Bruhat Decomposition

Straight-line programs with memory and matrix Bruhat decomposition

January 2019

Alice Niemeyer
Dominik Bernhardt
Daniel Rademacher
Christian Singendonk

This implementation follows the ideas of "Straight-line programs with memory and matrix Bruhat decomposition" by Alice Niemeyer, Tomasz Popiel and Cheryl Praeger.

Alice Niemeyer Email: alice.niemeyer@mathb.rwth-aachen.de

Dominik Bernhardt Email: bernhardt@mathb.rwth-aachen.de

Daniel Rademacher Email: daniel.rademacher@rwth-aachen.de

Christian Singendonk Email: christian.singendonk@rwth-aachen.de

Address: Lehrstuhl B für Mathematik
Pontdriesch 14/16
52062 Aachen
(Germany)

Copyright

Contents

1 Foreword **4**

2 Bruhat Decomposition for the special linear group (SL) **5**
2.1 Implemented Subfunctions (Part I) 6
2.2 UnipotentDecomposition (Part II - a) 7
2.3 UnipotentDecomposition saving Transvections (Part II - b) 7
2.4 Decomposing the Monomial Matrix (Part III) 7
2.5 Main Function (Part IV) 8
2.6 NC Version 8
2.7 Functions 9
2.8 Local functions for UnipotentDecomposition 17
2.9 Local functions for UnipotentDecompositionWithTi 18

3 Bruhat Decomposition for the special unitary group (SU) **20**
3.1 Functions 20

4 Bruhat Decomposition for the symplectic group (Sp) **23**
4.1 Functions 23

5 Bruhat Decomposition for the special orthogonal group (SO) **25**
5.1 Functions 25

References **26**

Chapter 1

Foreword

Let G be the SL, SU, Sp or SO. Let g be an element in G. We want to write $g = u_1 \cdot w \cdot u_2$ with u_1 and u_2 lower unitriangular matrices and w a monomial matrix.

This is already implemented for:

- Special linear group (SL) [2]
- Special unitary group (SU) [3]
- Symplectic group (Sp) [4]

Chapter 2

Bruhat Decomposition for the special linear group (SL)

This implementation follows the ideas of "Straight-line programs with memory and matrix Bruhat decomposition" by Alice Niemeyer, Tomasz Popiel and Cheryl Praeger.

Let $g \in SL(d, p^f)$. The Bruhat Decomposition computes $g = u_1 \cdot w \cdot u_2$, where

- u_1, u_2 are lower unitriangular matrices
- w is monomial matrix

In this algorithm we want to compute the Bruhat-Decomposition of g and give g (respectively u_1, w and u_2) as word in the "LGO standard generators". This generators can be found in "Constructive Recognition of Classical Groups in odd characteristic" by C. R. Leedham-Green and E. A. O'Brien. [NPP, Chapter 3.1]

- 1) While computing u_1 (resp u_2) with some kind of Gauß-Algorithm, we express the matrices as product of transvections. For $1 \leq j \leq\neq i \leq d : t_{i,j}(\alpha)$ is the matrix T with 1-entries on diagonal, $T_{i,j} = \alpha$, 0 elsewhere. Each $t_{i,j}(\alpha)$ can be computed from $t_{2,1}(\alpha)$ via recursion, where we have to distinguish the odd and even dimensons [NPP, Lemma 4.2]. This again can be expressed as a product of $t_{2,1}(\omega^{ell})$ (where ω is a primitive element and $0 \leq ell \leq\neq f$). The transvections as words in the standard generators are described in [NPP, Lemma 4.2]. This yields a decomposition of u_1 and u_2 in standard generators.
- 2) In a further step we will decompose the monomial Matrix w in a "product of permutations" and a diagonal Matrix. (How to associate this product of permutations with a product of generators is further described in "Implemented Subfunctions (Part I)" [2.1] and "Decomposing the Monomial Matrix (Part III)" [2.4]).
- 3) The last step is the decomposition of the diagonal Matrix in 2) as a word in the standard generators.

We won't do this matrix multiplications directly, but write them in a list to evaluate in a StraightLineProgram. [NPP, Section 2] Although described differently in the paper, we sometimes will allow instructions to multiply more than two elements (eg during conjugating). This doesn't affect the optimality of an SLP much, but higly increases the readability of our implementation.

2.1 Implemented Subfunctions (Part I)

Later we will need some additional functions. Why they are needed and where they are needed is described here.

- `MakeSLP()`: After the `BruhatDecomposition()` we get a list of instructions to calculate the matrices we want using the LGO standard generators. `MakeSLP()` is used to get a SLP out of these instructions.
- `CoefficientsPrimitiveElement()`: It expresses an element w in a field fld as a linear combination of a Primitive Element. This is important for the transvections. [NPP, Lemma 4.2]
- `MyPermutationMat()`: Turns a permutation into a permutation matrix. We need it to calculate the LGO standard generator.
- `LGOStandardGens()`: This function computes the standard generators of SL as given by C. R. Leedham-Green and E. A. O'Brien in "Constructive Recognition of Classical Groups in odd characteristic". [NPP, Chapter 3.1]
- `HighestSlotOfSLP()`: The following function determines the highest slot of a SLP constructed from the list slp will write in. This is important to glue SLPs together.
- `MatToWreathProd()` and `WreathProdToMat()`: In `PermSLP()` [2.7.22] we want to transform the monomial matrix w given by `UnipotentDecomposition()` into a diagonal matrix. (The exact procedure is described in `PermSLP()` [2.7.22])

  Since multiplying the LGO standard-generators s, v and x not only involves permutations but we also have to consider which non-zero entries are $+1$ and which -1, we want to associate this matrices with permutations on $2d$ points. (cf. Wreath-Product)

  $[s,v,x] \to Sym(2d), M \to$ `Mwr` where $i^{Mwr} = j$ and $(i+d)^{Mwr} = j+d$ if $M_{i,j} = 1$ and $i^{Mwr} = j+d$ and $(i+d)^{Mwr} = j$ if $M_{i,j} = -1$ for $1 \leq i \leq d$.

  Due to their relation to wreath-products, we will call denote the image of a matrix `M`$\in [s,v,x]$ by `Mwr`.

  In fact the association from `MatToWreathProd()` [2.7.9] is an isomorphism and we can associate to each permutation we compute during `PermSLP()` [2.7.22] a signed permutation matrix (a monomial matrix with only $+1$ and -1 as non-zero entries).

  $M_{i,j} = 1$ if $i^{Mwr} = j \leq d$ and $M_{i,j} = -1$ if $i^{Mwr} = j+d$
- `AEM()`: Write instructions for Ancient Egyptian Multiplication in slp. At several occasions we will need to compute a high power of some value saved in a memory slot.
- `TestIfMonomial()`: Tests if a given matrix M is monomial matrix. We use it to decide whether we are already finished in `UnipotentDecomposition()`.

For some functions also exist a NC version. See [2.6].

2.2 UnipotentDecomposition (Part II - a)

In this section is the `UnipotentDecomposition()` described. This method is used to compute the Unitriangular decomposition of the matrix g. [2.7.16]

For this we use five local functions in the `UnipotentDecomposition()`. They are `TransvecAtAlpha()`, `ShiftTransvections()`, `FastShiftTransvections()`, `BackShiftTransvections()` and `FastBackShiftTransvections()`. For further information to these functions look at [2.8].

The difference to `UnipotentDecompositionWithTi()` [2.3] is that this version won't store all the transvections $t_{i,i-1}(\omega^l)$. This will increase the runtime but reduce the memory usage by $(d-3)\cdot f$ compared to the `UnipotentDecompositionWithTi()`.

The function can be called for example by

Example

```
gap> d := 3;;
gap> q := 5;;
gap> L := SL(d, q);;
gap> m := PseudoRandom(L);;
gap> stdgens := LGOStandardGens(d, q);;
gap> UnipotentDecomposition( stdgens, g);;
```

2.3 UnipotentDecomposition saving Transvections (Part II - b)

In this section is the `UnipotentDecompositionWithTi()` described. This method is used to compute the Unitriangular decomposition of the matrix g. [2.7.18]

In this version we will store all the transvections $t_{i,i-1}(\omega^l)$. This will increase the memory usage by $(d-3)\cdot f$ but reduce runtime.

In `UnipotentDecompositionWithTi()` we use two local functions. They are `TransvectionAtAlpha()` and `ComputeAllTransvections()`. For further information to these functions look at [2.9].

The function can be called for example by

Example

```
gap> d := 3;;
gap> q := 5;;
gap> L := SL(d, q);;
gap> m := PseudoRandom(L);;
gap> stdgens := LGOStandardGens(d, q);;
gap> UnipotentDecompositionWithTi( stdgens, g);;
```

2.4 Decomposing the Monomial Matrix (Part III)

We use three functions to decompose the monomial matrix w we get from `UnipotentDecomposition()`. They are:

- `PermutationMonomialMatrix()`: Find the permutation (in Sym(d) corresponding to the monomial matrix w) and $diag$ a diagonal matrix, where $diag[i]$ is the non-zero entry of row i. [2.7.20]

- `PermSLP()`: In this function we will transform a monomial matrix $w \in \mathrm{SL}(d,q)$ into a diagonal matrix $diag$. Using only the standard-generators s,v,x. This will lead to a monomial matrix p_{sign} with only ± 1 in non-zero entries and $p_{sign} \cdot diag = w$ (i.e. $diag = (p_{sign})^{-1} \cdot w$).

  Furthermore we will return list `slp` of instructions which will (when evaluated at the LGO standard-generators) yield $diag$. It is sufficient for $diag$ to be diagonal, if the permutation associated with w (i.e. $i^{\pi_w} = j$ if $M_{i,j} \neq 0$) is the inverse of the permutation associated to p_{sign} (again only to $\mathrm{Sym}(d)$).

  In `PermSLP()` we thus transform π_w to $()$ using only $\{\pi_s, \pi_v, \pi_x\}$. In order to know $diag$ without computing all matrix multiplications, (we don't know the signs of p_{sign}), we compute a second permutation simultaneously (here using their identification with permutations in $\mathrm{Sym}(2d)$ and identifying $\{\pi_s, \pi_v, \pi_x\}$ with $\{s,v,x\}$). [2.7.22]

- `DiagonalDecomposition()`: Writes a list of instructions which evaluated on LGO standard-generators yield the diagonal matrix of the input. [2.7.24]

To these three functions is also a NC version implemented. See [2.6].

2.5 Main Function (Part IV)

In `BruhatDecomposition()` [2.7.26] we put everything together. We use the three functions `UnipotentDecomposition()` [2.7.16], `PermSLP()` [2.7.22] and `DiagonalDecomposition()` [2.7.24] to compute matrices with $u_1^{-1} \cdot p_{sign} \cdot diag \cdot u_2^{-1} = g$ and a SLP `pgr` that computes these matrices with the LGO standard generators.

Here is an exampel:

Example

```
gap> mat := [ [ Z(5)^2, Z(5)^0, Z(5)^2 ],
>             [ Z(5)^3, 0*Z(5), 0*Z(5) ],
>             [ 0*Z(5), Z(5)^2, Z(5)^2 ] ];;
gap> L := BruhatDecomposition(LGOStandardGens(3,5), mat);
gap> result := ResultOfStraightLineProgram(L[1], LGOStandardGens(3,5));
```

`BruhatDecompositionWithTi()` [2.7.28] works like `BruhatDecomposition()` [2.7.26] but uses `UnipotentDecompositionWithTi()` [2.7.18] instead of `UnipotentDecomposition()` [2.7.16].

You can use it in the same way like `BruhatDecomposition()`:

Example

```
gap> mat := [ [ Z(5)^2, Z(5)^0, Z(5)^2 ],
>             [ Z(5)^3, 0*Z(5), 0*Z(5) ],
>             [ 0*Z(5), Z(5)^2, Z(5)^2 ] ];;
gap> L := BruhatDecompositionWithTi(LGOStandardGens(3,5), mat);
gap> result := ResultOfStraightLineProgram(L[1], LGOStandardGens(3,5));
```

To both functions is also a NC version implemented. See [2.6].

2.6 NC Version

Here is the NC version of the Bruhat Decomposition described. In all implemented functions are all used functions replaced through their NC version (if one exists). Moreover are all checks from functions of MyBruhatDecomposition removed.

These functions has been modified by this actions and got a NC Version:

- `MakeSLP()`[2.7.1] → `MakeSLPNC()`[2.7.2] (uses the NC version of `StraightLineProgram`)
- `MyPermutationMat()` [2.7.4] → `MyPermutationMatNC()` [2.7.5] (uses the NC version of `ConvertToMatrixRep`)
- `LGOStandardGens()` [2.7.6] → `LGOStandardGensNC()` [2.7.7] (uses the NC version of `MyPermutationMat()`)
- `MatToWreathProd()` [2.7.9] → `MatToWreathProdNC()` [2.7.10] (no checks for user input)
- `TestIfMonomial()` [2.7.13] → `TestIfMonomialNC()` [2.7.14] (no checks for user input)
- `UnipotentDecomposition()` [2.7.16] → `UnipotentDecompositionNC()` [2.7.17] (no checks for user input)
- `UnipotentDecompositionWithTi()` [2.7.18] → `UnipotentDecompositionWithTiNC()` [2.7.19] (no checks for user input)
- `PermutationMonomialMatrix()` [2.7.20] → `PermutationMonomialMatrixNC()` [2.7.21] (no checks for user input)
- `PermSLP()` [2.7.22] → `PermSLPNC()` [2.7.23] (no checks for unser input and uses `PermutationMonomialMatrixNC()`)
- `DiagonalDecomposition()` [2.7.24] → `DiagonalDecompositionNC()` [2.7.25] (no checks for user input)
- `BruhatDecomposition()` [2.7.26] → `BruhatDecompositionNC()` [2.7.27] (uses `UnipotentDecompositionNC()`, `PermSLPNC()` and `DiagonalDecompositionNC`)
- `BruhatDecompositionWithTi()` [2.7.28] → `BruhatDecompositionWithTiNC()` [2.7.29] (uses `UnipotentDecompositionWithTiNC()`, `PermSLPNC()` and `DiagonalDecompositionNC()`)

2.7 Functions

2.7.1 MakeSLP

▷ `MakeSLP(`*`slp, genlen`*`)` (function)

Returns: An SLP using the instructions of slp and genlen inputs
Input:

- `slp`: A list of instructions for a straight-line program
- `genlen`: The number of inputs for our SLP (ie the number of generators)

Uses `StraightLineProgram` to make an SLP out of the instructions from slp and dependent on genlen.

2.7.2 MakeSLPNC

▷ `MakeSLPNC(slp, genlen)` (function)

Returns: An SLP using the instructions of slp and genlen inputs

This function works like `MakeSLP()` [2.7.1] but uses `StraightLineProgramNC` instead of `StraightLineProgram`.

2.7.3 CoefficientsPrimitiveElement

▷ `CoefficientsPrimitiveElement(fld, alpha)` (function)

Returns: A vector c such that for ω primitive element in `fld` is $\sum c[i]\omega^{(i-1)} =$`alpha`

Input:

- `fld`: A field
- `alpha` : An element of `fld`

The function has been written by Thomas Breuer. It expresses an element w in a field fld as a linear combination of a Primitive Element.

2.7.4 MyPermutationMat

▷ `MyPermutationMat(perm, dim, fld)` (function)

Returns: The permutation matrix $res \in fld^{dim \times dim}$ of `perm` (i.e. $res_{i,j} = 1_{fld}$ if $i^{perm} = j$).

Input:

- `perm`: A permutation
- `dim` : A natural number
- `fld` : A field

Given a permutation `perm`, an integer `dim`> 0 and a field `fld`, this function computes the permutation matrix $res \in fld^{dim \times dim}$.

2.7.5 MyPermutationMatNC

▷ `MyPermutationMatNC(perm, dim, fld)` (function)

Returns: The permutation matrix $res \in fld^{dim \times dim}$ of `perm` (i.e. $res_{i,j} = 1_{fld}$ if $i^{perm} = j$).

This function works like `MyPermutationMat()` [2.7.4] but uses `ConvertToMatrixRepNC` instead of `ConvertToMatrixRep`.

2.7.6 LGOStandardGens

▷ `LGOStandardGens(d, q)` (function)

Returns: `stdgens` the LGO standard-generators of $SL(d,q)$.

Input:

- `d`: the dimension of our matrix
- `q`: A prime power $q = p^f$, where F_q ist the field whereover the matrices are defined

This function computes the standard generators of SL as given by C. R. Leedham-Green and E. A. O'Brien in "Constructive Recognition of Classical Groups in odd characteristic". [LGO] [NPP, Chapter 3.1]

2.7.7 LGOStandardGensNC

▷ `LGOStandardGensNC(d, q)` (function)

Returns: `stdgens` the LGO standard-generators of $SL(d,q)$.
Input:

- `d`: the dimension of our matrix
- `q`: A prime power $q = p^f$, where F_q ist the field whereover the matrices are defined

This function works like `LGOStandardGens()` [2.7.6] but uses `MyPermutationMatNC()` instead of `MyPermutationMat()`.

2.7.8 HighestSlotOfSLP

▷ `HighestSlotOfSLP(slp)` (function)

Returns: `highestslot` the number of slots the SLP `slp` will need if evaluated.
Input:

- `slp`: A list of instructions satisfying the properties for an SLP

The function determines the highest slot a SLP constructed from the list slp will write in.

2.7.9 MatToWreathProd

▷ `MatToWreathProd(M)` (function)

Returns: `perm`: the permutation `Mwr` (see Describtion)
Input:

- `M`: A signed permutation matrix.

In `PermSLP()` [2.7.22] we want to transform the monomial matrix w given by `UnipotentDecomposition()` into a diagonal matrix. (The exact procedure is described in `PermSLP()` [2.7.22])

Since multiplying the LGO standard-generators s, v and x not only involves permutations but we also have to consider which non-zero entries are $+1$ and which -1, we want to associate this matrices with permutations on $2d$ points. (cf. Wreath-Product)

$[s,v,x] \to Sym(2d), M \to$ `Mwr` where $i^{Mwr} = j$ and $(i+d)^{Mwr} = j+d$ if $M_{i,j} = 1$ and $i^{Mwr} = j+d$ and $(i+d)^{Mwr} = j$ if $M_{i,j} = -1$ for $1 \leq i \leq d$.

Due to their relation to wreath-products, we will call denote the image of a matrix `M`$\in [s,v,x]$ by `Mwr`.

2.7.10 MatToWreathProdNC

▷ `MatToWreathProdNC(M)` (function)

Returns: `perm`: the permutation `Mwr` (see Describtion)
Input:

- `M`: signed permutation matrix.

This function works like `MatToWreathProd()` [2.7.9] but doesn't check the user input.

2.7.11 WreathProdToMat

▷ `WreathProdToMat(perm, dim, fld)` (function)

Returns: `res`: The Matrix `M` satisfying the properties from the description.
Input:

- `perm`: A permutation in $\mathrm{Sym}(2d)$ sth. $\{\{i, i+d\}\} : 1 \leq i \leq d$ are blocks
- `dim`: The dimension of the matrix we want perm send to
- `fld`: The field whereover the matrix is defined

In fact the association from `MatToWreathProd()` [2.7.9] is an isomorphism and we can associate to each permutation we compute during `PermSLP()` [2.7.22] a signed permutation matrix. $M_{i,j} = 1$ if $i^{Mwr} = j \leq d$ and $M_{i,j} = -1$ if $i^{Mwr} = j + d$

2.7.12 AEM

▷ `AEM(spos, respos, tmppos, k)` (function)

Returns: `instr`: Lines of an SLP that will (when evaluated) take the value b saved in `spos` and write b^k in `respos`
Input:

- `spos`: The memory slot, where a value b is saved in.
- `respos`: The memory slot we want the exponentation to be written in.
- `tmppos`: A memory slot for temporary results.
- `k`: An integer.

At several occasions we will need to compute a high power of some value saved in a memory slot. For this purpose this function is a variaton of AEM (Ancient Egytian Multiplication).

Remark: `tmpos` and `respos` must differ. If `spos` = `respos` or `spos` = `tmpos` it will be overwritten.

2.7.13 TestIfMonomial

▷ `TestIfMonomial(M)` (function)

Returns: true if `M` is Monomial, false else.
Input:

- `M`: A matrix.

Tests if a given matrix `M` is a monomial matrix. There is a function in GAP, however it does not seem to work for $\mathrm{SL}(d,q)$.

2.7.14 TestIfMonomialNC

▷ `TestIfMonomialNC(M)` (function)

Returns: true if `M` is Monomial, false else.

Input:

- `M`: A matrix.

This function works like `TestIfMonomial()` [2.7.13] but doesn't check the user input.

2.7.15 Transvections2

▷ `Transvections2(stdgens, omega, slp, pos)` (function)

Returns: slp: The list of instruction with additional instructions writing $t_{2,1}(\omega^{ell})$ in Slot $pos[ell+1]$ für $0 \leq ell \leq\neq f$.

Input:

- `stdgens`: The LGO standard-generators of $\mathrm{SL}(d,q)$.
- `omega`: A primitive element of $GF(q)$.
- `slp`: A list of instructions.
- `pos`: A list of numbers, denoting where to save the transvections $t_{2,1}(\omega^{ell})$ für $0 \leq ell \leq\neq f$.

Let `stdgens` be the list of standard generators for $\mathrm{SL}(d,p^f)$ and let ω be a primitive element of $G(p^f)$. This function computes T_2 and record what we do in `slp`.

This function coincides with [NPP, Equation (6), p12].

2.7.16 UnipotentDecomposition

▷ `UnipotentDecomposition(arg)` (function)

Returns: `slp`: A list of instructions yielding u_1,u_2 if evaluated as SLP, $[u_1,g,u_2]$ (the matrices of the Bruhat-Decomposition).

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in $\mathrm{SL}(d,q)$.

Computes the Unitriangular decomposition of the matrix g.

2.7.17 UnipotentDecompositionNC

▷ `UnipotentDecompositionNC(arg)` (function)

Returns: `slp`: A list of instructions yielding u_1,u_2 if evaluated as SLP, $[u_1,g,u_2]$ (the matrices of the Bruhat-Decomposition).

This function works like `UnipotentDecomposition()` [2.7.16] but doesn't check the user input.

2.7.18 UnipotentDecompositionWithTi

▷ `UnipotentDecompositionWithTi(arg)` (function)

Returns: `slp`: A list of instructions yielding u_1, u_2 if evaluated as SLP, $[u_1, g, u_2]$ (the matrices of the Bruhat-Decomposition).

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in $\mathrm{SL}(d,q)$.

Compute the Bruhat decomposition of the matrix g, given the standard generators for the group.

In this version we will store all the transvections $T_{i,i-1}(\omega^l)$. This will increase the memory usage by $(d-3)\cdot f$ but reduce runtime.

2.7.19 UnipotentDecompositionWithTiNC

▷ `UnipotentDecompositionWithTiNC(arg)` (function)

Returns: `slp`: A list of instructions yielding u_1, u_2 if evaluated as SLP, $[u_1, g, u_2]$ (the matrices of the Bruhat-Decomposition).

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in $\mathrm{SL}(d,q)$.

This function works like `UnipotentDecompositionWithTi()` [2.7.18] but doesn't check the user input.

2.7.20 PermutationMonomialMatrix

▷ `PermutationMonomialMatrix(M)` (function)

Returns:

- `diag`: The vector of non-zero entries where diag[i] is the non-zero entry of row i.
- `perm`: The permutation associated to `M` (ie $i^{perm} = j$ if $M_{i,j} \neq 0$).

Input:

- `M`: A monomial Matrix

Find the permutation (in $\mathrm{Sym}(d)$ corresponding to the monomial matrix `M`).

2.7.21 PermutationMonomialMatrixNC

▷ `PermutationMonomialMatrixNC(M)` (function)

Returns:

- `diag`: The vector of non-zero entries where diag[i] is the non-zero entry of row i.
- `perm`: The permutation associated to `M` (ie $i^{perm} = j$ if $M_{i,j} \neq 0$).

Input:

- `M`: A monomial Matrix

This function works like `PermutationMonomialMatrix()` [2.7.20] but doesn't check the user input.

2.7.22 PermSLP

▷ `PermSLP(stdgens, mat[, slp])` (function)

Returns:

- `slp`: A list of instructions to evaluate `p_sign` (if `slp` was Input then this instructions are added to `slp`).
- `p_sign`: The signed permutation matrix.
- `mat`: The diagonal matrix `diag`.

Input:

- `stdgens`: The LGO standard-generators.
- `mat`: A monomial matrix (i.e. w).
- `slp`: An already existing list of instructions. [optional]

In this function we will transform a monomial matrix $w \in \mathrm{SL}(d,q)$ into a diagonal matrix $diag$. Using only the standard-generators s,v,x. This will lead to a monomial matrix p_{sign} with only ± 1 in non-zero entries and $p_{sign} \cdot diag = w$ (i.e. $diag = (p_{sign})^{-1} \cdot w$).

Furthermore we will return list `slp` of instructions which will (when evaluated at the LGO standard-generators) yield $diag$. It is sufficient for $diag$ to be diagonal, if the permutation associated with w (i.e. $i^{\pi_w} = j$ if $M_{i,j} \neq 0$) is the inverse of the permutation associated to p_{sign} (again only to $\mathrm{Sym}(d)$).

In `PermSLP()` we thus transform π_w to () using only $\{\pi_s, \pi_v, \pi_x\}$. In order to know $diag$ without computing all matrix multiplications, (we don't know the signs of p_{sign}), we compute a second permutation simultaneously (here using their identification with permutations in $\mathrm{Sym}(2d)$ and identifying $\{\pi_s, \pi_v, \pi_x\}$ with $\{s,v,x\}$).

2.7.23 PermSLPNC

▷ `PermSLPNC(stdgens, mat[, slp])` (function)

Returns:

- `slp`: A list of instructions to evaluate `p_sign` (if `slp` was Input then this instructions are added to `slp`).
- `p_sign`: The signed permutation matrix.
- `mat`: The diagonal matrix `diag`.

Input:

- `stdgens`: The LGO standard-generators.
- `mat`: A monomial matrix (i.e. w).
- `slp`: An already existing list of instructions. [optional]

This function works like `PermSLP()` [2.7.22] but doesn't check the user input.

2.7.24 DiagonalDecomposition

▷ `DiagonalDecomposition(`*`stdgens, diam[, slp]`*`)` (function)

Returns:

- `slp`: A list of instructions to evaluate diag (if slp was Input then this instructions are added to slp).
- `hres`: The the identity matrix.

Input:

- `stdgens`: The LGO standard-generators.
- `diam`: A diagonal matrix (e.g. $diag$).
- `slp`: An already existing list of instructions. [optional]

Writes a list of instructions which evaluated on LGO standard-generators yield the diagonal matrix of the input.

2.7.25 DiagonalDecompositionNC

▷ `DiagonalDecompositionNC(`*`stdgens, diam[, slp]`*`)` (function)

Returns:

- `slp`: A list of instructions to evaluate diag (if slp was Input then this instructions are added to slp).
- `hres`: The the identity matrix.

Writes a list of instructions which evaluated on LGO standard-generators yield the diagonal matrix of the input.

This function works like `DiagonalDecomposition()` [2.7.24] but doesn't check the user input.

2.7.26 BruhatDecomposition

▷ `BruhatDecomposition(`*`stdgens, g`*`)` (function)

Returns: `pgr`: A SLP (to compute u_1, u_2, p_{sign} and $diag$) and the matrices u_1, u_2, p_{sign} and $diag$ itself.

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in $\mathrm{SL}(d,q)$.

Uses `UnipotentDecomposition()`, `PermSLP()` and `DiagonalDecomposition()` to write a matrix $g \in SL(d,q)$ as $g = (u_1)^{-1} \cdot p_{sign} \cdot diag \cdot (u_2)^{-1}$ where u_1, u_2 are lower unitriangular matrices, p_{sign} a monomial matrix with only $+1$ and -1 as non-zero entries and $diag$ a diagonal matrix.

It furthermore yields a SLP that returns the above matrices if evaluated at the LGO standard-generators.

2.7.27 BruhatDecompositionNC

▷ `BruhatDecompositionNC(stdgens, g)` (function)

Returns: `pgr`: A SLP (to compute u_1, u_2, p_{sign} and $diag$) and the matrices u_1, u_2, p_{sign} and $diag$ itself.

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in SL(d,q).

This function works like `BruhatDecomposition()` [2.7.26] but uses `UnipotentDecompositionNC()`, `PermSLPNC()` and `DiagonalDecompositionNC()` instead of `UnipotentDecomposition()`, `PermSLP()` and `DiagonalDecomposition()`.

2.7.28 BruhatDecompositionWithTi

▷ `BruhatDecompositionWithTi(stdgens, g)` (function)

Returns: `pgr`: A SLP (to compute u_1, u_2, p_{sign}, $diag$ and all transvections $t_{i,i-1}(\omega^{ell})$) and the matrices u_1, u_2, p_{sign} and $diag$ itself.

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in SL(d,q).

Works as `BruhatDecomposition()` [2.7.26] but replaces `UnipotentDecomposition()` by `UnipotentDecompositionWithTi()`.

2.7.29 BruhatDecompositionWithTiNC

▷ `BruhatDecompositionWithTiNC(stdgens, g)` (function)

Returns: slp: A list of instructions to compute $u1, u2, p_sign$ and $diag$.

Input:

- `stdgens`: The LGO standard-generators.
- g: A matrix in SL(d,q).

This function works like `BruhatDecompositionWithTi()` [2.7.28] but uses `UnipotentDecompositionWithTiNC()`, `PermSLPNC()` and `DiagonalDecompositionNC()` instead of `UnipotentDecompositionWithTi()`, `PermSLP()` and `DiagonalDecomposition()`.

2.8 Local functions for UnipotentDecomposition

2.8.1 TransvectionAtAlpha

▷ `TransvectionAtAlpha(alpha)` (function)

Returns: A list of instructions to evaluate diam (if slp was Input then this instructions are added to slp), the identity matrix

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Let slp be the list of instructions in UnipotentDecomposition and Tipos denote the slots where transvections $t_{i,j}(\omega^{ell})$ for $0 \leq ell \leq\neq f$ are saved. This function computes $t_{i,j}(\alpha) = \prod t_{i,j}(\omega^{ell})^{a_{ell}}$ [NPP, 4.2] where the exponents a_{ell} are given by `CoefficientsPrimitiveElement()`. [2.7.3] [NPP, p11]

2.8.2 ShiftTransvections

▷ `ShiftTransvections(i)` (function)

Returns: No output

Let Ti be the set of transvections $t_{i,i-1}(\omega^{ell})$ and Ti_1 the set of transvections $t_{i-1,i-2}(\omega^{ell})$.

ShiftTransvections computes $t_{i+1,i}(\omega^{ell})$ for given Ti and Ti_1 [NPP, p12], stores them in the variable Ti and stores the transvections $t_{i,i-1}(\omega^{ell})$ in the variable Ti_1.

This corresponds to [NPP, eq (7+8) p12].

2.8.3 FastShiftTransvecions

▷ `FastShiftTransvecions(i)` (function)

Returns: No output

Given $t_{2,1}$ we compute $t_{i,i-1}$ using fast exponentation. This algorithm will be called in each step of the main loop and is more efficient than calling ShiftTransvections [2.8.2] (r-2) times.

2.8.4 BackShiftTransvections

▷ `BackShiftTransvections(i)` (function)

Returns: No output

This function is very similar to ShiftTransvections [2.8.2], except it works in the reverse order, namely BackShiftTransvections computes $t_{i+1,i}$ given $t_{i+2,i}$ and $t_{i+3,i+2}$.

2.8.5 FastBackShiftTransvections

▷ `FastBackShiftTransvections(i)` (function)

Returns: No output

As for ShiftTransvections [2.8.2] , we need an efficient way to compute BackShiftTransvections [2.8.4] multiple times in a row.

2.9 Local functions for UnipotentDecompositionWithTi

2.9.1 TransvectionAtAlpha2

▷ `TransvectionAtAlpha2(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq\neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

2.9.2 ComputeAllTransvections

▷ `ComputeAllTransvections()` (function)

Returns: No output.

We first compute all the Ti for $i \geq 3$ and add them to the SLP. This are eq (7) and (8) p12 in References [NPP, eq (7+8) p12]. used instead of Schift- and BackshiftTransvections [2.8.2 and 2.8.4].

Chapter 3

Bruhat Decomposition for the special unitary group (SU)

In progress.

3.1 Functions

3.1.1 LGOStandardGensSU

▷ `LGOStandardGensSU(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.2 UnitriangularDecompositionSUEven

▷ `UnitriangularDecompositionSUEven(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.3 UnitriangularDecompositionSUOdd

▷ `UnitriangularDecompositionSUOdd(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.4 UnitriangularDecompositionSU

▷ `UnitriangularDecompositionSU(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.5 CoefficientsPrimitiveElementS

▷ `CoefficientsPrimitiveElementS(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.6 BruhatDecompositionSU

▷ `BruhatDecompositionSU(gens, g)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.7 MonomialSLPSUOdd

▷ `MonomialSLPSUOdd(arg)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.8 MonomialSLPSUEven

▷ `MonomialSLPSUEven(arg)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.9 CheckContinue

▷ `CheckContinue(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.10 CycleFromPermutation

▷ `CycleFromPermutation(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.11 CycleFromListMine

▷ `CycleFromListMine(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.12 DiagSLPSUOdd

▷ `DiagSLPSUOdd(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.13 DiagSLPSU

▷ `DiagSLPSU(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

3.1.14 DiagSLPSUEven

▷ `DiagSLPSUEven(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

Chapter 4

Bruhat Decomposition for the symplectic group (Sp)

In progress.

4.1 Functions

4.1.1 LGOStandardGensSp

▷ `LGOStandardGensSp(i, alpha)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

4.1.2 UnitriangularDecompositionSp

▷ `UnitriangularDecompositionSp(arg)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

4.1.3 MonomialSLPSp

▷ `MonomialSLPSp(arg)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

4.1.4 DiagSLPSp

▷ `DiagSLPSp(arg)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

4.1.5 BruhatDecompositionSp

▷ `BruhatDecompositionSp(`*gens, g*`)` (function)

Returns: true

Let $\alpha \in GF(p^f)$, $\alpha = \sum a_l \omega^l$, ω a primitive element. Suppose further that Tipos is a list of transvections of the form $t_{i,i-1}(\omega^{ell})$,$2 \leq i \leq d$, $0 \leq ell \leq \neq f$. Then this function computes $t_{i,i-1}(\alpha)$ by (Lemma 4.2) and saves the result in tvpos.

Chapter 5

Bruhat Decomposition for the special orthogonal group (SO)

In progress.

5.1 Functions

References

[LGO] C. R. Leedham-Green and E. A. O'Brien. *Constructive Recognition of Classical Groups in odd characteristic.* 11

[NPP] A. C. Niemeyer, T. Popiel, and C. E. Praeger. *Straight-line programs with memory and matrix Bruhat decomposition.* 5, 6, 11, 13, 18, 19