value_at(const rat_point &p) const
  {
    if (dim( ) ≠ p.dim( ))
      error_handler(1,"rat_hyperplane::value_at:␣dimensions␣disagree.");
    integer_t result = 0;
    for (int i = 0; i ≤ dim( ); i++) result += hcoord(i) * p.hcoord(i);
    return result;
  }
```

12. For *which_side* we compute $\sum_{0\le i\le d} h_i p_i$ and return its sign. Note that points and hyperplanes are invariant under positive multiples only.

```
⟨ functions 12 ⟩ ≡
  int rat_hyperplane::which_side(const rat_point &p) const
  {
    if (dim( ) ≠ p.dim( ))
      error_handler(1,"rat_hyperplane::which␣side:␣dimensions␣do␣not␣agree.");
    return sign(value_at(p));
  }
```

This code is used in chunk 3.

13. Weak Linear Order

Weak equality considers two hyperplanes equal if their coefficient vectors are multiples of each other. We define the weak linear order as the lexicographic order under weak equality. Let i be minimal such that either $h1_i$ or $h2_i$ is non-zero. We may assume that a non-zero value is positive (since we consider weak equality). Thus if exactly one of the vlaue is non-zero, we can decide the order right there: The vector with the entry zero is smaller. If both entries are non-zero, we compute scaling factors that make the i-th coefficients equal and positive and proceed.

```
⟨ compares 13 ⟩ ≡
  int rat_hyperplane::cmp(const rat_hyperplane &h1, const rat_hyperplane &h2)
        const
  {
    if (identical(h1, h2)) return 0;
    int i, c;
    int d = h1.dim( );
    for (i = 0; i ≤ d ∧ h1.hcoord(i) ≡ 0 ∧ h2.hcoord(i) ≡ 0; i++) ;  // no body
    if (h1.hcoord(i) ≡ 0) return −1;
    if (h2.hcoord(i) ≡ 0) return +1;
```

```
    int s = sign(h1.hcoord(i)) * sign(h2.hcoord(i));
    integer_t s1 = (integer_t) s * h2.hcoord(i);
    integer_t s2 = (integer_t) s * h1.hcoord(i);
    // s1 * h1.hcoord(i) is | h1.hcoord(i) * h2.hcoord(i) |
    i++;
    while (i ≤ d) {
      c = compare(s1 * h1.hcoord(i), s2 * h2.hcoord(i));
      if (c ≠ 0) return c;
      i++;
    }
    return 0;
  }
```

See also chunk 14.

This code is used in chunk 3.

14. Strong Linear Order

Strong equality considers two hyperplanes equal if their coefficient vectors are positive multiples of each other. We define the strong linear order as the lexicographic order under strong equality. Let i be minimal such that either $h1_i$ or $h2_i$ is non-zero. If the values have different signs we can decide the order right there: The vector with the smaller entry is smaller. If the entries have the same sign we compute positive scaling factors that make the i-th coefficients equal and proceed.

⟨ compares 13 ⟩ +≡

```
  int rat_hyperplane::strong_cmp(const rat_hyperplane &h1, const rat_hyperplane
          &h2) const
  {
    if (identical(h1, h2)) return 0;
    int i;
    int d = h1.dim( );
    for (i = 0; i ≤ d ∧ h1.hcoord(i) ≡ 0 ∧ h2.hcoord(i) ≡ 0; i++) ;   // no body
    int c1 = sign(h1.hcoord(i));
    int c2 = sign(h2.hcoord(i));
    if (c1 ≠ c2) return compare(c1, c2);
    integer_t s1 = (integer_t) sign(h2.hcoord(i)) * h2.hcoord(i);
    integer_t s2 = (integer_t) sign(h1.hcoord(i)) * h1.hcoord(i);
    i++;
    int c;
    while (i ≤ d) {
      c = compare(s1 * h1.hcoord(i), s2 * h2.hcoord(i));
      if (c ≠ 0) return c;
      i++;
    }
    return 0;
  }
```

15. A Test of class rat_hyperplane

We test the construction, access to the components and all the arithmetical operators.

```
⟨ rat_hyperplane-test.c 15 ⟩ ≡
#include "rat_hyperplane.h"
  main( )
  { /* construction and access */
    integer_vector vi1(4);
    integer_vector vi2(3);
    vi1[0] = 1; vi1[1] = 2; vi1[2] = 3; vi1[3] = 4; vi2[0] = −4; vi2[1] = −3; vi2[2] = −2;
    // two ivec inits
    rat_point p1 = rat_vector::unit(0, 3).to_rat_point( ), p2 = rat_vector::unit(1,
        3).to_rat_point( ), p3 = rat_vector::unit(2, 3).to_rat_point( );
    // one three point init
    rat_direction dir = rat_direction::d3(1, 1, 1);
    rat_point o = rat_point::d3(−1, −1, −1); // one dir init
    rat_hyperplane h0(vi1), h1(vi2, 1), h2 = rat_hyperplane::d3(p1, p2, p3, o, −1),
        h3 = rat_hyperplane::d3(rat_point::origin(3), rat_point::origin(3),
        rat_point::origin(3), o, −1), h4(p1, dir, o, 1), h5(3);
    // all kind of rat_hyperplane inits
    cout ≪ "three␣points␣p1,p2,p3:␣" ≪ p1 ≪ p2 ≪ p3;
    cout ≪ "\na␣direction␣dir:␣" ≪ dir;
    cout ≪ "\nfive␣hyperplanes␣h0,h1,h2(p1,p2,p3),h3(0,0,0),h4(p1,dir):\n";
    cout ≪ h0 ≪ h1 ≪ h2 ≪ h3 ≪ h4;
    cout ≪ "\nenter␣hyperplane␣h5:␣";
    cin ≫ h5;
    cout ≪ "access␣operations␣on␣h5:";
    cout ≪ "\ncoefficents:␣";
    for (int i = 0; i ≤ h5.dim( ); i++) cout ≪ h5[i] ≪ "␣";
    cout ≪ "\ncoefficent␣vector:␣" ≪ h5.coefficent_vector( );
    /* compares and other operations */
    cout ≪ "\ncompare(h0,h1)␣=␣" ≪ compare(h0, h1);
    cout ≪ "\ncompare(h2,h4)␣=␣" ≪ compare(h2, h4);
    cout ≪ "\nstrong_compare(h0,h1)␣=␣" ≪ strong_compare(h2, h4);
    cout ≪ "\nstrong_compare(h2,h4)␣=␣" ≪ strong_compare(h2, h4);
    cout ≪ "\nh2.normal_vector()␣=␣" ≪ h2.normal_vector( );
    cout ≪ "\nthree␣points␣p1,p2,p3␣which␣lie␣in␣h2:␣";
    cout ≪ p1 ≪ p2 ≪ p3;
    cout ≪ "\nh2.value_at(p1)(p2)(p3)␣=␣";
    cout ≪ h2.value_at(p1) ≪ "␣" ≪ h2.value_at(p2) ≪ "␣";
    cout ≪ h2.value_at(p3);
    cout ≪ "\nvector␣p1-p2␣=␣" ≪ p1 − p2;
    cout ≪ "\nh2.normal_vector()␣=␣" ≪ h2.normal_vector( );
    cout ≪ "\n(p1-p2)␣*␣h2.normal_vector()␣=␣";
    cout ≪ (p1 − p2) ∗ h2.normal_vector( );
    cout ≪ "\na␣reference␣point␣o:␣" ≪ o;
    cout ≪ "\nh2.which_side(o)␣=␣" ≪ h2.which_side(o);
    cout ≪ "\nh4.which_side(o)␣=␣" ≪ h4.which_side(o);
    cout ≪ "\n\n";
  }
```

Index

List of Refinements

Segments with Rational Coordinates in d-Space (class rat_segment)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_segment

1. Definition

An instance s of the data type *rat_segment* is a directed straight line segment connecting two rational points p and q. p is called the start or source point and q is called the target point of s, both points are called endpoints of s. A segment whose endpoints are equal is called *trivial*.

2. Creation

rat_segment s(*int* $d = 2$);

introduces a variable s of type *rat_segment* and initializes it to some segment in d-dimensional space.

rat_segment s(*rat_point* p, *rat_point* q);

introduces a variable s of type *rat_segment*. s is initialized to the segment (p, q).

rat_segment s(*integer* $x1$, *integer* $y1$, *integer* $x2$, *integer* $y2$);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1), (x2, y2)]$ in two-dimensional space.

3. Operations

3.1 Initialization, Access and Conversions

rat_segment *rat_segment*::d2(*integer* $x1$, *integer* $y1$, *integer* $D1$, *integer* $x2$, *integer* $y2$, *integer* $D2$)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_segment *rat_segment*::d2(*integer* $x1$, *integer* $y1$, *integer* $x2$, *integer* $y2$)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_segment *rat_segment*::d3(*integer* $x1$, *integer* $y1$, *integer* $z1$, *integer* $x2$, *integer* $y2$, *integer* $z2$)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_segment *rat_segment*::d3(*integer* $x1$, *integer* $y1$, *integer* $z1$, *integer* $D1$, *integer* $x2$, *integer* $y2$, *integer* $z2$, *integer* $D2$)

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int	*s*.dim( )	returns the dimension of the underlying space.
rat_point	*s*.source( )	returns the source point of segment *s*.
rat_point	*s*.point1( )	returns the source point of segment *s*.
rat_point	*s*.target( )	returns the target point of segment *s*.
rat_point	*s*.point2( )	returns the target point of segment *s*.
rational	*s*.coord1(*int i*)	returns the *i*-th cartesian coordinate of the source of *s*.
rational	*s*.coord2(*int i*)	returns the *i*-th cartesian coordinate of the target of *s*.
integer	*s*.hcoord1(*int i*)	returns the *i*-th homogeneous coordinate of the source of *s*.
integer	*s*.hcoord2(*int i*)	returns the *i*-th homogeneous coordinate of the target of *s*.
rat_segment	*s*.reverse( )	returns the segment (*target*( ), *source*( )).
rat_direction	*s*.direction( )	returns the direction of *s*. *Precondition*: *s* is non-trivial.
rat_vector	*s*.source_target_vector( )	returns the vector from source to target. *Precondition*: *s* is non-trivial.
rat_line	*s*.supporting_line( )	returns the supporting line of *s*. *Precondition*: *s* is non-trivial.
rat_segment	*s*.transform(*aff_transformation t*)	returns $t(s)$.

3.2 Tests and Calculations

bool	*s*.is_trivial( )	returns true if *s* is trivial.
bool	identical(*rat_segment s1*, *rat_segment s2*)	Test for identity.
bool	strong_eq(*rat_segment s1*, *rat_segment s2*)	Test for equality as oriented segments.
bool	$s \equiv t$	Test for equality as unoriented segments.
bool	s != t	Test for inequality.
bool	parallel(*rat_segment s1*, *rat_segment s2*)	Test if the supporting lines are parallel. *Precondition*: : *s1* and *s2* are not trivial.

bool *s*.contains(*rat_point p*)

returns true if p lies on s and false otherwise.

bool common_endpoint(*rat_segment s1*, *rat_segment s2*, *rat_point& common*)

if *s1* and *s2* touch in a common end point, this point is assigned to common and the result is true, otherwise the result is false.

int *s*.intersection(*rat_line t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ by the following means. The return value is one of the constants {*NO_I*, *PNT_I*, *SEG_I*, *RAY_I*, *LIN_I*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int *s*.intersection(*rat_ray t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ as above.

int *s*.intersection(*rat_segment t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $s \cap t$ as above.

3.3 Input and Output

ostream& *ostream& O* $\ll$ *s* writes the homogeneous coordinates of s to output stream O (in order source-target).

istream& *istream& I* $\gg$ *rat_segment& s*

reads the homogeneous coordinates of s from input stream I (in order source-target).

Additional Operations for segments in two-dimensional space

rational *s*.xcoord1( ) returns the x-coordinate of the source point of segment s.

rational *s*.xcoord2( ) returns the x-coordinate of the target point of segment s.

rational *s*.ycoord1( ) returns the y-coordinate of the source point of segment s.

rational *s*.ycoord2( ) returns the y-coordinate of the target point of segment s.

integer *s*.X1( ) returns the zeroth homogeneous coordinate of the start point of segment s.

integer *s*.X2( ) returns the zeroth homogeneous coordinate of the end point of segment s.

integer	*s*.Y1( )	returns the first homogeneous coordinate of the start point of segment s.
integer	*s*.Y2( )	returns the first homogeneous coordinate of the end point of segment s.
integer	*s*.W1( )	returns the homogenizing coordinate of the start point of segment s.
integer	*s*.W2( )	returns the homogenizing coordinate of the end point of segment s.
integer	*s*.dx( )	returns the normalized x-difference $X1 \cdot W2 - X2 \cdot W1$ of the segment.
integer	*s*.dy( )	returns the normalized y-difference $Y1 \cdot W2 - Y2 \cdot W1$ of the segment.
bool	*s*.vertical( )	returns true if s is vertical (or trivial) and false otherwise.
bool	*s*.horizontal( )	returns true if s is horizontal (or trivial) and false otherwise.
bool	*s*.intersection(*rat_segment t*, *rat_point*& *p*)	if s and t intersect in a single point the point of intersection is assigned to p and the result is true, otherwise the result is false. *Precondition:* The supporting lines are not parallel.
bool	*s*.intersection_of_lines(*rat_segment t*, *rat_point*& *p*)	if the lines supporting s and t are not parallel their point of intersection is assigned to p and the result is true, otherwise the result is false.
int	orientation(*rat_segment s*, *rat_point p*)	computes orientation(a, b, p), where a and b are the source and target of s respectively.
int	cmp_slopes(*rat_segment s1*, *rat_segment s2*)	returns *compare*(*slope*(*s1*), *slope*(*s2*)).

int cmp_at_line_defined_by(*rat_segment s1*, *rat_segment s2*, *rat_point r*)

Let L be the directed curve consisting of a vertical upward ray ending in $(r.xcoord(\,) + \epsilon^2, r.ycoord(\,) + \epsilon)$ followed by a horizontal segment ending in $(r.xcoord(\,) - \epsilon^2, r.ycoord(\,) + \epsilon)$,
followed by an upward vertical ray; here ϵ is a positive infinitesimal. If both segments are non-trivial then the result is the order of the two intersections along the line L. If at least one of the segments is trivial then the result is zero if one of the segments is contained in the other segment. If exactly one of the segments is trivial then the result is the position (above, on, or below) of this segment with respect to the other segment.

Precondition: For $i = 1, 2$ we have: if si is trivial then both endpoints are equal to r and if si is non-trivial then its smaller endpoint is less than or equal to r and its larger endpoint is larger than r.

bool intersection(*rat_segment s1*, *rat_segment s2*)

decides whether $s1$ and $s2$ intersect in one point when the supporting lines are not equal.

4. Implementation

Segments are implemented by a pair of points as an item type. All operations like creation, initialization, tests, the calculation of the direction and source-target vector, input and output on a segment s take time $O(s.dim(\,))$. $dim(\,)$, coordinate and end point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dim(\,))$. The space requirement is $O(s.dim(\,))$.

2. The Header File of class rat_segment

The type rat_segment is an item class with representation class geo_pair_rep. It shares this representation class with rat_line and rat_ray. We derive rat_segment from handle_base. By this representation scheme we obtain reference counting from the LEDA base classes. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_segment.h 2 ⟩ ≡
#ifndef LEDA_RAT_SEGMENT_H
#define LEDA_RAT_SEGMENT_H
#include "geo_pair_rep.h"
#include "rat_point.h"
#include "rat_line.h"
#include "rat_ray.h"
  class rat_segment : public handle_base {
  geo_pair_rep *ptr() const { return (geo_pair_rep *) PTR; }
  rat_segment(const handle_base &b) : handle_base(b) { }
  friend class rat_line;
  friend class rat_ray;
public:
  rat_segment(int d = 2)
  { PTR = new geo_pair_rep (rat_point(d), rat_point(d)); }
  rat_segment(const rat_point &p, const rat_point &q)
  { PTR = new geo_pair_rep (p, q); }
  rat_segment(const integer_t &x1, const integer_t &y1, const integer_t &x2, const
        integer_t &y2)
  { PTR = new geo_pair_rep (rat_point(x1, y1), rat_point(x2, y2)); }
  rat_segment(const rat_segment &s) : handle_base(s) { }
  ~rat_segment() { }
  rat_segment &operator=(const rat_segment &s)
  { handle_base::operator=(s); return *this; }
  static rat_segment d2(integer_t x1, integer_t y1, integer_t D1, integer_t
      x2, integer_t y2, integer_t D2);
  static rat_segment d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2);
  static rat_segment d3(integer_t x1, integer_t y1, integer_t z1, integer_t x2, integer_t
      y2, integer_t z2);
  static rat_segment d3(integer_t x1, integer_t y1, integer_t z1, integer_t
      D1, integer_t x2, integer_t y2, integer_t z2, integer_t D2);
  int dim() const { return (ptr()->source.dim()); }
  rat_point source() const { return ptr()->source; }
  rat_point point1() const { return ptr()->source; }
  rat_point start() const { return ptr()->source; }
  rat_point target() const { return ptr()->target; }
  rat_point point2() const { return ptr()->target; }
  rat_point end() const { return ptr()->target; }
  rational_t coord1(int i) const {
    return rational_t(ptr()->source.hcoord(i), ptr()->source.hcoord(dim())); }
```

```
rational_t coord2(int i) const {
  return rational_t(ptr()→target.hcoord(i), ptr()→target.hcoord(dim())); }
integer_t hcoord1(int i) const { return ptr()→source.hcoord(i); }
integer_t hcoord2(int i) const { return ptr()→target.hcoord(i); }
rat_segment reverse() const { return rat_segment(end(), start()); }
rat_direction direction() const;
rat_vector source_target_vector() const;
rat_line supporting_line() const;
rat_segment transform(const aff_transformation &t) const
{ return rat_segment(point1().transform(t), point2().transform(t)); }
bool is_trivial() const { return ptr()→source ≡ ptr()→target; }
friend bool identical(const rat_segment &s1, const rat_segment &s2);
friend bool strong_eq(const rat_segment &s1, const rat_segment &s2)
{ return (s1.source() ≡ s2.source() ∧ s1.target() ≡ s2.target()); }
bool operator≡(const rat_segment &t) const
{ return (strong_eq(*this, t) ∨ strong_eq(*this, t.reverse())); }
bool operator≠(const rat_segment &t) const
{ return ¬operator≡(t); }
friend bool parallel(const rat_segment &s1, const rat_segment &s2);
bool contains(const rat_point &p) const;
friend bool common_endpoint(const rat_segment &s1, const rat_segment
    &s2, rat_point &common);
int intersection(const rat_line &t, rat_point &i1, rat_point &i2) const;
int intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const;
int intersection(const rat_segment &t, rat_point &i1, rat_point &i2) const;
friend ostream &operator≪(ostream &O, const rat_segment &s);
friend istream &operator≫(istream &I, rat_segment &s);
rational xcoord1() const { return ptr()→source.xcoord(); }
rational xcoord2() const { return ptr()→target.xcoord(); }
rational ycoord1() const { return ptr()→source.ycoord(); }
rational ycoord2() const { return ptr()→target.ycoord(); }
integer_t X1() const { return ptr()→source.X(); }
integer_t X2() const { return ptr()→target.X(); }
integer_t Y1() const { return ptr()→source.Y(); }
integer_t Y2() const { return ptr()→target.Y(); }
integer_t W1() const { return ptr()→source.W(); }
integer_t W2() const { return ptr()→target.W(); }
integer_t dx() const { return ptr()→dx; }
integer_t dy() const { return ptr()→dy; }
bool vertical() const { return ptr()→dx ≡ 0; }
bool horizontal() const { return ptr()→dy ≡ 0; }
bool intersection(const rat_segment &t, rat_point &p) const;
bool intersection_of_lines(const rat_segment &t, rat_point &p) const
{ return ptr()→d2_intersection(*(t.ptr()), p); }
```

```
  friend int orientation(const rat_segment &s, const rat_point &p)
  { return orientation(s.source( ), s.target( ), p); }
  friend int cmp_slopes(const rat_segment &s1, const rat_segment &s2);
  friend int cmp_at_line_defined_by(const rat_segment &s1, const rat_segment
      &s2, const rat_point &r);
  friend bool intersection(const rat_segment &s1, const rat_segment &s2);
  int on_line_position(const rat_point &q) const;
  };
  inline void Print(const rat_segment &s, ostream &out) { out ≪ s; }
  inline void Read(rat_segment &s, istream &in) { in ≫ s; }
  inline bool identical(const rat_segment &s1, const rat_segment &s2)
  { return s1.ptr( ) ≡ s2.ptr( ); }
  inline bool parallel(const rat_segment &s1, const rat_segment &s2)
  { return s1.direction( ) ≡ s2.direction( ) ∨
        s1.direction( ) ≡ −s2.direction( ); }
  inline int cmp_slopes(const rat_segment &s1, const rat_segment &s2)
  { return sign(s1.dy( ) ∗ s2.dx( ) − s2.dy( ) ∗ s1.dx( )); }
  enum on_position {
    out_before, id_point1, in_between, id_point2, out_after
  };
#endif
```

3. The Implementation of class rat_segment

```
⟨ rat_segment.c 3 ⟩ ≡
#include "rat_segment.h"
  ⟨ initialization 4 ⟩
  ⟨ conversion functions 5 ⟩
  ⟨ the tests 6 ⟩
  ⟨ intersection operations 7 ⟩
  ⟨ input and output 14 ⟩
  ⟨ some 2-space operations 15 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_segment rat_segment::d2(integer_t x1, integer_t y1, integer_t D1, integer_t
        x2, integer_t y2, integer_t D2)
  { return rat_segment(rat_point::d2(x1, y1, D1), rat_point::d2(x2, y2, D2)); }
  rat_segment rat_segment::d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2)
  { return rat_segment(rat_point::d2(x1, y1), rat_point::d2(x2, y2)); }
  rat_segment rat_segment::d3(integer_t x1, integer_t y1, integer_t z1, integer_t
        x2, integer_t y2, integer_t z2)
  { return rat_segment(rat_point::d3(x1, y1, z1), rat_point::d3(x2, y2, z2)); }
  rat_segment rat_segment::d3(integer_t x1, integer_t y1, integer_t z1, integer_t
        D1, integer_t x2, integer_t y2, integer_t z2, integer_t D2)
  { return rat_segment(rat_point::d3(x1, y1, z1, D1), rat_point::d3(x2, y2, z2, D2)); }
```

This code is used in chunk 3.

5. Conversion Operations

```
⟨ conversion functions 5 ⟩ ≡
  rat_vector rat_segment::source_target_vector() const
  {
    if (is_trivial()) error_handler(1, "rat_segment::source_target_vector: triv\
        ial segment cannot be converted.");
    return ptr()→to_rat_vector();
  }
  rat_direction rat_segment::direction() const
  {
    if (is_trivial()) error_handler(1,
        "rat_segment::direction: trivial segment cannot be converted.");
    return ptr()→to_rat_direction();
  }
  rat_line rat_segment::supporting_line() const
  {
    if (is_trivial()) error_handler(1,
        "rat_segment::supporting_line: trivial segment cannot be converted.");
    return rat_line(*this);
  }
```

This code is used in chunk 3.

6. Containment Tests

```
⟨ the tests 6 ⟩ ≡
  bool rat_segment::contains(const rat_point &p) const
  {
    int d = dim( );
    rat_point s = start( );
    rat_point t = end( );
    integer_t lnum = (p.hcoord(0) * s.hcoord(d) − s.hcoord(0) * p.hcoord(d)) * t.hcoord(d);
    integer_t lden = (t.hcoord(0) * s.hcoord(d) − s.hcoord(0) * t.hcoord(d)) * p.hcoord(d);
    integer_t lnum_i, lden_i;
    if (lnum * lden < 0 ∨ abs(lnum) > abs(lden)) return false;
    for (int i = 1; i < d; i++) {
      lnum_i = (p.hcoord(i) * s.hcoord(d) − s.hcoord(i) * p.hcoord(d)) * t.hcoord(d);
      lden_i = (t.hcoord(i) * s.hcoord(d) − s.hcoord(i) * t.hcoord(d)) * p.hcoord(d);
      if (lnum * lden_i ≠ lnum_i * lden) return false;
    }
    return true;
  }
```

This code is used in chunk 3.

7. Intersection Operations

In this section we implement the intersection operation of a segment with all equal dimensional straight line objects (segments, rays, lines). If the segment is trivial, the check is done by the *constains* operation. If the supporting lines of both are parallel, they can be identical or different. In the first case we have to take a closer look, in the second case there's certainly no intersection. If the lines are not parallel, we use our d dimensional intersection routine of our **geo_pair_rep** class and determine with the help of the lambda reference parameters if in case of intersection the point lies on the segment, the ray or the line. For a segment $s = \overline{p_1 p_2}$ the routine delivers a λ such that the intersection point $i = p_1 + \lambda * (p_2 - p_1)$. Thus to be part of the segment the intersection point has to be a convex combination of the endpoints with $\lambda \in [0, 1]$. To be part of the ray through p_1 and p_2 starting in p_1 λ may not be negative. For the line there's nothing to check anyway.

```
⟨ intersection operations 7 ⟩ ≡
  int rat_segment::intersection(const rat_line &t, rat_point &i1, rat_point &i2) const
  {
    if (dim( ) ≠ t.dim( ))
      error_handler(1, "intersection: the dimensions of the objects must agree.");
    if (is_trivial( ))
      if (t.contains(point1( ))) {
        i1 = point1( );
        return PNT_I;
      }
      else return NO_I;  // now *this is not trivial
    if (parallel(supporting_line( ), t))
      if (t.contains(point1( ))) {
        i1 = point1( );
        i2 = point2( );
        return SEG_I;
```

```
    }
    else return NO_I;
  rational_t lambda1, lambda2;
  if (ptr()→d_intersection(*(t.ptr()), i1, lambda1, lambda2) ∧ 0 ≤ lambda1 ∧ lambda1 ≤ 1)
    return PNT_I;
  return NO_I;
}
```

See also chunks 8, 10, 12, and 13.

This code is used in chunk 3.

8. In this chunk we implement the intersection operation of a segment and a ray. Look above for more information.

```
⟨ intersection operations 7 ⟩ +≡
  int rat_segment::intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const
  {
    if (dim() ≠ t.dim())
      error_handler(1,"intersection: the dimensions of the objects must agree.");
    if (is_trivial())
      if (t.contains(point1())) {
        i1 = point1();
        return PNT_I;
      }
      else return NO_I;  // now *this is not trivial
    if (parallel(supporting_line(), t.supporting_line())) {
      if (t.supporting_line().contains(source()))
        ⟨ check segment *this and ray t on line position 9 ⟩
      else return NO_I;
    } // now *this and t are not parallel
    rational_t lambda1, lambda2;
    if (ptr()→d_intersection(*(t.ptr()), i1, lambda1,
          lambda2) ∧ 0 ≤ lambda1 ∧ lambda1 ≤ 1 ∧ 0 ≤ lambda2) return PNT_I;
    return NO_I;
  }
```

9. In this chunk we want to calculate the intersection set of a line segment $s =$ ***this** and a ray t which are part of a common line. There are three possibilities here: empty, a point or a segment (part of both segments. To calculate the result we take the following steps:

1. we orient s like t such that they have a common direction from *point1*( ) to *point2*( ).
2. we determine the relative position of *s.point1*( ) and *s.point2*( ) with respect to the points *t.point1*( ) and *t.point2*( ) in terms of the constants *out_before*, *id_point1*, *in_between*, *id_point2* and *out_after*. This is done in *on_line_position*( ).
3. by caseswitching over each of the five constants we determine the maybe start and end points *i1* and *i2* of the common segment. in certain situations we know that *s1* and *s2* must be disjoint and we return accordingly.
4. if there's a possible overlapping of s and t it can be a point set or a segment set. We determine this by comparing *i1* and *i2* afterwards.

⟨ check segment ∗**this** and ray t on line position 9 ⟩ ≡

```
{
  rat_point p1, p2, q1, q2;
  q1 = t.point1();
  q2 = t.point2();
  if (direction() ≡ t.direction()) {
    p1 = point1();
    p2 = point2();
  }
  else {
    p1 = point2();
    p2 = point1();
  }
  /* now we know that the segments p1p2 and q1q2 representing a ray starting in q1 have the
  same direction on the common supporting line */
  int p1_pos = rat_segment(q1, q2).on_line_position(p1);
  int p2_pos = rat_segment(q1, q2).on_line_position(p2);
  switch (p1_pos) {
  case out_before:
    i1 = q1; // maybe start
    break;
  case id_point1: case in_between: case id_point2: case out_after:
    i1 = p1; // certainly start
    break;
  }
  switch (p2_pos) {
  case out_before:
    return NO_I; // no overlapping possible
  case id_point1: case in_between: case id_point2: case out_after:
    i2 = p2; // certainly end
    break;
  }
  if (i1 ≡ i2) return PNT_I;
  else return SEG_I;
}
```

This code is used in chunk 8.

10. In this chunk we implement the intersection operation of two segments. Look above for more information.

⟨ intersection operations 7 ⟩ +≡

```
int rat_segment::intersection(const rat_segment &t, rat_point &i1, rat_point &i2)
        const
{
  if (dim() ≠ t.dim())
    error_handler(1,"intersection: the dimensions of the objects must agree.");
  if (is_trivial())
    if (t.contains(point1())) {
      i1 = point1();
```

```
      return PNT_I;
    }
    else return NO_I;
  if (t.is_trivial( ))
    if (contains(t.point1( ))) {
      i1 = t.point1( );
      return PNT_I;
    }
    else return NO_I;  // now *this is not trivial
  if (parallel(supporting_line( ), t.supporting_line( )))
    if (t.supporting_line( ).contains(source( )))
      ⟨ check segment *this and segment t on line position 11 ⟩
    else return NO_I;
  rational_t lambda1, lambda2;
  if (ptr( )→d_intersection(*(t.ptr( )), i1, lambda1,
        lambda2) ∧ 0 ≤ lambda1 ∧ lambda1 ≤ 1 ∧ 0 ≤ lambda2 ∧ lambda2 ≤ 1)
    return PNT_I;
  return NO_I;
}
```

11. In this chunk we want to calculate the intersection set of two line segments $s1 = *\mathbf{this}$ and $s2 = t$ which are part of a common line. There are three possibilities here: empty, a point or a segment (part of both segments. To calculate the result we take the following steps:

1. we orient s and t such that they have a common direction from *point1( )* to *point2( )*.
2. we determine the relative position of *s1.point1( )* and *s1.point2( )* with respect to the points *s2.point1( )* and *s2.point2( )* in terms of the constants *out_before*, *id_point1*, *in_between*, *id_point2* and *out_after*. This is done in *on_line_position( )*.
3. by caseswitching over each of the five constants we determine the maybe start and end points *i1* and *i2* of the common segment. in certain situations we know that *s1* and *s2* must be disjoint and we return accordingly.
4. if there's a possible overlapping of *s1* and *s2* it can be a point set or a segment set. We determine this by comparing *i1* and *i2* afterwards.

```
⟨ check segment *this and segment t on line position 11 ⟩ ≡
  {
    rat_point p1, p2, q1, q2;
    p1 = point1( );
    p2 = point2( );
    if (direction( ) ≡ t.direction( )) {
      q1 = t.point1( );
      q2 = t.point2( );
    }
    else {
      q1 = t.point2( );
      q2 = t.point1( );
    }
    /* now we know that the segments p1p2 and q1q2 have the same direction on the common
    supporting line */
```

```
    int p1_pos = rat_segment(q1, q2).on_line_position(p1);
    int p2_pos = rat_segment(q1, q2).on_line_position(p2);
    switch (p1_pos) {
    case out_before: i1 = q1; // maybe start
      break;
    case id_point1: case in_between: case id_point2:
      i1 = p1; // certainly start
      break;
    case out_after:
      return NO_I; // no overlapping possible
    }
    switch (p2_pos) {
    case out_before:
      return NO_I; // no overlapping possible
    case id_point1: case in_between: case id_point2:
      i2 = p2; // certainly end
      break;
    case out_after:
      i2 = q2; // maybe end
      break;
    }
    if (i1 ≡ i2) return PNT_I;
    else return SEG_I;
  }
```

This code is used in chunk 10.

12. In this chunk we provide an operation which allows us to examine the position of a point q on the supporting line of a segment $\overline{st}$. We know that $s.supporting\_line().contains(q) \equiv true$. But we want to find the position of the point in terms of the constants *out_before*, *id_point1*, *in_between*, *id_point2*, *out_after*. We just calculate the λ in the equation $q = s + \lambda * (t - s)$ And compare it to the interval $[0, 1]$.

```
⟨ intersection operations 7 ⟩ +≡
  int rat_segment::on_line_position(const rat_point &q) const
  {
    int d = dim();
    rat_point s = start();
    rat_point t = end();
    integer_t lnum = (q.hcoord(0) * s.hcoord(d) - s.hcoord(0) * q.hcoord(d)) * t.hcoord(d);
    integer_t lden = (t.hcoord(0) * s.hcoord(d) - s.hcoord(0) * t.hcoord(d)) * q.hcoord(d);
    if (lden < 0) {
      lnum = -lnum;
      lden = -lden;
    } // now lden ¿ 0
    if (lnum ≡ 0) return id_point1;
    if (lnum ≡ lden) return id_point2;
    if (lnum < 0) return out_before;
    if (lnum > lden) return out_after;
    return in_between;
```

```
}
```

13. Sometimes we want to check if two segments touch in a common endpoint.

```
⟨ intersection operations 7 ⟩ +≡
  bool common_endpoint(const rat_segment &s1, const rat_segment &s2, rat_point
        &common)
  {
    if (s1.start( ) ≡ s2.start( )) {
      common = s1.start( );
      return true;
    }
    if (s1.start( ) ≡ s2.end( )) {
      common = s1.start( );
      return true;
    }
    if (s1.end( ) ≡ s2.start( )) {
      common = s1.end( );
      return true;
    }
    if (s1.end( ) ≡ s2.end( )) {
      common = s1.end( );
      return true;
    }
    return false;
  }
```

14. Input and Output

```
⟨ input and output 14 ⟩ ≡
  ostream &operator≪(ostream &O, const rat_segment &s)
  {
    O ≪ s.ptr( );
    return O;
  }
  istream &operator≫(istream &I, rat_segment &s)
  {
    int d = s.dim( );
    if (s.refs( ) > 1) s = rat_segment(d);
    I ≫ s.ptr( );
    return I;
  }
```

This code is used in chunk 3.

15. 2d-Operations

⟨ some 2-space operations 15 ⟩ ≡

```
bool intersection(const rat_segment &s1, const rat_segment &s2)
{ /* decides whether the segments s1 and s2 intersect. */
  int o1 = orientation(s1, s2.start( ));
  int o2 = orientation(s1, s2.end( ));
  int o3 = orientation(s2, s1.start( ));
  int o4 = orientation(s2, s1.end( ));
  return (o1 ≠ o2 ∧ o3 ≠ o4);
}
bool rat_segment::intersection(const rat_segment &t, rat_point &p) const
{ /* decides whether t and this segment intersect and, if so, returns the intersection in p. It
     is assumed that both segments have non-zero length */
  if (¬::intersection(*this, t)) return false;
  return ptr( )→d2_intersection(*(t.ptr( )), p);
}
int cmp_at_line_defined_by(const rat_segment &s1, const rat_segment &s2, const
        rat_point &r)
{
  error_handler(0, "cmp_at_line_defined_by not yet implemented.");
  return 0;
}
```

This code is used in chunk 3.

16. A Test of class rat_segment

```
⟨ rat_segment-test.c 16 ⟩ ≡
#include "rat_segment.h"
  main( )
  {
    ⟨ 2d tests 17 ⟩
    ⟨ 3d tests 18 ⟩
  }
```

17. In this chunk we test the special 2d procedures.

```
⟨ 2d tests 17 ⟩ ≡
  { /* some construction test */
    cout ≪ "\n2-SPACE␣MODULE:";
    rat_point p1(−5, 1), p2(5, 1);
    rat_segment s1(p1, p2), s2 = rat_segment::d2(−1, −5, −1, 5), s3;
    cout ≪ "\ntwo␣segments␣s1,s2:\n" ≪ s1 ≪ s2;
    cout ≪ "\ns1.horizontal()␣=␣" ≪ s1.horizontal( );
    cout ≪ "\ns2.vertical()␣=␣" ≪ s2.vertical( );
    /* some input and access test */
    cout ≪ "\nenter␣segment␣s3:␣";
    cin ≫ s3;
    cout ≪ "segment␣access␣ops:\n";
    cout ≪ s3.X1( ) ≪ "␣" ≪ s3.Y1( ) ≪ "␣" ≪ s3.W1( ) ≪ "␣";
    cout ≪ s3.X2( ) ≪ "␣" ≪ s3.Y2( ) ≪ "␣" ≪ s3.W2( ) ≪ "\n";
    cout ≪ s3.xcoord1( ) ≪ "␣" ≪ s3.ycoord1( ) ≪ "␣";
    cout ≪ s3.xcoord2( ) ≪ "␣" ≪ s3.ycoord2( );
    /* intersection and orientation */
    rat_point ipnt1, ipnt2;
    cout ≪ "\ns1.intersection(s3,ipnt1)␣=␣" ≪ s1.intersection(s3, ipnt1);
    cout ≪ "\nwith␣ipnt1␣=␣" ≪ ipnt1;
    cout ≪ "\ns1.intersection(s3,ipnt1,ipnt2)␣=␣";
    cout ≪ s1.intersection(s3, ipnt1, ipnt2);
    cout ≪ "\nwith␣ipnt1/2␣=␣" ≪ ipnt1 ≪ ipnt2;
    cout ≪ "\ncmp_at_line_defined_by(s1,s3,ipnt1)␣=␣";
    cout ≪ cmp_at_line_defined_by(s1, s3, ipnt1);
    cout ≪ "\ns1.intersection_of_lines(s3,ipnt1)␣=␣";
    cout ≪ s1.intersection_of_lines(s3, ipnt1);
    cout ≪ "\nwith␣ipnt1␣=␣" ≪ ipnt1;
    cout ≪ "\norientation(s1,ipnt1)␣=␣" ≪ orientation(s1, ipnt1);
    cout ≪ "\norientation(s3,ipnt1)␣=␣" ≪ orientation(s3, ipnt1);
    cout ≪ "\ncmp_slopes(s1,s3)␣=␣" ≪ cmp_slopes(s1, s3);
    cout ≪ "\n";
  }
```

This code is used in chunk 16.

18. Next we do some tests in 3-space. Here we test the d-dimensional components.

⟨ 3d tests 18 ⟩ ≡

```
{
/* some construction test */
cout ≪ "\nd-SPACE␣MODULE:␣(d=3)";
rat_point p0 = rat_point::d3(1,1,1), p(3), q(3), ipnt1(3), ipnt2(3); rat_vector e1 = 5
*rat_vector::unit(0,3);  // the first unit vector
rat_point p1(p0 + e1);
rat_segment s1(p0, p1), s2 = rat_segment::d3(5,1,1,5,5,5), s3(s1.reverse());
cout ≪ "\nthree␣segments␣s1,s2,s3:\n" ≪ s1 ≪ s2 ≪ s3;
/* some equality and containment tests */
cout ≪ "\n(s1␣==␣s3)␣=␣" ≪ (s1 ≡ s3);
cout ≪ "\nstrong_eq(s1,s3)␣=␣" ≪ strong_eq(s1, s3);
cout ≪ "\na␣point␣p␣=␣" ≪ (p = p0 + e1/2);
cout ≪ "\ns1.contains(p)␣=␣" ≪ s1.contains(p);
rat_point cpnt(3);
cout ≪ "\ncommon_endpoint(s1,s2,cpnt)␣=␣";
cout ≪ common_endpoint(s1, s2, cpnt);
cout ≪ "\nwith␣cpnt␣=␣" ≪ cpnt;
cout ≪ "\nparallel(s1,s3)␣=␣" ≪ parallel(s1, s3);
/* input */
cout ≪ "\nenter␣two␣points␣to␣define␣a␣line␣object:␣";
cin ≫ p ≫ q;
rat_segment s4(p, q);
cout ≪ "\nsegment␣s4␣(we␣test␣against␣s1):␣" ≪ s4;
cout ≪ "\naccess␣operations␣on␣s4:";
cout ≪ "\ncartesian:␣";
int i;
for (i = 0; i < s4.dim(); i++) cout ≪ s4.coord1(i) ≪ "␣";
for (i = 0; i < s4.dim(); i++) cout ≪ s4.coord2(i) ≪ "␣";
cout ≪ "\nhomogenous:␣";
for (i = 0; i ≤ s4.dim(); i++) cout ≪ s4.hcoord1(i) ≪ "␣";
for (i = 0; i ≤ s4.dim(); i++) cout ≪ s4.hcoord2(i) ≪ "␣";
cout ≪ "\nsegment␣between:␣" ≪ s4.point1() ≪ s4.point2();
cout ≪ "\ns4.is_trivial()␣=␣" ≪ s4.is_trivial();
if (¬s4.is_trivial()) cout ≪ "\nparallel(s1,s4)␣=␣" ≪ parallel(s1, s4);
cout ≪ "\ns1.intersection(s4,ipnt1,ipnt2)␣=␣";
cout ≪ s1.intersection(s4, ipnt1, ipnt2);
cout ≪ "\nwith␣ipnt1/2␣=␣" ≪ ipnt1 ≪ ipnt2;
rat_ray r(p, q);
cout ≪ "\nray␣r␣(we␣test␣against␣s1):␣" ≪ r;
cout ≪ "\ns1.intersection(r,ipnt1,ipnt2)␣=␣";
cout ≪ s1.intersection(r, ipnt1, ipnt2);
cout ≪ "\nwith␣ipnt1/2␣=␣" ≪ ipnt1 ≪ ipnt2;
rat_line l(p, q);
cout ≪ "\nline␣l␣(we␣test␣against␣s1):␣" ≪ l;
cout ≪ "\ns1.intersection(l,ipnt1,ipnt2)␣=␣";
cout ≪ s1.intersection(l, ipnt1, ipnt2);
cout ≪ "\nwith␣ipnt1/2␣=␣" ≪ ipnt1 ≪ ipnt2;
```

cout $\ll$ "\n\n"; }

This code is used in chunk 16.

Index

List of Refinements

⟨ 2d tests 17 ⟩ Used in chunk 16.
⟨ 3d tests 18 ⟩ Used in chunk 16.
⟨ check segment ∗**this** and ray t on line position 9 ⟩ Used in chunk 8.
⟨ check segment ∗**this** and segment t on line position 11 ⟩ Used in chunk 10.
⟨ conversion functions 5 ⟩ Used in chunk 3.
⟨ initialization 4 ⟩ Used in chunk 3.
⟨ input and output 14 ⟩ Used in chunk 3.
⟨ intersection operations 7, 8, 10, 12, 13 ⟩ Used in chunk 3.
⟨ `rat_segment-test.c` 16 ⟩
⟨ `rat_segment.c` 3 ⟩
⟨ `rat_segment.h` 2 ⟩
⟨ some 2-space operations 15 ⟩ Used in chunk 3.
⟨ the tests 6 ⟩ Used in chunk 3.

List of Refinements

⟨ arithmetic operations 7, 8, 9 ⟩ Used in chunk 3.
⟨ conversions 5 ⟩ Used in chunk 3.
⟨ initialization 4 ⟩ Used in chunk 3.
⟨ input and output 6 ⟩ Used in chunk 3.
⟨ linear operations 11, 12, 13, 14 ⟩ Used in chunk 3.
⟨ `rat_vector-test.c` 15 ⟩
⟨ `rat_vector.c` 3 ⟩
⟨ `rat_vector.h` 2 ⟩

Rays with Rational Coordinates in d-Space (class rat_ray)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_ray

1. Definition

An instance of data type *rat_ray* is a ray in d-dimensional Euclidian space. *rat_ray* is an item type.

2. Creation

rat_ray r(*int* $d = 2$); introduces a ray in d-dimensional space

rat_ray r(*rat_point* p, *rat_point* q);
introduces a ray through p and q and starting at p.
Precondition: p and q are distinct and have the same dimension.

rat_ray r(*rat_point* p, *rat_direction* *dir*);
introduces a ray starting in p with direction *dir*.
Precondition: p and *dir* have the same dimension.

rat_ray r(*rat_segment* s);
introduces a ray through *s.source*( ) and *s.target*( ) and starting at *s.source*( ).
Precondition: s is not trivial.

3. Operations

3.1 Initialization, Access and Conversions

rat_ray *rat_ray*::d2(*integer x1*, *integer y1*, *integer D1*, *integer x2*, *integer y2*, *integer D2*)
introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_ray *rat_ray*::d2(*integer x1*, *integer y1*, *integer x2*, *integer y2*)
introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_ray *rat_ray*::d3(*integer x1*, *integer y1*, *integer z1*, *integer x2*, *integer y2*, *integer z2*)
introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_ray *rat_ray*::d3(*integer x1*, *integer y1*, *integer z1*, *integer D1*, *integer x2*, *integer y2*, *integer z2*, *integer D2*)
introduces a variable r of type *rat_ray*. r is initialized to the ray $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int r.dim( ) returns the dimension of the underlying space.

rat_point	r.source( )	returns the source point of r.
rat_point	r.point1( )	returns the source point of r.
rat_point	r.point2( )	returns a point on r distinct from r.*source*( ).
rat_direction	r.direction( )	returns the direction of r.
rat_line	r.supporting_line( )	returns the supporting line of r.
rat_ray	r.transform(*aff_transformation t*)	returns $t(l)$.

3.2 Tests and Calculations

bool	identical(*rat_ray r1*, *rat_ray r2*)	test for identity.
bool	*r1* $\equiv$ *r2*	test for equality
bool	*r1* != *r2*	test for inequality.
int	parallel(*rat_ray r1*, *rat_ray r2*)	returns true if *r1* and *r2* are parallel and false otherwise.
bool	r.contains(*rat_point p*)	returns true if p lies on r.
bool	r.contains(*rat_segment s*)	returns true if s is part of r and false otherwise.

3.3 Intersection Calculations

bool	r.intersection(*rat_hyperplane h*, *rat_point*& *p*)	returns true if h and r intersect in a single point and false otherwise. In the first case the point of intersection is assigned to p.
rat_point	r.intersection(*rat_hyperplane h*)	returns the intersection of the hyperplane h with the ray r *Precondition*: h and r intersect in a single point.

int r.intersection(*rat_line t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $r \cap t$ by the following means. The return value is one of the constants {*NO_I*, *PNT_I*, *SEG_I*, *RAY_I*, *LIN_I*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int r.intersection(*rat_ray t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $r \cap t$ as above.

int r.intersection(*rat_segment t*, *rat_point& i1*, *rat_point& i2*)

returns the intersection set $r \cap t$ as above.

3.4 Input and Output

ostream& *ostream& O* $\ll$ *r* writes the coefficients of r to output stream O.

istream& *istream& I* $\gg$ *rat_ray& r*

reads the coefficients of r from input stream I. This operator uses the current dimension of r.

Additional Operations for rays in two-dimensional space

bool r.vertical( ) returns true if r is vertical or trivial and false otherwise.

bool r.horizontal( ) returns true if r is horizontal or trivial and false otherwise.

bool r.intersection(*rat_ray l1*, *rat_point& p*)

if r and l_1 are not parallel the point of intersection is assigned to p and the result is true, otherwise the result is false.

int orientation(*rat_ray r*, *rat_point p*)

computes orientation(a, b, p), where a and b are *source* and *another_point* of r respectively.

4. Implementation

Rays are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a ray r take time $O(r.dim(\,))$. $dim(\,)$, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dim(\,))$. The space requirement is $O(v.dim(\,))$.

2. The Header File of class rat_ray

The type rat_ray is an item class with representation class geo_pair_rep. It shares this representation class with rat_segment and rat_line. We derive rat_ray from handle_base. By this representation scheme we obtain reference counting from the LEDA base classes. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_ray.h 2 ⟩ ≡
#ifndef LEDA_RAT_RAY_H
#define LEDA_RAT_RAY_H
#include "geo_pair_rep.h"
#include "rat_point.h"
#include "rat_line.h"
#include "rat_segment.h"
  class rat_ray : public handle_base {
  geo_pair_rep *ptr() const { return (geo_pair_rep *) PTR; }
  rat_ray(const handle_base &b) : handle_base(b) {}
  friend class rat_line;
  friend class rat_segment;
public:
  rat_ray(int d = 2)
  { PTR = new geo_pair_rep (rat_point(d), rat_point(d)); }
  rat_ray(const rat_point &p, const rat_point &q)
  {
    if (p ≡ q)
      error_handler(1, "rat_ray::constructor: the two points must be different.");
    PTR = new geo_pair_rep (p, q);
  }
  rat_ray(const rat_point &p, const rat_direction &dir)
  {
    rat_point q = p + dir.to_rat_vector();
    PTR = new geo_pair_rep (p, q);
  }
  rat_ray(const rat_segment &s);
  rat_ray(const rat_ray &p) : handle_base(p) {}
  ~rat_ray() {}
  rat_ray &operator=(const rat_ray &p)
  { handle_base::operator=(p); return *this; }
  static rat_ray d2(integer_t x1, integer_t y1, integer_t D1, integer_t x2, integer_t
      y2, integer_t D2);
  static rat_ray d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2);
  static rat_ray d3(integer_t x1, integer_t y1, integer_t z1, integer_t x2, integer_t
      y2, integer_t z2);
  static rat_ray d3(integer_t x1, integer_t y1, integer_t z1, integer_t D1, integer_t
      x2, integer_t y2, integer_t z2, integer_t D2);
  int dim() const { return ptr()→source.dim(); }
  rat_point source() const { return ptr()→source; }
```

```
  rat_point point1( ) const { return ptr( )→source; }
  rat_point point2( ) const { return ptr( )→target; }
  rat_direction direction( ) const;
  rat_line supporting_line( ) const;
  rat_ray transform(const aff_transformation &t) const
  { return rat_ray(point1( ).transform(t), point2( ).transform(t)); }
  friend bool identical(const rat_ray &r1, const rat_ray &r2)
  { return r1.ptr( ) ≡ r2.ptr( ); }
  friend bool operator≡(const rat_ray &r1, const rat_ray &r2)
  { return r1.source( ) ≡ r2.source( ) ∧
        r1.direction( ) ≡ r2.direction( ); }
  friend bool operator≠(const rat_ray &r1, const rat_ray &r2)
  { return (¬(r1 ≡ r2)); }
  friend int parallel(const rat_ray &r1, const rat_ray &r2);
  bool contains(const rat_point &p) const;
  bool contains(const rat_segment &s) const;
  friend int cmp(const rat_ray &, const rat_ray &)
  { error_handler(1, "not␣implemented"); return 0; }
  bool intersection(const rat_hyperplane &h, rat_point &p);
  rat_point intersection(const rat_hyperplane &h);
  int intersection(const rat_line &t, rat_point &i1, rat_point &i2) const;
  int intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const;
  int intersection(const rat_segment &t, rat_point &i1, rat_point &i2) const;
  friend ostream &operator≪(ostream &O, const rat_ray &r);
  friend istream &operator≫(istream &I, rat_ray &r);
  bool vertical( ) const { return ptr( )→dx ≡ 0; }
  bool horizontal( ) const { return ptr( )→dy ≡ 0; }
  bool intersection(const rat_ray &l1, rat_point &p) const;
  friend int orientation(const rat_ray &r, const rat_point &p);
  } ;
  inline void Print(const rat_ray &l, ostream &out) { out ≪ l; }
  inline void Read(rat_ray &l, istream &in) { in ≫ l; }
  inline int orientation(const rat_ray &r, const rat_point &p)
  { return orientation(r.source( ), r.point2( ), p); }
  inline int parallel(const rat_ray &r1, const rat_ray &r2)
  { return (r1.direction( ) ≡ r2.direction( )) ∨
        (r1.direction( ) ≡ −(r2.direction( ))); }
#endif
```

3. The Implementation of class rat_ray

```
⟨ rat_ray.c 3 ⟩ ≡
#include "rat_ray.h"
  ⟨ initialization 4 ⟩
  ⟨ conversion operations 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ intersection operations 8 ⟩
  ⟨ the containment test 7 ⟩
  ⟨ some 2-dim operations 13 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_ray rat_ray::d2(integer_t x1, integer_t y1, integer_t D1, integer_t x2, integer_t
      y2, integer_t D2)
  { return rat_ray(rat_point::d2(x1, y1, D1), rat_point::d2(x2, y2, D2)); }
  rat_ray rat_ray::d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2)
  { return rat_ray(rat_point::d2(x1, y1), rat_point::d2(x2, y2)); }
  rat_ray rat_ray::d3(integer_t x1, integer_t y1, integer_t z1, integer_t x2, integer_t
      y2, integer_t z2)
  { return rat_ray(rat_point::d3(x1, y1, z1), rat_point::d3(x2, y2, z2)); }
  rat_ray rat_ray::d3(integer_t x1, integer_t y1, integer_t z1, integer_t D1, integer_t
      x2, integer_t y2, integer_t z2, integer_t D2)
  { return rat_ray(rat_point::d3(x1, y1, z1, D1), rat_point::d3(x2, y2, z2, D2)); }
```

This code is used in chunk 3.

5. Conversion Operations

To determine the direction of the line we just call the corresponding operation of the representation class. To transform the ray into a line we use the normally hidden copy construction.

```
⟨ conversion operations 5 ⟩ ≡
  rat_direction rat_ray::direction() const
  { return ptr()→to_rat_direction(); }
  rat_line rat_ray::supporting_line() const
  { return rat_line(*this); }
  rat_ray::rat_ray(const rat_segment &s) : handle_base(s) { }
```

This code is used in chunk 3.

6. Input and Output

We just use the input and output routines of our base class geo_pair_rep. Note that in case of input we have to take care not to overwrite a shared storage object.

```
⟨ input and output 6 ⟩ ≡
  ostream &operator≪(ostream &O, const rat_ray &r)
  {
    O ≪ r.ptr();
    return O;
  }
```

```
istream &operator≫(istream &I, rat_ray &r)
{
  int d = r.dim( );
  if (r.refs( ) > 1) r = rat_ray(d);
  I ≫ r.ptr( );
  if (r.source( ) ≡ r.point2( ))
    error_handler(1, "operator<<: defining points of ray must be different.");
  return I;
}
```

This code is used in chunk 3.

7. Containment

A ray r starting in point s and going through t contains a point p if there is a non-negative λ such that

$$\frac{p_i}{p_d} - \frac{s_i}{s_d} = \lambda \cdot (\frac{t_i}{t_d} - \frac{s_i}{s_d})$$

for all i. In other words, the numbers $(p_i s_d - s_i p_d)t_d / p_d(t_i s_d - s_i t_d)$ must all agree and be non-negative.

⟨ the containment test 7 ⟩ ≡

```
bool rat_ray::contains(const rat_point &p) const
{
  int d = dim( );
  rat_point s = source( );
  rat_point t = point2( );
  integer_t lnum = (p.hcoord(0) * s.hcoord(d) - s.hcoord(0) * p.hcoord(d)) * t.hcoord(d);
  integer_t lden = (t.hcoord(0) * s.hcoord(d) - s.hcoord(0) * t.hcoord(d)) * p.hcoord(d);
  integer_t lnum_i, lden_i;
  if (lnum * lden < 0) return false;
  for (int i = 1; i < d; i++) {
    lnum_i = (p.hcoord(i) * s.hcoord(d) - s.hcoord(i) * p.hcoord(d)) * t.hcoord(d);
    lden_i = (t.hcoord(i) * s.hcoord(d) - s.hcoord(i) * t.hcoord(d)) * p.hcoord(d);
    if (lnum * lden_i ≠ lnum_i * lden) return false;
  }
  return true;
}
bool rat_ray::contains(const rat_segment &s) const
{ return contains(s.start( )) ∧ contains(s.end( )); }
```

This code is used in chunk 3.

8. Intersection Calculation

To calculate the intersection point of a ray and a hyperplane we use our intersection procedure of the line case and check afterwards if our intersection point is contained in the ray.

⟨ intersection operations 8 ⟩ ≡

```
bool rat_ray::intersection(const rat_hyperplane &h, rat_point &p)
{
  if (supporting_line( ).intersection(h, p) ∧ contains(p))
    return true;
```

```
  return false;
}
rat_point rat_ray::intersection(const rat_hyperplane &h)
{
  rat_point p(dim());
  if (¬intersection(h, p))
    error_handler(1, "intersection: h and *this must intersect.");
  return p;
}
```

See also chunks 9, 10, and 12.

This code is used in chunk 3.

9. In this chunk we implement the intersection operation of a ray with a line. If the supporting line of the ray and the line are parallel, they can be identical or different. In the first case the ray is just a subset of the line, in the second case there's certainly no intersection. If the lines are not parallel, we use our d dimensional intersection routine of our **geo_pair_rep** class and determine with the help of the lambda reference parameters if in case of intersection the point lies on the ray. For a segment $s = \overline{p_1 p_2}$ the routine delivers a λ such that the intersection point $i = p_1 + \lambda * (p_2 - p_1)$. To be part of the ray through p_1 and p_2 starting in p_1 λ may not be negative. For the line there's nothing to check anyway.

⟨ intersection operations 8 ⟩ +≡

```
int rat_ray::intersection(const rat_line &t, rat_point &i1, rat_point &i2) const
{
  if (dim() ≠ t.dim())
    error_handler(1, "intersection: the dimensions of the objects must agree.");
  if (parallel(supporting_line(), t))
    if (t.contains(point1())) {
      i1 = point1();
      i2 = point2();
      return RAY_I;
    }
    else return NO_I;
  rational_t lambda1, lambda2;
  if (ptr()¬d_intersection(*(t.ptr()), i1, lambda1, lambda2) ∧ 0 ≤ lambda1) return PNT_I;
  return NO_I;
}
```

10. In this chunk we implement the intersection operation of two rays. If the supporting lines of the rays are parallel, they can be identical or different. In the first case we have to take a closer look, in the second case there's certainly no intersection. If the lines are not parallel, we use our d dimensional intersection routine of our **geo_pair_rep** class and determine with the help of the lambda reference parameters if in case of intersection the point lies on the ray. For a segment $s = \overline{p_1 p_2}$ the routine delivers a λ such that the intersection point $i = p_1 + \lambda * (p_2 - p_1)$. To be part of the ray through p_1 and p_2 starting in p_1 λ may not be negative.

⟨ intersection operations 8 ⟩ +≡

```
int rat_ray::intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const
{
```

```
  if (dim( ) ≠ t.dim( ))
    error_handler(1, "intersection: the dimensions of the objects must agree.");
  if (parallel(supporting_line( ), t.supporting_line( ))) {
    if (t.supporting_line( ).contains(source( )))
      ⟨ check ray *this and ray t on line position 11 ⟩
    else return NO_I;
  } // now *this and t are not parallel
  rational_t lambda1, lambda2;
  if (ptr( )→d_intersection(*(t.ptr( )), i1, lambda1, lambda2) ∧ 0 ≤ lambda1 ∧ 0 ≤ lambda2)
    return PNT_I;
  return NO_I;
}
```

11. In this chunk we want to calculate the intersection set of two rays $s = *\mathbf{this}$ and t which are part of a common line. There are four possibilities here: empty, a point or a segment (part of both rays or a ray. To calculate the result we have to check if the rays are equally oriented or not. If they'r equally oriented, one is a subset of the other. If their directions are the opposite of each other we have to check $t.source( )$ against s and decide the result.

```
⟨ check ray *this and ray t on line position 11 ⟩ ≡
{
  rat_point p1 = point1( );
  rat_point p2 = point2( );
  int t_source_pos = rat_segment(p1, p2).on_line_position(t.source( ));
  if (direction( ) ≡ t.direction( )) {
    switch (t_source_pos) {
    case out_before: case id_point1:
      i1 = p1;
      i2 = p2;
      break;
    case in_between: case id_point2: case out_after:
      i1 = t.point1( );
      i2 = t.point2( );
      break;
    }
    return RAY_I;
  }
  else // the directions are opposed to each other
  {
    switch (t_source_pos) {
    case out_before:
      return NO_I;
    case id_point1:
      i1 = p1;
      return PNT_I;
    case in_between: case id_point2: case out_after:
      i1 = p1;
      i2 = t.point1( );
      return SEG_I;
```

```
    }
  }
}
```
This code is used in chunk 10.

12. The rest is done by a call to the segment class member.

```
⟨ intersection operations 8 ⟩ +≡
  int rat_ray::intersection(const rat_segment &t, rat_point &i1, rat_point &i2) const
  { return t.intersection(*this, i1, i2); }
```

13. Special Operations in 2-Space

In this chunk we implement necessary operations for rays in the plane.

```
⟨ some 2-dim operations 13 ⟩ ≡
  bool rat_ray::intersection(const rat_ray &r1, rat_point &p) const
  { /* The intersection routine is hidden in the class geo_pair_rep */
    bool result = ptr()->d2_intersection(*(r1.ptr()), p);
    return result ∧ contains(p) ∧ r1.contains(p);
  }
```
This code is used in chunk 3.

14. A Test of class rat_ray

```
⟨ rat_ray-test.c 14 ⟩ ≡
#include "rat_ray.h"
  main( )
  {
    ⟨ 2d tests 15 ⟩
    ⟨ 3d tests 16 ⟩
  }
```

15. In this chunk we test the special 2d procedures.

```
⟨ 2d tests 15 ⟩ ≡
  { /* some construction test */
    cout ≪ "\n2-SPACE␣MODULE:";
    rat_point p1(1, 1), p2(−2, 1), p3(1, −1);
    rat_direction dir(1, 1, 1);
    rat_ray r1(p1, p2), r2(p2, p3), r3(p1, dir), r4;
    cout ≪ "\nthree␣rays␣r1,r2,r3:\n" ≪ r1 ≪ r2 ≪ r3;
    cout ≪ "\nr1.horizontal()␣=␣" ≪ r1.horizontal( );
    cout ≪ "\nr2.vertical()␣=␣" ≪ r2.vertical( );
    rat_point ipnt;
    cout ≪ "\nr1.intersection(r2,ipnt)␣=␣" ≪ r1.intersection(r2, ipnt);
    cout ≪ "\nwith␣ipnt␣=␣" ≪ ipnt;
    /* some input and access test */
    cout ≪ "\nenter␣ray␣r4:␣";
    cin ≫ r4;
    cout ≪ "ray␣through:␣" ≪ r4.point1( ) ≪ r4.point2( );
    cout ≪ "\norientation(r4,ipnt)␣=␣" ≪ orientation(r4, ipnt);
    cout ≪ "\n";
  }
```

This code is used in chunk 14.

16. Next we do some tests in 3-space. Here we test the d-dimensional components.

```
⟨ 3d tests 16 ⟩ ≡
  {
  /* some construction test */
  cout ≪ "\nd-SPACE␣MODULE:␣(d=3)";
  rat_point p0 = rat_point::d3(1, 1, 1, 1); // a point
  rat_vector e1 = 3
  *rat_vector::unit(0, 3); // the first unit vector
  rat_point p1(p0 + e1), p(3), q(3);
  rat_ray r1(p0, p1), r2(p0, r1.direction( )), r3(3);
  cout ≪ "\ntwo␣rays␣r1,r2:␣" ≪ r1 ≪ r2;
  /* some equality and containment tests */
  cout ≪ "\n(r1␣==␣r2)␣=␣" ≪ (r1 ≡ r2);
  cout ≪ "\ntwo␣points␣p,q␣=␣" ≪ (p = p0 + 5 ∗ e1) ≪ (q = p0 − 9 ∗ e1);
  cout ≪ "\nr1.contains(p)␣=␣" ≪ r1.contains(p);
  cout ≪ "\nr1.contains(q)␣=␣" ≪ r1.contains(q);
```

```
rat_segment s(p0, p);
cout << "\nnew rat_segment s = " << s;
cout << "\nr1.contains(s) = " << r1.contains(s);
/* input */
cout << "\nenter ray r3 (we test against r1): ";
cin >> r3;
cout << "ray through: " << r3.point1() << r3.point2();
cout << "\nparallel(r1,r3) = " << parallel(r1, r3);
/* intersection tests */
rat_hyperplane h(p0, r1.direction(), rat_point::origin(3));
rat_point ipnt1(3), ipnt2(3);
cout << "\na hyperplane h = " << h;
cout << "\nr3.intersection(h,ipnt1) = " << r3.intersection(h, ipnt1);
cout << "\nwith ipnt1 = " << ipnt1;
cout << "\nr3.intersection(r1,ipnt1,ipnt2) = ";
cout << r3.intersection(r1, ipnt1, ipnt2);
cout << "\nwith ipnt1/2 = " << ipnt1 << ipnt2;
rat_line l(3);
cout << "\nenter line l (we test against r1): ";
cin >> l;
cout << "line = " << l;
cout << "\nr1.intersection(l,ipnt1,ipnt2) = ";
cout << r1.intersection(l, ipnt1, ipnt2);
cout << "\nwith ipnt1/2 = " << ipnt1 << ipnt2;
cout << "\n\n"; }
```

This code is used in chunk 14.

Index

List of Refinements

⟨ 2d tests 15 ⟩ Used in chunk 14.
⟨ 3d tests 16 ⟩ Used in chunk 14.
⟨ check ray $*\mathbf{this}$ and ray t on line position 11 ⟩ Used in chunk 10.
⟨ conversion operations 5 ⟩ Used in chunk 3.
⟨ initialization 4 ⟩ Used in chunk 3.
⟨ input and output 6 ⟩ Used in chunk 3.
⟨ intersection operations 8, 9, 10, 12 ⟩ Used in chunk 3.
⟨ `rat_ray-test.c` 14 ⟩
⟨ `rat_ray.c` 3 ⟩
⟨ `rat_ray.h` 2 ⟩
⟨ some 2-dim operations 13 ⟩ Used in chunk 3.
⟨ the containment test 7 ⟩ Used in chunk 3.

Lines with Rational Coordinates in d-Space (class rat_line)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class rat_line

1. Definition

An instance of data type *rat_line* is an oriented line in d-dimensional Euclidian space. *rat_line* is an item type.

2. Creation

rat_line l(*int* $d = 2$);

introduces a variable l of type *rat_line* and initializes it to some line in d-dimensional space

rat_line l(*rat_point* p, *rat_point* q);

introduces a line through p and q and oriented from p to q.
Precondition: p and q are distinct and have the same dimension.

rat_line l(*rat_point* p, *rat_direction* *dir*);

introduces a line through p with direction *dir*.
Precondition: p and *dir* have the same dimension.

rat_line l(*rat_segment* s);

introduces a variable l of type *rat_line* and initializes it to the line through s.*source*( ) and s.*target*( ).
Precondition: s is not trivial.

3. Operations

3.1 Initialization, Access and Conversions

rat_line *rat_line*::d2(*integer x1*, *integer y1*, *integer D1*, *integer x2*, *integer y2*, *integer D2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, D1), (x2, y2, D2)]$ in two-dimensional space.

rat_line *rat_line*::d2(*integer x1*, *integer y1*, *integer x2*, *integer y2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, 1), (x2, y2, 1)]$ in two-dimensional space.

rat_line *rat_line*::d3(*integer x1*, *integer y1*, *integer z1*, *integer x2*, *integer y2*, *integer z2*)

introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, z1, 1), (x2, y2, z2, 1)]$ in three-dimensional space.

rat_line *rat_line*::d3(*integer x1*, *integer y1*, *integer z1*, *integer D1*, *integer x2*, *integer y2*, *integer z2*, *integer D2*)
introduces a variable l of type *rat_line*. l is initialized to the line $[(x1, y1, z1, D1), (x2, y2, z2, D2)]$ in three-dimensional space.

int l.dim( ) returns the dimension of the underlying space.

rat_point l.point1( ) returns a point on l.

rat_point l.point2( ) returns a point on l distinct from l.*point1*( ). The line is directed from *point1* to *point2*.

void l.two_points(*rat_point*& *p1*, *rat_point*& *p2*)
after the call *p1* and *p2* are two different points on l. The line is directed from *p1* to *p2*.

rat_direction l.direction( ) returns the direction of l.

rat_line l.transform(*aff_transformation t*)
returns $t(l)$.

3.2 Tests

bool identical(*rat_line l1*, *rat_line l2*)
test for identity.

bool *l1* $\equiv$ *l2* equality as unoriented lines.

bool *l1* != *l2* inequality as unoriented lines.

bool strong_eq(*rat_line l1*, *rat_line l2*)
equality as oriented lines.

bool l.contains(*rat_point p*)
returns true if p lies on l and false otherwise.

bool l.contains(*rat_segment s*)
returns true if s is part of l and false otherwise.

int parallel(*rat_line l1*, *rat_line l2*)
returns true if *l1* and *l2* are parallel and false otherwise.

3.3 Intersection Calculations

bool l.intersection(*rat_hyperplane h*, *rat_point*& *p*)
returns true if h and l intersect in a single point and false otherwise. In the first case the point of intersection is assigned to p.

rat_point *l*.intersection(*rat_hyperplane h*)

returns the intersection of hyperplane h with line l
Precondition: h and l intersect in a single point.

int *l*.intersection(*rat_line t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ by the following means. The return value is one of the constants $\{NO\_I, PNT\_I, SEG\_I, RAY\_I, LIN\_I\}$. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
NO_I	empty
PNT_I	*rat_point*(*i1*)
SEG_I	*rat_segment*(*i1*, *i2*)
RAY_I	*rat_ray*(*i1*, *i2*)
LIN_I	*rat_line*(*i1*, *i2*)

int *l*.intersection(*rat_ray t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ as above.

int *l*.intersection(*rat_segment t*, *rat_point*& *i1*, *rat_point*& *i2*)

returns the intersection set $l \cap t$ as above.

3.4 Input and Output

ostream& *ostream*& $O \ll l$ writes the coefficients of l to output stream O.

istream& *istream*& $I \gg$ *rat_line*& l

reads the coefficients of l from input stream I. This operator uses the current dimension of l.

Additional Operations for segments in two-dimensional space

bool *l*.vertical( ) returns true if l is vertical and false otherwise.

bool *l*.horizontal( ) returns true if l is horizontal and false otherwise.

bool *l*.intersection(*rat_line l1*, *rat_point*& *p*)

if l and l_1 are not parallel the point of intersection is assigned to p and the result is true, otherwise the result is false.

rat_line *l*.perpendicular(*rat_point p*)

computes the perpendicular line of l through p.

int orientation(*rat_line l*, *rat_point p*)

computes orientation(a, b, p), where a and b are point1 and point2 of l respectively.

int cmp_slopes(*rat_line l1*, *rat_line l2*)

returns *compare*(*slope*(*l1*), *slope*(*l2*)).

4. Implementation

Lines are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a line l take time $O(l.dim(\,))$. $dim(\,)$, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(l.dim(\,))$. The space requirement is $O(l.dim(\,))$.

2. The Header File of class rat_line

The type rat_line is an item class with representation class geo_pair_rep. It shares this representation class with rat_segment and rat_ray. We derive rat_line from handle_base and derive geo_pair_rep from handle_rep. This gives us reference counting for free. We give all implementations which are trivial directly in the header file and postpone all others to the next section. Aside from this the header file is in one-to-one correspondence to the manual page.

```
⟨ rat_line.h 2 ⟩ ≡
#ifndef LEDA_RAT_LINE_H
#define LEDA_RAT_LINE_H
#include "geo_pair_rep.h"
#include "rat_point.h"
#include "rat_hyperplane.h"
#include "rat_segment.h"
#include "rat_ray.h"
  class rat_line : public handle_base {
  geo_pair_rep *ptr() const { return (geo_pair_rep *) PTR; }
  rat_line(const handle_base &b) : handle_base(b) { }
  friend class rat_ray;
  friend class rat_segment;
public:
  rat_line(int d = 2)
  { PTR = new geo_pair_rep (rat_point(d), rat_point(d)); }
  rat_line(const rat_point &p, const rat_point &q)
  {
    if (p ≡ q)
      error_handler(1, "rat_line::constructor: the two points must be different.");
    PTR = new geo_pair_rep (p, q);
  }
  rat_line(const rat_point &p, const rat_direction &dir)
  {
    rat_point q = p + dir.to_rat_vector();
    PTR = new geo_pair_rep (p, q);
  }
  rat_line(const rat_segment &s);
  rat_line(const rat_line &p) : handle_base(p) { }
  ~rat_line() { }
  rat_line &operator=(const rat_line &p)
  { handle_base::operator=(p); return *this; }
  static rat_line d2(integer_t x1, integer_t y1, integer_t D1, integer_t x2, integer_t
      y2, integer_t D2);
  static rat_line d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2);
  static rat_line d3(integer_t x1, integer_t y1, integer_t z1, integer_t x2, integer_t
      y2, integer_t z2);
  static rat_line d3(integer_t x1, integer_t y1, integer_t z1, integer_t D1, integer_t
      x2, integer_t y2, integer_t z2, integer_t D2);
  int dim() const { return (ptr()->source.dim()); }
```

```
rat_point point1( ) const { return ptr( )→source; }
rat_point point2( ) const { return ptr( )→target; }
void two_points(rat_point &p1, rat_point &p2) const
{ p1 = ptr( )→source; p2 = ptr( )→target; }
rat_direction direction( ) const;
rat_line transform(const aff_transformation &t) const
{ return rat_line(point1( ).transform(t), point2( ).transform(t)); }
friend bool identical(const rat_line &l1, const rat_line &l2)
{ return l1.ptr( ) ≡ l2.ptr( ); }
friend bool operator≡(const rat_line &l1, const rat_line &l2)
{ return l1.contains(l2.point1( )) ∧ l1.contains(l2.point2( )); }
friend bool operator≠(const rat_line &l1, const rat_line &l2)
{ return (¬(l1 ≡ l2)); }
friend bool strong_eq(const rat_line &l1, const rat_line &l2)
{ return l1.contains(l2.point1( )) ∧
      (l1.direction( ) ≡ l2.direction( )); }
bool contains(const rat_point &p) const;
bool contains(const rat_segment &s) const;
friend int parallel(const rat_line &l1, const rat_line &l2);
int cmp(const rat_line &, const rat_line &)
{
  error_handler(1, "cmp: for rat_line not implemented.");
  return 0;
}
bool intersection(const rat_hyperplane &h, rat_point &p);
rat_point intersection(const rat_hyperplane &h);
int intersection(const rat_line &t, rat_point &i1, rat_point &i2) const;
int intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const;
int intersection(const rat_segment &t, rat_point &i1, rat_point &i2) const;
friend ostream &operator≪(ostream &O, const rat_line &l);
friend istream &operator≫(istream &I, rat_line &l);
bool vertical( ) const { return ptr( )→dx ≡ 0; }
bool horizontal( ) const { return ptr( )→dy ≡ 0; }
bool intersection(const rat_line &l1, rat_point &p) const;
rat_line perpendicular(const rat_point &p) const;
friend int orientation(const rat_line &l, const rat_point &p);
int cmp_slope(const rat_line &l2) const
{ return sign(ptr( )→dy * l2.ptr( )→dx − l2.ptr( )→dy * ptr( )→dx); }
friend int cmp_slopes(const rat_line &l1, const rat_line &l2);
} ;
inline void Print(const rat_line &l, ostream &out)
{ out ≪ l; }
inline void Read(rat_line &l, istream &in)
{ in ≫ l; }
inline int orientation(const rat_line &l, const rat_point &p)
{ return orientation(l.point1( ), l.point2( ), p); }
```

```
  inline int parallel(const rat_line &l1, const rat_line &l2)
  { return (l1.direction( ) ≡ l2.direction( )) ∨
        (l1.direction( ) ≡ −(l2.direction( ))); }
  inline int cmp_slopes(const rat_line &l1, const rat_line &l2)
  { return l1.cmp_slope(l2); }
#endif
```

3. The Implementation of class rat_line

The tasks to implement are line-hyperplane intersection and point containment check.

```
⟨ rat_line.c 3 ⟩ ≡
#include "rat_line.h"
  ⟨ initialization 4 ⟩
  ⟨ conversion operations 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ intersection operations 7 ⟩
  ⟨ the containment test 10 ⟩
  ⟨ some 2-dim operations 11 ⟩
```

4. Initialization

We provide some easy initialization operations for the two and three dimensional case.

```
⟨ initialization 4 ⟩ ≡
  rat_line rat_line::d2(integer_t x1, integer_t y1, integer_t D1, integer_t x2, integer_t
        y2, integer_t D2)
  { return rat_line(rat_point::d2(x1, y1, D1), rat_point::d2(x2, y2, D2)); }
  rat_line rat_line::d2(integer_t x1, integer_t y1, integer_t x2, integer_t y2)
  { return rat_line(rat_point::d2(x1, y1), rat_point::d2(x2, y2)); }
  rat_line rat_line::d3(integer_t x1, integer_t y1, integer_t z1, integer_t x2, integer_t
        y2, integer_t z2)
  { return rat_line(rat_point::d3(x1, y1, z1), rat_point::d3(x2, y2, z2)); }
  rat_line rat_line::d3(integer_t x1, integer_t y1, integer_t z1, integer_t D1, integer_t
        x2, integer_t y2, integer_t z2, integer_t D2)
  { return rat_line(rat_point::d3(x1, y1, z1, D1), rat_point::d3(x2, y2, z2, D2)); }
```

This code is used in chunk 3.

5. Conversion Operations

To determine the direction of the line we just call the corresponding operation of the representation class.

```
⟨ conversion operations 5 ⟩ ≡
  rat_direction rat_line::direction() const
  { return ptr()→to_rat_direction(); }
  rat_line::rat_line(const rat_segment &s) : handle_base(s)
  { if (s.is_trivial())
      error_handler(1, "rat_line::constructor: segment is trivial."); }
```

This code is used in chunk 3.

6. Input and Output

We just use the input and output routines of our base class geo_pair_rep. Note that in case of input we have to take care not to overwrite a shared representant.

```
⟨ input and output 6 ⟩ ≡
  ostream &operator≪(ostream &O, const rat_line &l)
  {
    O ≪ l.ptr( );
    return O;
  }
  istream &operator≫(istream &I, rat_line &l)
  {
    int d = l.dim( );
    if (l.refs( ) > 1) l = rat_line(d);
    I ≫ l.ptr( );
    if (l.point1( ) ≡ l.point2( ))
      error_handler(1, "operator<<: defining points of line must be different.");
    return I;
  }
```

This code is used in chunk 3.

7. Intersection Calculation

Let l be defined by points p and q. The point of intersection is of the form $x = \lambda q + (1 - \lambda)p$ and satisfies $h(x) = 0$. Since h is a linear function this implies $\lambda = h(p)/(h(p) - h(q))$. Thus $x = (h(p)q - h(q)p)/(h(p) - h(q))$. The i-th component of x is therefore equal to $(q_i h(p)p_d - p_i h(q)q_d)/(p_d q_d(h(p) - h(q))$.

```
⟨ intersection operations 7 ⟩ ≡
  bool rat_line::intersection(const rat_hyperplane &h, rat_point &p)
  {
    int d = h.dim( );
    rat_point p1 = point1( );
    rat_point p2 = point2( );
    if (d ≠ dim( )) error_handler(1,
        "intersection: dimensions of hyperplane and line don't agree");
    integer_t hp1 = h.value_at(p1); // still need to divide by p1[d]
    integer_t hp2 = h.value_at(p2); // as above
    integer_t D = hp1 * p2.hcoord(d) - hp2 * p1.hcoord(d);
    if (D ≡ 0) return false;
    integer_vector c(d);
    for (int i = 0; i < d; i++) c[i] = p2.hcoord(i) * hp1 - p1.hcoord(i) * hp2;
    p = rat_point(c, D);
    return true;
  }
  rat_point rat_line::intersection(const rat_hyperplane &h)
  {
    rat_point p(dim( ));
    if (¬intersection(h, p))
      error_handler(1, "intersection: h and *this must intersect.");
    return p;
  }
```

See also chunks 8 and 9.

This code is used in chunk 3.

8. In this chunk we implement the intersection operation of two lines. If the two lines are parallel, they can be identical or different. In the first case the intersection set is a line, in the second case there's certainly no intersection. If the lines are not parallel, we use our d dimensional intersection routine of our **geo_pair_rep** class the intersection point if they intersect.

```
⟨ intersection operations 7 ⟩ +≡
  int rat_line::intersection(const rat_line &t, rat_point &i1, rat_point &i2) const
  {
    if (dim() ≠ t.dim())
      error_handler(1, "intersection: the dimensions of the objects must agree.");
    if (parallel(*this, t))
      if (t.contains(point1())) {
        i1 = point1();
        i2 = point2();
        return LIN_I;
      }
      else return NO_I;
    rational_t lambda1, lambda2;
    if (ptr()->d_intersection(*(t.ptr()), i1, lambda1, lambda2)) return PNT_I;
    return NO_I;
  }
```

9. The rest is done by a call to the ray and segment class members.

```
⟨ intersection operations 7 ⟩ +≡
  int rat_line::intersection(const rat_ray &t, rat_point &i1, rat_point &i2) const
  { return t.intersection(*this, i1, i2); }
  int rat_line::intersection(const rat_segment &t, rat_point &i1, rat_point &i2) const
  { return t.intersection(*this, i1, i2); }
```

10. Containment

A line l though points s and t contains a point p if there is a λ such that

$$\frac{p_i}{p_d} - \frac{s_i}{s_d} = \lambda \cdot (\frac{t_i}{t_d} - \frac{s_i}{s_d})$$

for all i. In other words, the numbers $(p_i s_d - s_i p_d)t_d / p_d(t_i s_d - s_i t_d)$ must all agree.

```
⟨ the containment test 10 ⟩ ≡
  bool rat_line::contains(const rat_point &p) const
  {
    int d = dim();
    rat_point s = point1();
    rat_point t = point2();
    integer_t lnum = (p.hcoord(0) * s.hcoord(d) - s.hcoord(0) * p.hcoord(d)) * t.hcoord(d);
    integer_t lden = (t.hcoord(0) * s.hcoord(d) - s.hcoord(0) * t.hcoord(d)) * p.hcoord(d);
    integer_t lnum_i, lden_i;
    for (int i = 1; i < d; i++) {
      lnum_i = (p.hcoord(i) * s.hcoord(d) - s.hcoord(i) * p.hcoord(d)) * t.hcoord(d);
      lden_i = (t.hcoord(i) * s.hcoord(d) - s.hcoord(i) * t.hcoord(d)) * p.hcoord(d);
      if (lnum * lden_i ≠ lnum_i * lden) return false;
```

```
    }
    return true;
  }
  bool rat_line::contains(const rat_segment &s) const
  { return contains(s.start()) ∧ contains(s.end()); }
```

This code is used in chunk 3.

11. Special Operations in 2-Space

In this chunk we implement necessary operations for lines in the plane.

```
⟨ some 2-dim operations 11 ⟩ ≡
  bool rat_line::intersection(const rat_line &l1, rat_point &p) const
  { /* The intersection routine is hidden in the class geo_pair_rep */
    return ptr()→d2_intersection(*(l1.ptr()), p);
  }
  rat_line rat_line::perpendicular(const rat_point &p) const
  {
    rat_point s = point1();
    rat_point t = point2();
    /* the homogenous params of the supporting line of l: */
    integer_t a(s.Y() * t.W() - t.Y() * s.W());
    integer_t b(t.X() * s.W() - s.X() * t.W());
    integer_t c(s.X() * t.Y() - t.X() * s.Y());
    /* the homogenous params of the perpendicular through p: */
    integer_t ap(b * p.W());
    integer_t bp(-a * p.W());
    integer_t cp(a * p.Y() - b * p.X());
    /* the intersection point of two lines: */
    rat_point on(b * cp - bp * c, ap * c - a * cp, a * bp - ap * b);
    /* an offset point on the line: */
    rat_point through = on + (t - s);
    /* this is perpendicular of *this through p oriented counterclockwise rotated by 90 degree
    about the intersection point on of the two lines: */
    return rat_line(on, through.rotate90(on));
  }
```

This code is used in chunk 3.

12. A Test of class rat_line

We want to test construction, equality, intersection with hyperplanes and containment of a point. We take a hyperplane in d-Space take a line which intersects it and one which doesn't. The we make our test. Afterwards we do the same for a line and two points on the line and outside.

```
⟨ rat_line-test.c 12 ⟩ ≡
#include "rat_line.h"
  main( )
  {
    ⟨ 2d tests 13 ⟩
    ⟨ 3d tests 14 ⟩
  }
```

13. In this chunk we test the special 2d procedures.

```
⟨ 2d tests 13 ⟩ ≡
  { /* some construction test */
    cout ≪ "\n2-SPACE␣MODULE:";
    rat_point p1(1, 1), p2(−2, 1), p3(1, −1);
    rat_direction dir(1, 1, 1);
    rat_line l1(p1, p2), l2(p2, p3), l3(p1, dir), l4(2), l5(2);
    cout ≪ "\nthree␣lines␣l1,l2,l3:\n" ≪ l1 ≪ l2 ≪ l3;
    cout ≪ "\nl1.horizontal()␣=␣" ≪ l1.horizontal( );
    cout ≪ "\nl2.vertical()␣=␣" ≪ l2.vertical( );
    rat_point ipnt;
    cout ≪ "\nl1.intersection(l2,ipnt)␣=␣" ≪ l1.intersection(l2, ipnt);
    cout ≪ "\nwith␣ipnt␣=␣" ≪ ipnt;
    /* some input and access test */
    cout ≪ "\nenter␣line␣l4:␣";
    cin ≫ l4;
    cout ≪ "line␣through:␣" ≪ l4.point1( ) ≪ l4.point2( );
    cout ≪ "\n(l5␣=␣l4.perpendicular(ipnt))␣=␣";
    cout ≪ (l5 = l4.perpendicular(ipnt));
    cout ≪ "\ninner␣product␣of␣direction␣vectors(l4,l5)␣=␣";
    cout ≪ (l4.direction( ).to_rat_vector( ) * l5.direction( ).to_rat_vector( ));
    cout ≪ "\norientation(l4,ipnt)␣=␣" ≪ orientation(l4, ipnt);
    cout ≪ "\norientation(l5,ipnt)␣=␣" ≪ orientation(l5, ipnt);
    cout ≪ "\ncmp_slopes(l1,l4)␣=␣" ≪ cmp_slopes(l1, l4);
    cout ≪ "\n";
  }
```

This code is used in chunk 12.

14. Next we do some tests in 3-space. Here we test the d-dimensional components.

```
⟨ 3d tests 14 ⟩ ≡
  {
  /* some construction test */
  cout ≪ "\nd-SPACE␣MODULE:␣(d=3)";
```

```
rat_point p0 = rat_point::d3(1,1,1);  // a point
rat_vector e1 = 5
*rat_vector::unit(0,3);  // the first unit vector
rat_point p1(p0 + e1);
rat_line l1(p0,p1), l2(p0,-l1.direction()), l3(3);
cout << "\ntwo lines l1,l2: " << l1 << l2;
/* some equality and containment tests */
cout << "\n(l1 == l2) = " << (l1 == l2);
cout << "\nstrong_eq(l1,l2) = " << strong_eq(l1,l2);
cout << "\nl1.contains((7,2,2,1)) = " << l1.contains(p0 + 5 * e1);
rat_segment s(p0 - 3 * e1, p0 + 7 * e1);
cout << "\nnew rat_segment s = " << s;
cout << "\nl1.contains(s) = " << l1.contains(s);
/* input */
cout << "\nenter line l3 (we test against l1): ";
cin >> l3;
cout << "line through: " << l3.point1() << l3.point2();
cout << "\nparallel(l1,l3) = " << parallel(l1,l3);
/* intersection tests */
rat_hyperplane h(p0, l1.direction(), rat_point::origin(3));
rat_point ipnt1(3), ipnt2(3);
cout << "\na hyperplane h = " << h;
cout << "\nl3.intersection(h,ipnt1) = " << l3.intersection(h,ipnt1);
cout << "\nwith ipnt1 = " << ipnt1;
cout << "\nl3.intersection(l1,ipnt1,ipnt2) = ";
cout << l3.intersection(l1,ipnt1,ipnt2);
cout << "\nwith ipnt1/2 = " << ipnt1 << ipnt2;
cout << "\n\n"; }
```

This code is used in chunk 12.

Index

Affine Transformations in d-Space (class aff_transformation)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class aff_transformation

1. Definition

An instance of the data type *aff_transformation* is an affine transformation of d-dimensional space. It is specified by a square integer matrix M of dimension $d+1$. All entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero. A point p with homogeneous coordinates $(p[0], \ldots, p[d])$ is transformed into the point $M * p$.

aff_transformation is an item type.

2. Creation

aff_transformation $t(int\ d\ =\ 2)$;
introduces the identity transformation in d-dimensional space.

aff_transformation $t(integer\_matrix\ M)$;
the transformation of d-space specified by matrix M.
Precondition: M is a square matrix of dimension $d+1$.

aff_transformation $t(rat\_vector\ v)$;
translation by vector v.

aff_transformation $t(integer\_vector\ v)$;
the transformation of d-space specified by a diagonal matrix with vector v on the diagonal (a scaling of the space).
Precondition: v is a vector of dimension $d+1$.

3. Operations

aff_transformation *aff_transformation*::d2_scale(*integer num*, *integer den*)
returns a scaling by a scale factor *num*/*den*.

aff_transformation *aff_transformation*::d2_transl(*rat_vector vec*)
returns a translation by a vector *vec*.

aff_transformation *aff_transformation*::d2_rot(*integer sin_num*, *integer cos_num*, *integer den*)
returns a rotation with sine and cosine values *sin_num*/*den* and *cos_num*/*den*.
Precondition: $sin\_num^2 + cos\_num^2 = den^2$.

aff_transformation *aff_transformation*::d2_rot_approx(*rat_direction dir*, *integer num*, *integer den*)
returns a rotation of 2-space. Approximates the rotation given by direction *dir*, such that the difference between the sindes and soines of the rotation given by *dir* and the approximation rotation are at most *num*/*den* each.

aff_transformation	*aff_transformation*::d2_trafo(*integer m11*, *integer m12*, *integer m13*, *integer m21*, *integer m22*, *integer m23*, *integer m33*)	returns a general affine transformation in the 3×3 matrix form. The sub matrix $((m_{11}, m_{21})^t, (m_{21}, m_{22})^t)$ contains the scaling and rotation information, the vector $(m_{13}, m_{23})^t$ contains the translational part of the transformation.
aff_transformation	*aff_transformation*::d2_trafo(*integer m11*, *integer m12*, *integer m21*, *integer m22*, *integer m33*)	returns a general linear transformation in the 2×2 matrix form. The sub matrix $((m_{11}, m_{21})^t, (m_{21}, m_{22})^t)$ contains the scaling and rotation information. There's no translational part.
int	*t*.dim( )	the dimension of the underlying space
integer_matrix	*t*.matrix( )	returns the transformation matrix
aff_transformation	*t*.inverse( )	returns inverse transfomation
aff_transformation	*t1* $*$ *t2*	composition of transformations
ostream&	*ostream&* $O \ll a$	writes the transformation matrix of a to output stream O.
istream&	*istream&* $I \gg$ *aff_transformation&* a	reads the coordinates of the transformation matrix from input stream I. This operator uses the current dimension of a.

4. Implementation

Affine Transformations are implemented by matrices of integers as an item type. All operations like creation, initialization, input and output on a transformation t take time $O(t.dim(\,)^2)$. $dim(\,)$ takes constant time. The operations for inversion and composition have the cubic costs of the used matrix operations. The space requirement is $O(t.dim(\,)^2)$.

2. The Header File of class aff_transformation

```
⟨ aff_transform.h 2 ⟩ ≡
#ifndef LEDA_AFF_TRANSFORM_H
#define LEDA_AFF_TRANSFORM_H
#include <math.h>
#include "integer_matrix.h"
#include "rat_vector.h"
#include "rat_direction.h"
  class aff_transformation;
  class rat_vector;
  class rat_direction;
  class aff_transformation_rep : public handle_rep {
    friend class aff_transformation;
    friend ostream &operator≪(ostream &O, const aff_transformation &a);
    friend istream &operator≫(istream &I, aff_transformation &a);
    friend aff_transformation operator ∗ (const aff_transformation &t1, const
        aff_transformation &t2);
    int d;
    integer_matrix M;
  public:
    aff_transformation_rep(const integer_matrix &M_init)
    {
      M = M_init;
      d = M_init.dim1( ) − 1;
    }
    ∼aff_transformation_rep( ) { }
  };
  class aff_transformation : public handle_base
  {
    aff_transformation_rep ∗ptr( ) const
    { return (aff_transformation_rep ∗) PTR; }
  public:
    aff_transformation(int d = 2);
    aff_transformation(const integer_matrix &M);
    aff_transformation(const rat_vector &v);
    aff_transformation(const integer_vector &v);
    static aff_transformation d2_scale(integer_t num, integer_t den);
    static aff_transformation d2_transl(rat_vector vec);
    static aff_transformation d2_rot(integer_t sin_num, integer_t cos_num, integer_t den);
    static aff_transformation d2_rot_approx(rat_direction dir, integer_t num, integer_t den);
    static aff_transformation d2_trafo(integer_t m11, integer_t m12, integer_t m13,
        integer_t m21, integer_t m22, integer_t m23, integer_t m33);
    static aff_transformation d2_trafo(integer_t m11, integer_t m12, integer_t m21,
        integer_t m22, integer_t m33);
    int dim( ) const { return ptr( )→d; }
    integer_matrix matrix( ) const { return ptr( )→M; }
```

```
    aff_transformation inverse( ) const;
    friend aff_transformation operator * (const aff_transformation &t1, const
        aff_transformation &t2);
    friend ostream &operator≪(ostream &O, const aff_transformation &a);
    friend istream &operator≫(istream &I, aff_transformation &a);
  } ;
#endif
```

3. The Implementation of class aff_transformation

```
⟨ aff_transform.c 3 ⟩ ≡
#include "aff_transform.h"
  inline void swap(integer_t & x, integer_t & y)
  {
    integer_t tmp = x;
    x = y;
    y = tmp;
  }
  inline integer_t sqr(integer_t x)
  { return x * x; }
  ⟨ construction 4 ⟩
  ⟨ initialization 5 ⟩
  ⟨ input and output 10 ⟩
  ⟨ basic operations 11 ⟩
```

4. Construction

The constructors provide a simple interface to create affine transformations. We have constructors

- to create the identity transformation in d-space.
- to create an affine transformation from a matrix.
- to create an affine transformation based on integer entries on the diagonal picked from an **integer_vector**.
- to create an translational transformation according to a rational vector.

```
⟨ construction 4 ⟩ ≡
  aff_transformation::aff_transformation(int d)
  { PTR = new aff_transformation_rep (identity(d + 1)); }
  aff_transformation::aff_transformation(const integer_matrix &M)
  { PTR = new aff_transformation_rep (M); }
  aff_transformation::aff_transformation(const integer_vector &v)
  {
    integer_matrix M(v.dim( ), v.dim( ));
    for (int i = 0; i < v.dim( ); i++) M(i, i) = v[i];
    PTR = new aff_transformation_rep (M);
  }
  aff_transformation::aff_transformation(const rat_vector &v)
  {
    int d = v.dim( );
    integer_matrix M(d + 1, d + 1);
    for (int i = 0; i < d; i++) {
      M(i, i) = v.hcoord(d);
      M(i, d) = v.hcoord(i);
    }
```

```
    M(d,d) = v.hcoord(d);
    PTR = new aff_transformation_rep (M);
  }
```

This code is used in chunk 3.

5. Initialization

In this chunk we implement the special initialization operations for dimension 2. First we provide a simple scaling transformation. We put the numerator of the rational scaling factor into the Matrix elements $M_{0,0}$ and $M_{1,1}$ and the denominator into the lower right corner element $M_{2,2}$:

$$\begin{pmatrix} num & & \\ & num & \\ & & den \end{pmatrix}$$

```
⟨ initialization 5 ⟩ ≡
  aff_transformation aff_transformation::d2_scale(integer_t num, integer_t den)
  {
    integer_matrix M(3,3);
    M(0,0) = M(1,1) = num;
    M(2,2) = den;
    return aff_transformation(M);
  }
```

See also chunks 6, 7, 8, and 9.

This code is used in chunk 3.

6. To obtain a translational transformation we call the standard constructor doing the same job. The homogenous representation of the rational vector is put into the last column of the matrix and the rest of the diagonal is filled with the homogenizing element:

$$\begin{pmatrix} vec.W() & & vec.X() \\ & vec.W() & vec.Y() \\ & & vec.W() \end{pmatrix}$$

```
⟨ initialization 5 ⟩ +≡
  aff_transformation aff_transformation::d2_transl(rat_vector vec)
  {
    if (vec.dim() ≠ 2) error_handler(1,"d2_transl: wrong init vector dimension.");
    return aff_transformation(vec);
  }
```

7. To implement a rational rotation we put the standard 2d rotation matrix into the left upper corner and the denominator of the sine and cosine values into the right lower corner:

$$\begin{pmatrix} cos\_num & -sin\_num & \\ sin\_num & cos\_num & \\ & & den \end{pmatrix}$$

```
⟨ initialization 5 ⟩ +≡
  aff_transformation aff_transformation::d2_rot(integer_t sin_num, integer_t cos_num,
          integer_t den)
  {
    if (sin_num * sin_num + cos_num * cos_num ≠ den * den)
      error_handler(1, "d2_rot: rotation parameters disobey precondition.");
    integer_matrix M(3, 3);
    M(0, 0) = cos_num; M(0, 1) = −sin_num;
    M(1, 0) = sin_num; M(1, 1) = cos_num;
    M(2, 2) = den;
    return aff_transformation(M);
  }
```

8. Here we implement a special rotation calculation procedure starting from a direction *dir* and a rational error bound *num*/*den*. We want to find a triple (*sin*, *cos*, *denom*) which obeys the equality $sin^2+cos^2 = denom^2$ and at the same time approximates the rotation given by direction *dir*, such that the differeces between the sindes and cosines of *dir* and the approximation are at most *num*/*den*.

The code is based on the rational rotation method presented by Canny and Ressler at the 8th SCG 1992. The approximation is based on Farey sequences. To check the quality of the current approximation we have to compare a rational and a (possibly) non-rational number. The implementation used division and modulus operation % (the division is always exact, that is, it is known that there is no remainder).

```
⟨ initialization 5 ⟩ +≡
  aff_transformation aff_transformation::d2_rot_approx(rat_direction
          dir, integer_t num, integer_t den)
  {
    if (dir.dim( ) ≠ 2)
      error_handler(1, "d2_rot_approx: dir has to be 2 dimensional.");
    if (num ≤ 0 ∨ den ≤ num)
    error_handler(1, "d2_rot_approx: num and den have to be positive.");
    // now num/den is a rational greater zero
    integer_t sin;
    integer_t cos;
    integer_t denom;
    integer_t dx = abs(dir.X( ));
    integer_t dy = abs(dir.Y( ));
    integer_t sq_hypotenuse = dx * dx + dy * dy;
    integer_t common_part;
    integer_t diff_part;
    integer_t rhs;
    bool lower_ok;
    bool upper_ok;
    if (dy > dx) {
      swap(dx, dy);
    }
    /* approximate sin = dy/sqrt(sq_hypotenuse)
    if (dy/sqrt(sq_hypotenuse) < num/den) */
    if (dy * dy * den * den < sq_hypotenuse * num * num) {
```

```
    cos = denom = 1;
    sin = 0;
  }
  else {
    integer_t p, q, p0, q0, p1, q1;
    p0 = 0;
    q0 = p1 = q1 = 1;
    for ( ; ; ) {
      p = p0 + p1;
      q = q0 + q1;
      sin = integer_t(2) * p * q;
      denom = sqr(p) + sqr(q);  // sanity check for approximation
      // sin/denom < dy/sqrt(hypotenuse) + num/den
      // ∧ sin/denom > dy/sqrt(hypotenuse) − num/den
      // ≡ sin/denom − num/den < dy/sqrt(sq_hypotenuse)
      // ∧ sin/denom + num/den > dy/sqrt(sq_hypotenuse)
      // ≡ (sqr(sin)sqr(den) + sqr(num)sqr(denom))sq_hypotenuse − 2..
      // < sqr(dy)sqr(den)sqr(denom)
      // ∧ (sqr(sin)sqr(den) + sqr(num)sqr(denom))sq_hypotenuse + 2..
      // > sqr(dy)sqr(den)sqr(denom)
      common_part = (sqr(sin) * sqr(den) + sqr(num) * sqr(denom)) * sq_hypotenuse;
      diff_part = integer_t(2) * num * sin * den * denom * sq_hypotenuse;
      rhs = sqr(dy) * sqr(den) * sqr(denom);
      upper_ok = (common_part − diff_part < rhs);
      lower_ok = (common_part + diff_part > rhs);
      if (lower_ok ∧ upper_ok) {
        if (sqr(p) % 2 + sqr(q) % 2 > 1) {
          sin = p * q;
          cos = (sqr(q) − sqr(p))/2;  // exact division
          denom = (sqr(p) + sqr(q))/2;  // exact division
        }
        else {
          cos = sqr(q) − sqr(p);
        }
        break;
      }
      else {
        /* if (dy/sqrt(sq_hypotenuse) < sin/denom) */
        if (sqr(dy) * sqr(denom) < sqr(sin) * sq_hypotenuse) {
          p1 = p;
          q1 = q;
        }
        else {
          p0 = p;
          q0 = q;
        }
      }
    }  // for(;;)
  }
  dx = dir.X( );
```

```
    dy = dir.Y();
    if (dy > dx) swap(sin, cos);
    if (dx < 0) sin = -sin;
    if (dy < 0) cos = -cos;
    return d2_rot(sin, cos, denom);
  }
```

9. The last two constructors are just initilizing the transformation matrix with the corresponding values:

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ 0 & 0 & m_{3,3} \end{pmatrix}$$

```
⟨ initialization 5 ⟩ +≡
  aff_transformation aff_transformation::d2_trafo(integer_tm11, integer_tm12,
        integer_tm13, integer_tm21, integer_tm22, integer_tm23, integer_tm33)
  {
    integer_matrix M(3, 3);
    M(0, 0) = m11; M(0, 1) = m12; M(0, 2) = m13;
    M(1, 0) = m21; M(1, 1) = m22; M(1, 2) = m23;
    M(2, 2) = m33;
    return aff_transformation(M);
  }
  aff_transformation aff_transformation::d2_trafo(integer_tm11, integer_tm12,
        integer_tm21, integer_tm22, integer_tm33)
  {
    integer_matrix M(3, 3);
    M(0, 0) = m11; M(0, 1) = m12;
    M(1, 0) = m21; M(1, 1) = m22;
    M(2, 2) = m33;
    return aff_transformation(M);
  }
```

10. We still need the input and output operators.

```
⟨ input and output 10 ⟩ ≡
  ostream &operator≪(ostream &O, const aff_transformation &a)
  {
    O ≪ a.ptr()→M;
    return O;
  }
  istream &operator≫(istream &I, aff_transformation &a)
  {
    I ≫ a.ptr()→M;
    return I;
  }
```

This code is used in chunk 3.

11. Here we implement the combination of two transformations by matrix multiplication and the inversion of a transformation.

```
⟨ basic operations 11 ⟩ ≡
  aff_transformation aff_transformation::inverse() const
  {
    integer_matrix Inv;
    integer_t D;
    integer_vector dummy;
    if (¬::inverse(ptr()→M, Inv, D, dummy))
      error_handler(1, "aff_transformation::inverse:␣*this␣not␣invertible.");
    return aff_transformation(Inv);
  }
  aff_transformation operator ∗ (const aff_transformation &t1, const
          aff_transformation &t2)
  { return aff_transformation(t1.ptr()→M ∗ t2.ptr()→M); }
```

This code is used in chunk 3.

12. A Test of class aff_transformation

```
⟨ aff_transform-test.c 12 ⟩ ≡
#include "aff_transform.h"
#include "rat_hyperplane.h"
#include "rat_segment.h"
#include "rat_ray.h"
#include "rat_line.h"
  main( )
  {
    integer_vector iv(3);
    iv[0] = 2; iv[1] = 3; iv[2] = 1;
    rat_vector rv = rat_vector::d2(5, 0);
    rat_direction rd = rat_direction::d2(2, 3);
    aff_transformation at1(2),   // id
    at2(iv);   // diag
    aff_transformation at3 = aff_transformation::d2_scale(3, 1),   // scale
    at4 = aff_transformation::d2_transl(rv),   // trans
    at5 = aff_transformation::d2_rot(1, 0, 1),   // rot pi/2
    at6 = aff_transformation::d2_rot_approx(rd, 1, 1000),   // rot
    at7 = aff_transformation::d2_trafo(−1, 0, 0, −1, 1),   // rot pi
    at8 = aff_transformation::d2_trafo(0, −1, 1, 1, 0, 1, 1);   // rot -pi/2 + trans
    cout ≪ "\n2-SPACE␣MODULE:";
    cout ≪ "\nsome␣affine␣transformations␣at1␣-␣at8␣:\n";
    cout ≪ "\nat1␣id␣trafo";
    cout ≪ at1;
    cout ≪ "\nat2␣diag␣trafo";
    cout ≪ at2;
    cout ≪ "\nat3␣3␣scaling␣trafo";
    cout ≪ at3;
    cout ≪ "\nat4␣(5,0)␣translation␣trafo";
    cout ≪ at4;
    cout ≪ "\nat5␣pi/2␣rotation␣trafo";
    cout ≪ at5;
    cout ≪ "\nat6␣(2,3)␣approx␣rotation␣trafo";
    cout ≪ at6;
    cout ≪ "\nat7␣pi␣rotation␣trafo";
    cout ≪ at7;
    cout ≪ "\nat8␣-pi/2␣rotation␣+␣(1,1)␣translation␣trafo";
    cout ≪ at8;
    cout ≪ "\nat5.inverse()␣=␣" ≪ at5.inverse( );
    cout ≪ "\nat8*at8.inverse()␣=␣" ≪ at8 ∗ at8.inverse( );
    rat_point p = rat_point::d2(1, 1);
    cout ≪ "\n\npoint␣p␣=␣" ≪ p;
    cout ≪ "\np.transform(at1)␣=␣" ≪ p.transform(at1);
    cout ≪ "\np.transform(at2)␣=␣" ≪ p.transform(at2);
    cout ≪ "\np.transform(at3)␣=␣" ≪ p.transform(at3);
    cout ≪ "\np.transform(at4)␣=␣" ≪ p.transform(at4);
    cout ≪ "\np.transform(at5)␣=␣" ≪ p.transform(at5);
    cout ≪ "\np.transform(at6)␣=␣" ≪ p.transform(at6);
```

```
  cout << "\np.transform(at7) = " << p.transform(at7);
  cout << "\np.transform(at8) = " << p.transform(at8);
  rat_vector v = rat_vector::d2(1,1);
  cout << "\n\nvector v = " << v;
  cout << "\nv.transform(at1) = " << v.transform(at1);
  cout << "\nv.transform(at2) = " << v.transform(at2);
  cout << "\nv.transform(at3) = " << v.transform(at3);
  cout << "\nv.transform(at4) = " << v.transform(at4);
  cout << "\nv.transform(at5) = " << v.transform(at5);
  cout << "\nv.transform(at6) = " << v.transform(at6);
  cout << "\nv.transform(at7) = " << v.transform(at7);
  cout << "\nv.transform(at8) = " << v.transform(at8);
  rat_direction dir = rat_direction::d2(5,1);
  cout << "\n\ndirection dir = " << dir;
  cout << "\ndir.transform(at8) = " << dir.transform(at8);
  rat_point q = rat_point::d2(-5,1);
  rat_hyperplane hyp = rat_hyperplane::d2(p,q,rat_point::origin(),-1);
  rat_hyperplane hyp_trans = hyp.transform(at7);
  cout << "\n\nhyperplane hyp = " << hyp;
  cout << "\nwith through points p,q: " << p << q;
  cout << "\nhyp_trans = hyp.transform(at8) = " << hyp_trans;
  cout << "\nhyp_trans.contains(p.transform(at8)) = ";
  cout << hyp_trans.contains(p.transform(at8));
  cout << "\nhyp_trans.contains(q.transform(at8)) = ";
  cout << hyp_trans.contains(q.transform(at8));
  rat_segment s(p,q);
  rat_ray r(p,q);
  rat_line l(p,q);
  cout << "\n\nsegment s(p,q) = " << s;
  cout << "\nray r(p,q) = " << r;
  cout << "\nline l(p,q) = " << l;
  cout << "\nall three transformed by at8 = \n";
  cout << s.transform(at8) << "\n";
  cout << r.transform(at8) << "\n";
  cout << l.transform(at8) << "\n\n";
}
```

Index

List of Refinements

List of Refinements

⟨ 2d tests 13 ⟩ Used in chunk 12.
⟨ 3d tests 14 ⟩ Used in chunk 12.
⟨ conversion operations 5 ⟩ Used in chunk 3.
⟨ initialization 4 ⟩ Used in chunk 3.
⟨ input and output 6 ⟩ Used in chunk 3.
⟨ intersection operations 7, 8, 9 ⟩ Used in chunk 3.
⟨ `rat_line-test.c` 12 ⟩
⟨ `rat_line.c` 3 ⟩
⟨ `rat_line.h` 2 ⟩
⟨ some 2-dim operations 11 ⟩ Used in chunk 3.
⟨ the containment test 10 ⟩ Used in chunk 3.

Representation of Geometric Objects (class geo_rep)

Geokernel

May 21, 1996

Contents

1. The Manual Page of class geo_rep

1. Definition

The class *geo_rep* is used to represent *rat_points*, *rat_vectors*, *rat_hyperplanes*, and *rat_directions*. The latter four classes are item types that use *geo_rep* as their representation class. We derive *geo_rep* from handle_rep and each one of the item classes from handle_base. The class *geo_rep* is for internal LEDA-use only and is not visible at the user level, in particular, its manual page does not appear in the LEDA-manual.

An instance of *geo_rep* contains an int *dim* and a C++-vector v of $dim + 1$ integers. The types that use class *geo_rep* usually put additional constraints on the instances, e.g., rat_points and rat_hyperplanes always require $v[dim]$ to be positive. The class *geo_rep* does neither enforce nor check these additional constraints. This is the sole responsibility of the using type.

The class *geo_rep* serves several purposes. Firstly, it is a representation class in the sense of independent item types of LEDA, secondly, it encapsulates the memory management for many geometric objects, and thirdly, it provides some functions common to all types referring to *geo_reps*.

2. Creation

geo_rep g; creates an instance g of type *geo_rep* of dimension 0. (oder besser -1) All components of g are initialized to zero.

geo_rep $g(int\ d)$; creates an instance g of type *geo_rep* of dimension d. All components of g are initialized to zero.

geo_rep $g(integer\_vector\ c)$; creates an instance g of type *geo_rep* of dimension $c.dim(\,) - 1$. g is initialized componentwise by c.

geo_rep $g(integer\_vector\ c, integer\ D)$;
creates an instance g of type *geo_rep* of dimension $c.dim(\,)$. $v[dim]$ is initialized to D and the other components are copied from c.

geo_rep $g(integer\ a, integer\ b, integer\ D)$;
creates an instance g of type *geo_rep* of dimension 2. g is initialized to (a, b, D).

integer_vector g.ivec( ) returns the represented integer tuple.

int g.cmp_rat_coords($geo\_rep * a$, $geo\_rep * b$)
the linear order of cartesian coordinates.

int g.cmp_hom_coords($geo\_rep * a$, $geo\_rep * b$)
the linear order of homogenous coordinates.

void g.init3($integer\ x0$, $integer\ x1$, $integer\ x2$)
initializes the slots 0 to 2 of g.

void g.init4(*integer x0*, *integer x1*, *integer x2*, *integer x3*)
initializes the slots 0 to 3 of g.

void g.copy(*geo_rep* $*$ *g1*) copies the entries of $*g$ into g.

void g.negate(*int d*) inverts the sign of the first d entries.

ostream& *ostream& out* $\ll$ *geo_rep* $*$ *p*
writes the coordinates of p to output stream O.

istream& *istream& in* $\gg$ *geo_rep* $*$ *p*
reads the coordinates of p from input stream I. This operator uses the current dimension of p.

void c_add(*geo_rep* $*$ *res*, *geo_rep* $*$ *a*, *geo_rep* $*$ *b*)
calculates the cartesian sum $*res = *a + *b$ where the three geo_reps are homogenous representations of rational vectors.

void c_sub(*geo_rep* $*$ *res*, *geo_rep* $*$ *a*, *geo_rep* $*$ *b*)
calculates the cartesian difference $*res = *a - *b$ where the three geo_reps are homogenous representations of rational vectors.

2. The Header File of class geo_rep

The class *geo_rep* is used to represent rat_points, rat_hyperplanes, rat_directions, and rat_vectors. The latter four classes are item types that use geo_rep as their representation class. We derive geo_rep from handle_rep and each one of the item classes from handle_base. The class geo_rep is for internal LEDA-use only and is not visible at the user level, in particular, its manual page does not appear in the LEDA-manual.

An instance of class geo_rep contains an int *dim* and a C++-vector v of $dim+1$ integers. The types that use class geo_rep usual ly put additional constraints on the instances, e.g., rat_points and rat_hyperplanes always require $v[dim]$ to be positive. The class geo_rep does neither enforce nor check these additional constraints. This is the sole responsibility of the using type.

The class geo_rep serves several purposes. Firstly, it is a representation class in the sense of independent item types of LEDA, secondly, it encapsulates the memory management for many geometric objects, and thirdly, it provides some functions common to all types referring to geo_reps.

The class geo_rep also has a static member *LEDA_SMALL* that is needed by the memory management.

⟨ data members of class geo_rep 2 ⟩ ≡

```
int dim;
integer_t *v;
static int LEDA_SMALL;
```

This code is used in chunk 3.

3. ⟨ geo_rep.h 3 ⟩ ≡

```
#ifndef LEDA_GEO_REP_H
#define LEDA_GEO_REP_H
  class rat_vector;
  class rat_direction;
  class rat_hyperplane;
  class rat_point;
#include <iostream.h>
#include <math.h>
#include <ctype.h>
#include <LEDA/integer.h>
#include <LEDA/rational.h>
#include "integer_vector.h"
  typedef integer integer_t;
  typedef rational rational_t;
  class geo_rep : public handle_rep
  {
    friend class rat_point;
    friend class rat_hyperplane;
    friend class rat_direction;
    friend class rat_vector;
    ⟨ data members of class geo_rep 2 ⟩
    ⟨ functions to allocate and deallocate small arrays 5 ⟩
    ⟨ friends of class geo_rep 10 ⟩
  public:
```

```
    geo_rep();
    geo_rep(int d);
    geo_rep(const integer_vector &c);
    geo_rep(const integer_vector &c, integer_t D);
    geo_rep(integer_t a, integer_t b, integer_t D);
    ~geo_rep();
    integer_vector ivec() const;
    int cmp_rat_coords(geo_rep *a, geo_rep *b) const;
    int cmp_hom_coords(geo_rep *a, geo_rep *b) const;
  private: void init3(integer_t x0, integer_t x1, integer_t x2);
    void init4(integer_t x0, integer_t x1, integer_t x2, integer_t x3);
    void copy(geo_rep *g1);
    void negate(int d);
    friend ostream &operator<<(ostream &out, geo_rep *p);
    friend istream &operator>>(istream &in, geo_rep *p);
    friend void c_add(geo_rep *res, geo_rep *a, geo_rep *b);
    friend void c_sub(geo_rep *res, geo_rep *a, geo_rep *b);
    LEDA_MEMORY(geo_rep)
  };
#endif
```

4. The Implementation of class geo_rep

```
⟨ geo_rep.c 4 ⟩ ≡
#include "geo_rep.h"
  int geo_rep::LEDA_SMALL = MAX_SIZE_OF_SMALL_OBJECT/sizeof(integer_t);
  ⟨ construction and destruction 7 ⟩;
  ⟨ access 9 ⟩
  ⟨ compare function 14 ⟩;
  ⟨ input and output 16 ⟩
  ⟨ basic arithmetic 17 ⟩
```

5. Memory management.

A constructor allocates and initializes memory and a destructor reverses this process. The simplest way to allocate memory for the array v in a geo_rep of dimension d is to write $v =$ **new integer_t** $[d+1]$. The disadvantage of this approach is that it gives the control over memory management to C++. LEDA has its own memory manager for small objects. The LEDA memory manager has built in garbage collection and we use it for small geo_reps.

We first define procedures *allocate_small* and *deallocate_small* that allocate and deallocate small v's respectively. *Allocate_small* first gets an appropriate piece of memory from the LEDA memory manager and then initializes each cell by an inplace new. The new operator used here is defined in `LEDA/param_types.h`. *Deallocate_small* calls the destructor for type **integer_t** for each cell of the tuple and then returns the piece of memory to the LEDA memory manager.

```
⟨ functions to allocate and deallocate small arrays 5 ⟩ ≡
  void allocate_small(integer_t *&v, int d)
  {
    v = (integer_t *) allocate_bytes(d * sizeof(integer_t));
    integer_t *p = v + d − 1;
    while (p ≥ v) { new (p, 0) integer_t; p−−; }
  }
  void deallocate_small(integer_t *v, int d)
  {
    integer_t *p = v + d − 1;
    while (p ≥ v) { p→∼integer_t( ); p−−; }
    deallocate_bytes(v, d * sizeof(integer_t));
  }
```

This code is used in chunk 3.

6. Construction and Destruction.

All constructors follow a common scheme. We set *dim* to the correct value, allocate space for v and then initialize v. We factor out the second part.

```
⟨ allocate space for v 6 ⟩ ≡
  if (dim < LEDA_SMALL) allocate_small(v, dim + 1);
  else v = new integer_t [dim + 1];
```

This code is used in chunk 7.

7. ⟨ construction and destruction 7 ⟩ ≡

```
geo_rep::geo_rep()
{
  dim = 0;
  ⟨ allocate space for v 6 ⟩;
  v[dim] = 0;
}
geo_rep::geo_rep(int d)
{
  dim = d;
  ⟨ allocate space for v 6 ⟩;
}
geo_rep::geo_rep(integer_t a, integer_t b, integer_t D)
{
  dim = 2;
  ⟨ allocate space for v 6 ⟩;
  init3(a, b, D);
}
geo_rep::geo_rep(const integer_vector &c, integer_t D)
{
  dim = c.dim();
  ⟨ allocate space for v 6 ⟩;
  v[dim] = D;
  for (int i = 0; i < dim; i++) v[i] = c[i];
}
geo_rep::geo_rep(const integer_vector &c)
{
  dim = c.dim() − 1;
  ⟨ allocate space for v 6 ⟩;
  for (int i = 0; i ≤ dim; i++) v[i] = c[i];
}
```

See also chunks 8, 11, 12, and 13.

This code is used in chunk 4.

8. Destruction. The destructor calls *deallocate_small* if the geo_rep is small and **delete** [] otherwise.

⟨ construction and destruction 7 ⟩ +≡

```
geo_rep::~geo_rep()
{
  if (dim < LEDA_SMALL) deallocate_small(v, dim + 1);
  else delete []v;
}
```

9. Access. We need an conversion from the represented tuple to an integer_vector.

⟨ access 9 ⟩ ≡

```
integer_vector geo_rep::ivec() const
{
  integer_vector res(dim + 1);
```

```
    for (int i = 0; i ≤ dim; i++) res[i] = v[i];
    return res;
  }
```
This code is used in chunk 4.

10. Some friends of **geo_rep**. Some outside operations have to be allowed to access private parts of this class.

⟨ friends of class geo_rep 10 ⟩ ≡
```
  friend istream &operator≫(istream &, rat_point &);
  friend istream &operator≫(istream &, rat_vector &);
  friend istream &operator≫(istream &, rat_direction &);
```
This code is used in chunk 3.

11. Initialization.

For our basic dimensions 2 and 3 we want to have special initialization operations.

⟨ construction and destruction 7 ⟩ +≡
```
  void geo_rep::init3(integer_t x0, integer_t x1, integer_t x2)
  {
    v[0] = x0;
    v[1] = x1;
    v[2] = x2;
  }
  void geo_rep::init4(integer_t x0, integer_t x1, integer_t x2, integer_t x3)
  {
    v[0] = x0;
    v[1] = x1;
    v[2] = x2;
    v[3] = x3;
  }
```

12. Copying Contents.

Sometimes we want to copy the content of a geo_rep into another object to modify it further.

⟨ construction and destruction 7 ⟩ +≡
```
  void geo_rep::copy(geo_rep *g1)
  {
    for (int i = 0; i ≤ dim; i++) v[i] = g1→v[i];
  }
```

13. Negation.

We need this operator to invert the sign of a prefix of the integer tuple. It is used either as a correction after an input operations when a negative homogenizing coordinate has to be corrected or as part of the unary minus operation.

⟨ construction and destruction 7 ⟩ +≡
```
  void geo_rep::negate(int d)
  {
    for (int i = 0; i < d; i++) v[i] = −v[i];
  }
```

14. Compare Functions.

The default order on the item types based on geo_rep depends on the item type. Some use the lexicographic ordering of the underlying cartesian coordinate vector.

```
⟨ compare function 14 ⟩ ≡
  int geo_rep::cmp_rat_coords(geo_rep *a, geo_rep *b) const
  {
    if (a→dim ≠ b→dim) error_handler(1, "geo_rep::cmp:␣dimensions␣disagree.");
    integer_t aw = a→v[a→dim];
    integer_t bw = b→v[b→dim];
    int signadim = (aw > 0 ? 1 : −1);
    int signbdim = (bw > 0 ? 1 : −1);
    int s = signadim * signbdim;
    for (int i = 0; i < a→dim; i++) {
      integer_t aibw = a→v[i] * bw;
      integer_t biaw = b→v[i] * aw;
      int S = compare(aibw, biaw);
      if (S ≠ 0) return s * S;
    }
    return 0;
  }
```

See also chunk 15.

This code is used in chunk 4.

15. Another way to compare geo_reps is to simply use the lexicographic ordering on the vector v

```
⟨ compare function 14 ⟩ +≡
  int geo_rep::cmp_hom_coords(geo_rep *a, geo_rep *b) const
  {
    if (a→dim ≠ b→dim) error_handler(1, "geo_rep::cmp:␣dimensions␣disagree.");
    for (int i = 0; i ≤ a→dim; i++) {
      int S = compare(a→v[i], b→v[i]);
      if (S ≠ 0) return S;
    }
    return 0;
  }
```

16. Input and Output.

We write the object componentwise on the output stream. The entries are separated by commas and the object is enclosed in brackets.

```
⟨ input and output 16 ⟩ ≡
  ostream &operator≪(ostream &out, geo_rep *p)
  {
    out ≪ "(";
    for (int i = 0; i < p→dim; i++) out ≪ p→v[i] ≪ ",";
    out ≪ p→v[p→dim] ≪ ")";
    return out;
  }
```

```
istream &operator≫(istream &in, geo_rep *p)
{  /* syntax: (x_0, x_1, ..., x_d) */
  int d = p→dim;
  integer_t x, y, w;
  char c;
  do in.get(c); while (in ∧ isspace(c));
  if (¬in) return in;
  if (c ≠ '(') {
    in.putback(c);
    return in;
  }
  long inputnum;
  for (int i = 0; i < d; i++) {
    in ≫ inputnum;
    p→v[i] = inputnum;
    do in.get(c); while (isspace(c));
    if (c ≠ ',') {
      in.putback(c);
      return in;
    }
  }
  in ≫ inputnum;
  p→v[d] = inputnum;
  do in.get(c); while (isspace(c));
  if (c ≠ ')') in.putback(c);
  return in;
}
```

This code is used in chunk 4.

17. Basic Arithmetic. In this chunk we implement all basic arithmetic operations.

⟨ basic arithmetic 17 ⟩ ≡

```
void c_add(geo_rep *res, geo_rep *a, geo_rep *b)
{  /* We expect res to have the same dim as a */
  int d = a→dim;
  if (d ≠ b→dim) error_handler(1, "cartesian+: dimensions disagree.");
  integer_t aw = a→v[d];
  integer_t bw = b→v[d];
  for (int i = 0; i < d; i++) res→v[i] = a→v[i] * bw + b→v[i] * aw;
  res→v[d] = aw * bw;
}
void c_sub(geo_rep *res, geo_rep *a, geo_rep *b)
{  /* We expect res to have the same dim as a */
  int d = a→dim;
  if (d ≠ b→dim) error_handler(1, "cartesian-: dimensions disagree.");
  integer_t aw = a→v[d];
  integer_t bw = b→v[d];
```

```
    for (int i = 0; i < d; i++) res→v[i] = a→v[i] * bw - b→v[i] * aw;
    res→v[d] = aw * bw;
  }
```

This code is used in chunk 4.

18. Inplace Construction.

Many of the item types that refer to geo_reps have operations $+=$, $-=$, and $\gg$. Consider for example an assignment $p += v$ where p belongs to an item type. The simplest way to realize this assignment is to reduce it to $p = p + v$. This will construct a new object $p + v$ and then assign this object to p. If the old geo_rep pointed to by p is only pointed to by p we could construct the new object in the place of the old geo_rep. The following lines of code encapsulate this reasoning.

```
item_type old(p);
if (ptr()→count > 2) { ptr()→count--; PTR = new geo_rep (dim()); }
```

We first make a copy of p and then check whether the object pointed to by PTR is owned by p. If not, then we decrease the count of the geo_rep pointed to by PTR by one and construct a new geo_rep of the appropiate size. After this line we can fill the v of p. Of course, we have to pay attention to the fact that p and *old* may share the same representation.

19. Optimizations.

An instance of class geo_rep never changes its dimension nor its array of coordinates. Moreover, it is only allocated on the heap. Note that it is a class for internal use only and hence we can guarantee that all constructor calls are preceeded by **new**. We could therefore allocate the space for the array in the object itself. This would save one level of indirection. However, every access operation would have to check whether the object is small or not.

Index

Representation of Geometric Line Objects (class geo_pair_rep)

Geokernel

May 21, 1996

Abstract

The class *geo_pair_rep* is used to represent rat_segments, rat_lines, and rat_rays. The latter three classes are item types that use geo_pair_rep as their representation class. We derive geo_pair_rep from handle_rep and each one of the item classes from handle_base. The class geo_pair_rep is for internal LEDA-use only and is not visible at the user level, in particular, its manual page does not appear in the LEDA-manual.

An instance of class geo_pair_rep contains two rat_points.

Contents

1. The Manual Page of class geo_pair_rep

1. Definition

The class *geo_pair_rep* is used to represent *rat_segments*, *rat_lines*, and *rat_rays*. The latter three classes are item types that use *geo_pair_rep* as their representation class. We derive *geo_pair_rep* from handle_rep and each one of the item classes from handle_base. The class *geo_pair_rep* is for internal LEDA-use only and is not visible at the user level, in particular, its manual page does not appear in the LEDA-manual.

An instance of class *geo_pair_rep* contains two *rat_points*.

2. Creation

geo_pair_rep $g(int\ d\ =\ 0)$: creates an instance g of type *geo_pair_rep* of dimension 0.

geo_pair_rep $g(rat\_point\ p,\ rat\_point\ q)$;
creates an instance g of type *geo_pair_rep* and dimension $p.dim(\,)$ which contains the pair (p, q).

rat_direction g.to_rat_direction( ) returns the direction of g.

rat_vector g.to_rat_vector( ) returns the vector from source to target of g.

ostream& $ostream\&\ out\ \ll\ geo\_pair\_rep * l$
writes the coefficients of g to output stream O.

istream& $istream\&\ in\ \gg\ geo\_pair\_rep * l$
reads the coefficients of g from input stream I. This operator uses the current dimension of g.

bool g.d2_intersection(*geo_pair_rep g1*, *rat_point& p*)
returns true if the lines which are represented by g and *g1* intersect in a single point andfalse otherwise. In the first case the point of intersection is assigned to p.
Precondition: : Both *geo_pair_rep* objects are of the same dimension 2 and their point pairs represent legal lines (the points are different)

bool g.d_intersection(*geo_pair_rep g1*, *rat_point& p*, *rational& l1*, *rational& l2*)
returns true if the lines which are represented by g and *g1* intersect in a single point and false otherwise. In the first case the point of intersection is assigned to p.
Precondition: : Both *geo_pair_rep* objects are of the same dimension d and their point pairs represent legal lines (the points are different)

2. The Header File of class geo_pair_rep

The class *geo_pair_rep* is used to represent rat_segments, rat_lines, and rat_rays. The latter three classes are item types that use geo_pair_rep as their representation class. We derive geo_pair_rep from handle_rep and each one of the item classes from handle_base. The class geo_pair_rep is for internal LEDA-use only and is not visible at the user level, in particular, its manual page does not appear in the LEDA-manual.

An instance of class geo_pair_rep contains two rat_points.

```
⟨ geo_pair_rep.h 2 ⟩ ≡
#ifndef LEDA_GEO_PAIR_REP_H
#define LEDA_GEO_PAIR_REP_H

#include <ctype.h>
#include "rat_point.h"
#include "rat_vector.h"
#include "rat_direction.h"

  typedef integer integer_t;
  typedef rational rational_t;

  enum intersection_types {
    NO_I, PNT_I, SEG_I, RAY_I, LIN_I
  };

  class geo_pair_rep : public handle_rep {
    friend class rat_line;
    friend class rat_ray;
    friend class rat_segment;
    rat_point source;
    rat_point target;
    integer_t dx;  // for 2-dim geometry
    integer_t dy;  // for 2-dim geometry
    /* Any line object in d-space is defined by two points called source and target respectively.
    There exists an orientation from source to target. The classes rat_line, rat_segment and
    rat_ray are handle classes. */
  public:
    geo_pair_rep(int d = 0);
    geo_pair_rep(const rat_point &p, const rat_point &q);
    ~geo_pair_rep() {}
    rat_direction to_rat_direction() const;
    rat_vector to_rat_vector() const;
    friend ostream &operator≪(ostream &out, geo_pair_rep *l);
    friend istream &operator≫(istream &in, geo_pair_rep *l);
    bool d2_intersection(const geo_pair_rep &g1, rat_point &p) const;
    bool d_intersection(const geo_pair_rep &g1, rat_point &p, rational &l1, rational
        &l2) const;
    LEDA_MEMORY(geo_pair_rep)
  };
#endif
```

3. The Implementation of class geo_pair_rep

```
⟨ geo_pair_rep.c 3 ⟩ ≡
#include "geo_pair_rep.h"
  ⟨ construction 4 ⟩
  ⟨ conversion 5 ⟩
  ⟨ input and output 6 ⟩
  ⟨ basic line intersection 7 ⟩
```

4. In this chunk the construction is implemented.

```
⟨ construction 4 ⟩ ≡
  geo_pair_rep::geo_pair_rep(int d)
  {
    source = rat_point(d);
    target = rat_point(d);
    dx = 0;
    dy = 0;
  }
  geo_pair_rep::geo_pair_rep(const rat_point &p, const rat_point &q)
  {
    if (p.dim() ≠ q.dim())
      error_handler(1,
          "geo_pair_rep::constructor: source and target must have the \
          same dimension.");
    source = p;
    target = q;
    dx = p.X() * q.W() − q.X() * p.W();
    dy = p.Y() * q.W() − q.Y() * p.W();
  }
```

This code is used in chunk 3.

5. In this chunk we calculate some geo object from the two points in the container.

```
⟨ conversion 5 ⟩ ≡
  rat_direction geo_pair_rep::to_rat_direction() const
  { return (target − source).to_rat_direction(); }
  rat_vector geo_pair_rep::to_rat_vector() const
  { return (target − source); }
```

This code is used in chunk 3.

6. In this chunk we do the basic input and output.

```
⟨ input and output 6 ⟩ ≡
  ostream &operator≪(ostream &out, geo_pair_rep *l)
  {
    out ≪ "[" ≪ l→source ≪ "===" ≪ l→target ≪ "]";
    return out;
  }
```

```
istream &operator≫(istream &in, geo_pair_rep *l)
{  /* syntax: [ p === q ] */
  int d = l→source.dim( );
  rat_point p(d), q(d);
  char c;
  do in.get(c); while (isspace(c));
  if (c ≠ '[') {
    in.putback(c);
    return in;
  }
  do in.get(c); while (isspace(c));
  in.putback(c);
  in ≫ p;
  do in.get(c); while (isspace(c));
  while (c ≡ '=') in.get(c);
  while (isspace(c)) in.get(c);
  in.putback(c);
  in ≫ q;
  do in.get(c); while (isspace(c));
  if (c ≠ ']') in.putback(c);
  l→source = p;
  l→target = q;
  l→dx = p.X( ) * q.W( ) − q.X( ) * p.W( );
  l→dy = p.Y( ) * q.W( ) − q.Y( ) * p.W( );
  return in;
}
```

This code is used in chunk 3.

7. First we implement some two dimensional intersection routine like the one which was in the former LEDA 2d rational geo module.

⟨ basic line intersection 7 ⟩ ≡

```
bool geo_pair_rep::d2_intersection(const geo_pair_rep &g2, rat_point &p) const
{
  integer_t w = dy * g2.dx − dx * g2.dy;
  if (w ≡ 0) return false;  // same slope
  integer_t c1 = target.X( ) * source.Y( ) − source.X( ) * target.Y( );
  integer_t c2 = g2.target.X( ) * g2.source.Y( ) − g2.source.X( ) * g2.target.Y( );
  p = rat_point(c1 * g2.dx − c2 * dx, c1 * g2.dy − c2 * dy, w);
  return true;
}
```

See also chunk 8.

This code is used in chunk 3.

8. Secondly we implement a d-dimensional intersection routine for two non parallel lines. We do the intersection and node calculation by the help of some matrix calculation. We know that a common point x of two lines s_1t_1 and s_2t_2 through two d-points each has to obey the equations:

$$x = s_1 + \lambda_1(t_1 - s_1) \quad (1)$$
$$x = s_2 + \lambda_2(t_2 - s_2) \quad (2)$$

If we put this together and rearrange with variables λ_1 and λ_2 we get the $d \times 2$ cartesian matrix system

$$\begin{pmatrix} t_{1_0} - s_{1_0} & s_{2_0} - t_{2_0} \\ \vdots & \vdots \\ t_{1_{d-1}} - s_{1_{d-1}} & s_{2_{d-1}} - t_{2_{d-1}} \end{pmatrix} * \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} s_{2_0} - s_{1_0} \\ \vdots \\ s_{2_{d-1}} - s_{1_{d-1}} \end{pmatrix}$$

The code below puts this linear system into an integer matrix after multiplication of each row with its common denominator $s_{1_d} s_{2_d} t_{1_d} t_{2_d}$. Afterwards we solve the system. If we get a solution we calculate the intersection point x and deliver additionally λ_1 and λ_2 as rational numbers for outside use.

⟨ basic line intersection 7 ⟩ +≡

```
bool geo_pair_rep::d_intersection(const geo_pair_rep &g2, rat_point &p, rational_t
        &l1, rational_t &l2) const
{
  int d = source.dim( );
  integer_matrix M(d, 2);
  integer_vector b(d);
  integer_vector lambda(2);
  integer_t D;
  rat_point s1 = source, t1 = target, s2 = g2.source, t2 = g2.target;
  integer_t s1w = s1.W( );
  integer_t t1w = t1.W( );
  integer_t s2w = s2.W( );
  integer_t t2w = t2.W( );
  integer_t g1w = s1w * t1w;
  integer_t g2w = s2w * t2w;
  integer_t t12w = t1w * t2w;
  /* init d × 2-matrix M and d-vector b */
  for (int i = 0; i < d; i++) {
    M(i, 0) = g2w * (t1.hcoord(i) * s1w - s1.hcoord(i) * t1w);
    M(i, 1) = g1w * (s2.hcoord(i) * t2w - t2.hcoord(i) * s2w);
    b[i] = t12w * (s2.hcoord(i) * s1w - s1.hcoord(i) * s2w);
  }
  if (linear_solver(M, b, lambda, D)) {
    l1 = rational_t(lambda[0], D);
    l2 = rational_t(lambda[1], D);
    p = s1 + l1 * (t1 - s1);
    return true;
  }
  return false;
}
```

Index

List of Refinements

⟨ basic line intersection 7, 8 ⟩ Used in chunk 3.
⟨ construction 4 ⟩ Used in chunk 3.
⟨ conversion 5 ⟩ Used in chunk 3.
⟨ `geo_pair_rep.c` 3 ⟩
⟨ `geo_pair_rep.h` 2 ⟩
⟨ input and output 6 ⟩ Used in chunk 3.

List of Refinements

⟨ access 9 ⟩ Used in chunk 4.
⟨ allocate space for v 6 ⟩ Used in chunk 7.
⟨ basic arithmetic 17 ⟩ Used in chunk 4.
⟨ compare function 14, 15 ⟩ Used in chunk 4.
⟨ construction and destruction 7, 8, 11, 12, 13 ⟩ Used in chunk 4.
⟨ data members of class geo_rep 2 ⟩ Used in chunk 3.
⟨ friends of class geo_rep 10 ⟩ Used in chunk 3.
⟨ functions to allocate and deallocate small arrays 5 ⟩ Used in chunk 3.
⟨ `geo_rep.c` 4 ⟩
⟨ `geo_rep.h` 3 ⟩
⟨ input and output 16 ⟩ Used in chunk 4.