1. Overview. The purpose of this program is to be used in the proof of:

Theorem. Let N be a closed orientable hyperbolic 3-manifold. Then if $f : M \to N$ is a homotopy equivalence where M is an irreducible manifold, then f is homotopic to a homeomorphism.

The details of the proof and how it uses the result of this program are beyond the scope of this document. It is not so clear what exactly the scope should be, but I have chosen to make it as narrow as possible, with occasional references to higher-level concepts.

I have made an effort to make this program comprehensible to mathematicians without an extensive background in programming. It is written in C++, but mostly it is a C program – the only departures from C are in defining expressions like $x + y$ when x and y are composite types (e.g., complex numbers), and for constructing composite types (e.g., $\mathbf{XComplex}(a, b)$) from their component parts, and declaring variables as they are needed (instead of at the beginning of the function). My hope is that anyone who has written a short C program will be able to understand this program.

This draft includes the program, and statements of the lemmas needed for a proof of its correctness.

If $g, h \in \mathbf{PSL_2C}$, we will define the translation length $L(g) \in \mathbf{C}$ of g and the ortholength $L_h(g) \in \mathbf{C}$ of g with respect to h.

A pair $(f, w) \in \mathbf{PSL_2C}$ is minimal if, for all $g \in < f, w >, g \neq I$, (i) $|L(f)| \leq |L(g)|$ and (ii) $|L_f(w)| \leq |L_f(g)|$.

We will define maps $s : \mathbf{C} \to \mathbf{PSL_2C}$ and $c : \mathbf{C^2} \to \mathbf{PSL_2C}$, and a domain $D \subset \mathbf{C^3}$ (chosen so that $(s, c) : \mathbf{C^3} \to \mathbf{PSL_2CxPSL_2C}$ covers the space of groups of interest, modulo conjugation).

We will also define a "recursive partition" $Z : \{0, 1\}^* \to 2^{\mathbf{C^3}}$ of D such that $Z(s) = Z(s0) \cup Z(s1)$ for all $s \in \{0, 1\}^*$ and $Z(\lambda) \supset D$.

If $R \subset \mathbf{C^3}$, and if $(s(x_0), c(x_1, x_2))$ is not minimal for any $x \in R \cap D$, we say that R is MNG-free.

LEMMA If this program finishes, and prints "verified p - $\{m_0\ m_1\ \ldots\ m_k\}$.", then $Z(p) - \bigcup_i Z(m_i)$ is MNG-free.

The input to the program is a sequence of integers, which encode how the proof is to be carried out. This input can affect how long the program will run, and determines whether or not it will finish, but does not affect the correctness of its proof. All of these inputs were produced by programs, some of which are a good deal more complicated than this one, but fortunately they are also beyond the present scope. It suffices to treat the inputs as oracular advice.

2. The program will split the proof into many pieces. For each piece $Z(p_i)$, it will either show that $Z(p_i) \cap D$ is empty or show that $(s(x_0), c(x_1, x_2))$ is not minimal for any $x \in Z(p_i)$. In the second case, this is an exhibition that (i) or (ii) does not hold: a word a in the free group generated by the symbols f and w such that $g \neq I$, $g \neq -I$, and either (i) $|L(g)| < |L(f)|$ or (ii) $|L_f(g)| < |L_f(w)|$ whenever $f = s(x_0), w = c(x_1, x_2)$, g is the corresponding product, and $x \in Z(p_i)$.

If we treat $L(f)$, $L(g)$, $L_f(w)$, and $L_f(g)$ as functions from $Z(p_i)$ to $\mathbf{C}$, the problem can be thought of in terms of bounding the range of $|L(g)/L(f)|$ and $|L_f(g)/L_f(w)|$.

To get these bounds, we work with local complex-affine approximations: given $(f, f_0, f_1, f_2, \epsilon)$, the class of all functions $g : H \to \mathbf{C}$ such that $|g(x) - (f + f_0x_0 + f_1x_1 + f_2x_2)| < \epsilon$ for all $x \in H$, where H is D^3: $H = \{z : |z_i| <= 1, \text{ for } i = 0, 1, 2\}$.

Each of the functions we use is constructed from simple functions using only addition, subtraction, multiplication, division, and square root. At each stage in the construction, the program calculates a local complex-affine approximation which contains the class of possible result functions, given local complex-affine approximation of possible input function(s).

3. The program has a simple structure. In *verify*, the problem is broken into pieces, corresponding to strings p_i such that $Z(p_i)$ can be treated with a single calculation.

roundoff discusses the properties of computer arithmetic.

The remainder of the program is presented in reverse order, starting with the highest-level concepts and ending with the most basic.

inequalityHolds uses the specified inequality to prove that the specified piece is MNG-free. To do this, it uses two abstractions: approximate complex 1-jets (ACJ), and 2x2 matrices of ACJ.

The **SL2ACJ** class reduces operations on matrices of ACJ to operations on ACJ. It also defines the length and ortholength functions and the s and c maps.

The **ACJ** class defines operations on ACJ in terms of operations on intervals of complex numbers.

The **AComplex** and **XComplex** classes define operations on intervals of complex numbers.

4. Covering the parameter space.

verify pieces together the results of *inequalityHolds* to cover larger regions.

Codes from the input mean one of three things, as far as *verify* is concerned. When $code < 0$, *where* is an exceptional case, to be handled by some other entity, so *verify* prints *where* and returns. When $code = 0$, *where* is a composite case, so *verify* calls itself, first on *where0* and then on *where1*. When $code > 0$, *where* is an atomic case, so *verify* uses *inequalityHolds* to show that $Z(where)$ is MNG-free. If this isn't possible, *verify* tries the same code on two parts of *where*.

If $autocode > 0$, then *verify* doesn't read from the input, and uses *autocode* instead.

LEMMA If $verify(where, depth, code)$ finishes, and prints "$m_0 \ldots m_k$", then $Z(where) - \bigcup_i Z(m_i)$ is MNG-free.

⟨ Definition of *verify* 4 ⟩ ≡

```
void verify(char *where, int depth, int autocode)
{
  int code;
  if (depth ≥ MAXDEPTH) {
    where[depth] = '\0';
    fprintf(stderr, "verify: fatal error at %s\n", where);
    exit(1);
  }
  if (autocode ≡ 0) {
    scanf("%d", &code);
  }
  else {
    code = autocode;
  }
  if (code < 0) {
    where[depth] = '\0';
    printf("%s ", where);
  }
  else if (code ≠ 0 ∧ inequalityHolds(inequalityFor(code), where, depth)) {      /* where is MNG-free */
  }
  else {
    where[depth] = '0';
    verify(where, depth + 1, code);
    where[depth] = '1';
    verify(where, depth + 1, code);
  }
}
```

This code is used in section 70.

5. Positive codes refer to line numbers in the file "conditionlist", which lists all of the inequalities used, one per line. Given a code number, *inequalityFor* returns the corresponding inequality from the file. The first time it's called, *inequalityFor* also reads the file into *inequalities*, and initializes an array of pointers to the beginning of each line.

```
⟨ Definition of inequalityFor 5 ⟩ ≡
  const char *inequalityFor(int code)
  {
    const int max_n_inequalities = 13200;
    const int max_inequalities_size = 300000;
    static int n_inequalities = 0;
    static char inequalities[max_inequalities_size];
    static char *inequalityIndex[max_n_inequalities];
    if (n_inequalities ≡ 0) {
      FILE *fp = fopen("conditionlist","r");
      if (fp ≡ Λ) {
        fprintf(stderr,"can't␣open␣conditionlist\n");
        exit(1);
      }
      int n_read = fread(inequalities,1,max_inequalities_size,fp);
      char *ip = inequalities;
      n_inequalities = 1;
      inequalityIndex[n_inequalities++] = inequalities;
      while (n_inequalities < max_n_inequalities ∧ ip < inequalities + n_read − 1) {
        if (*ip ≡ '\n') inequalityIndex[n_inequalities++] = ip + 1;
        ip++;
      }
      fclose(fp);
    }
    if (code < 1 ∨ code ≥ n_inequalities) {
      fprintf(stderr,"code␣%d␣out␣of␣range␣[1,%d]␣in␣inequalityFor\n",code,n_inequalities);
      exit(1);
    }
    return inequalityIndex[code];
  }
```

This code is used in section 70.

6. The *main* program is almost trivial.

```
⟨ Definition of main 6 ⟩ ≡
  main(int argc, char **argv)
  {
    if (argc ≠ 2) {
      fprintf(stderr, "Usage: %s position < data\n", argv[0]);
      exit(1);
    }
    char where[MAXDEPTH];
    int depth;
    for (depth = 0; argv[1][depth] ≠ '\0'; depth++) {
      if (argv[1][depth] ≠ '0' ∧ argv[1][depth] ≠ '1') {
        fprintf(stderr, "bad position %s\n", argv[1]);
        exit(1);
      }
      where[depth] = argv[1][depth];
    }
    where[depth] = '\0';
    printf("verified %s - { ", where);
    initialize_roundoff( );
    verify(where, depth, 0);
    if (¬roundoff_ok( )) {
      printf(". underflow may have occurred\n");
      exit(1);
    }
    printf("}.\n");
    exit(0);
  }
```

This code is used in section 70.

7.

```
⟨ Definition of MAXDEPTH 7 ⟩ ≡
  const int MAXDEPTH = 200;
```

This code is used in section 70.

8. Computer arithmetic. The arithmetic operations defined on a computer are of necessity inexact. In this section, we discuss the relevant properties of computer arithmetic. We also define the constants involved, and the procedures needed to control the floating-point behavior.

The model we use for floating-point arithmetic is the IEEE-754 double-precision standard; it is used on all of the computers the computation is run on.

It would be possible to regard floating-point numbers as a subset of the rationals. Instead, we regard them as separate objects, and use a map $S : \mathbf{double} \to \mathbf{R}$.

This program does not allow underflow. We ask the system to signal an error if underflow occurs; and the error bounds we use are only valid when there is no underflow.

In order to understand the error bounds, we need to introduce some properties of floating-point arithmetic. The IEEE standard implies (assuming there is no underflow) that the error of the operations $+$, $-$, $*$, $/$ and *sqrt* is bounded by `EPS`$/2$ times the absolute value of the result. More specifically, there is a finite set of numbers which are representable on the computer, and the result of these operations is always the closest representable number to the true solution. The standard also implies that if n is not too large and x is representable, then $1 + n$`EPS`, $2^n x$ and $-x$ are representable and that $\mathbf{S}((1 + \texttt{EPS}) * x) > \mathbf{S}(\mathbf{x})$ and $\mathbf{S}((1 - \texttt{EPS}) * x) < \mathbf{S}(x)$ for $x \neq 0$.

Finally, according to the documentation provided for the *hypot* function, there cannot be a representable number between $\mathbf{S}(\mathit{hypot}(x))$ and $\sqrt{\mathbf{S}(\mathbf{x})}$.

⟨ Definition of `EPS` and `HALFEPS` 8 ⟩ ≡
```
#define EPS  DBL_EPSILON
#define HALFEPS  (EPS/2)
```
This code is used in section 58.

9. It is unfortunate that there is no standard way to use the IEEE functions for floating-point control. We have to use a different method for each kind of machine.

⟨ Declaration of floating-point functions 9 ⟩ ≡
```
#ifdef __GNUG__
  inline double fabs(double x)
  {
    return x < 0 ? -x : x;
  }
  extern "C"
  {
#endif      /* C++ */
    void initialize_roundoff( );
    int roundoff_ok( );
#ifndef __sparc__
    extern double infinity( );
#endif      /* sparc */
#ifdef __GNUG__
  }
#endif      /* C++ */
```
This code is used in section 58.

10. ⟨ Definition of floating-point functions 10 ⟩ ≡

```
#ifndef __sparc__
  inline double infinity( )
  {
    return 1.0/0.0;
  }
#endif
#ifdef sgi
#include <sys/fpu.h>
  void initialize_roundoff( )
  {
    union fpc_csr csr;
    csr.fc_word = get_fpc_csr( );
    csr.fc_struct.en_underflow = 1;
    set_fpc_csr(csr.fc_word);
  }
#else
#ifdef __sparc__
#include <floatingpoint.h>
  void initialize_roundoff( )
  {
    ieee_handler("set","underflow",SIGFPE_ABORT);
  }
#else    /* sparc */
  void initialize_roundoff( )
  { }
#endif    /* sparc */
#endif    /* sgi */
#if defined (sgi) ∨ defined (__sparc__)
  int roundoff_ok( )
  {
    return 1;
  }
#else    /* sgi sparc */
#if defined (AIX)
  int roundoff_ok( )
  {
    return fp_underflow( ) ≡ 0;
  }
#else    /* AIX */
  int roundoff_ok( )
  {
    return 0;
  }
#endif    /* AIX */
#endif    /* sgi sparc */
```

This code is used in section 59.

11. Proving inequalities. LEMMA If inequalityHolds(*what*, *where*) returns 1, then $Z(where)$ is MNG-free.

```
⟨ Definition of inequalityHolds 11 ⟩ ≡
  int inequalityHolds(const char *code, const char *where, int depth)
  {
    double pos[6], size[6], scale[6];
    ⟨ Find pos and size of where, depth 12 ⟩
    ACJ along((XComplex(pos[0], pos[3])), XComplex(size[0], size[3]), 0, 0, 0);
    ACJ ortho((XComplex(pos[1], pos[4])), 0, XComplex(size[1], size[4]), 0, 0);
    ACJ whirle((XComplex(pos[2], pos[5])), 0, 0, XComplex(size[2], size[5]), 0);
    switch (code[0]) {
    case 's':    /* short : |along| < e^0.0978... */
      return absUB(along) < 1.10274;
    case 'l':    /* long : |along| > e^1.28978... */
      return absLB(along) > 3.63201;
    case 'n':    /* near : |ortho| < 1 */
      return absUB(ortho) < 1;
    case 'f':    /* far : |ortho| > 3 */
      return absLB(ortho) > 3;
    case 'W':    /* whirle small : |whirle| < 1 */
      return absUB(whirle) < 1;
    case 'w':    /* whirle big : |whirle^2| > |along| */
      double wh = absLB(whirle);
      return (1 - EPS) * wh * wh > absUB(along);
    default:
      {
        SL2ACJ g(evaluateWord(code + 1, along, ortho, whirle));
        ACJ l = length(g);
        switch (code[0]) {
        case 'O':    /* ortho : |ortho(g)| < |ortho(w)| and g not a power of f */
          ACJ o = orthodist(g);
          return absUB(o/ortho) < 1 ∧ absLB(o * ortho) > 1 ∧ notFPower(g);
        case 'L':    /* length : |length(g)| < |length(w)| and g nontrivial */
          return notIdentity(g) ∧ absUB(l/along) < 1 ∧ absLB(l * along) > 1;
        case '2':    /* identity : |length(g)| < |length(w)| and code is simple */
          return wordImpliesCommuting(code + 1) ∧ absUB(l/along) < 1 ∧ absLB(l * along) > 1;
        default: assert(0);
          return (0);
        }
      }
    }
  }
```

This code is used in section 69.

12. We may take the values of scale to be exact, since $pow(2, (5 - i)/6.0)$ takes the same values on all of the machines on which the program was run.

```
⟨ Find pos and size of where, depth 12 ⟩ ≡
  for (int i = 0; i < 6; i++) {
    pos[i] = 0;
    size[i] = 4;
    scale[i] = pow(2, (5 − i)/6.0);
  }
  for (int d = 0; d < depth; d++) {
    size[d % 6] /= 2;
    if (where[d] ≡ '0') {
      pos[d % 6] −= size[d % 6];
    }
    else if (where[d] ≡ '1') {
      pos[d % 6] += size[d % 6];
    }
    else {
      assert(0);
    }
  }
  for (i = 0; i < 6; i++) {
    pos[i] *= scale[i];
    size[i] = (1 + 2 * EPS) * (size[i] * scale[i] + HALFEPS * fabs(pos[i]));
  }
```

This code is used in section 11.

13. ⟨ Definition of *evaluateWord* 13 ⟩ ≡

```
SL2ACJ evaluateWord(const char *word, const ACJ &along, const ACJ &ortho, const ACJ
        &whirle)
{
  ACJ one(1), zero(0);
  SL2ACJ f(shortGenerator(along));
  SL2ACJ w(closeGenerator(ortho, whirle));
  SL2ACJ F(inverse(f));
  SL2ACJ W(inverse(w));
  SL2ACJ g(one, zero, zero, one);
  int i;
  for (i = 1; word[i] ≠ ')'; i++) {
    switch (word[i]) {
    case 'w': g = g * w;
      break;
    case 'W': g = g * W;
      break;
    case 'f': g = g * f;
      break;
    case 'F': g = g * F;
      break;
    default: assert(0);
    }
  }
  return g;
}
```

This code is used in section 69.

14. For any integers k, l, $f^k w^l = 1$ implies that f and w commute. If *wordImpliesCommuting*(*word*) returns 1, then *word* is of the form $f^k w^l$.

⟨ Definition of *wordImpliesCommuting* 14 ⟩ ≡

```
int wordImpliesCommuting(const char *word)
{
  for (word++; word[0] ≠ ')' ∧ word[1] ≠ ')'; word++) {
    if ((word[0] ≡ 'f' ∧ word[1] ≡ 'f') ∨ (word[0] ≡ 'F' ∧ word[1] ≡ 'F') ∨ (word[0] ≡ 'w' ∧ word[1] ≡
          'w') ∨ (word[0] ≡ 'W' ∧ word[1] ≡ 'W') ∨ (word[0] ≡ 'f' ∧ word[1] ≡ 'w') ∨ (word[0] ≡
          'f' ∧ word[1] ≡ 'W') ∨ (word[0] ≡ 'F' ∧ word[1] ≡ 'w') ∨ (word[0] ≡ 'F' ∧ word[1] ≡ 'W')) ;
        /* word is OK – keep checking */
    else return 0;
  }
  return 1;
}
```

This code is used in section 69.

15. Matrices of Approximate Complex 1-Jets. This section implements some operations on matrices of **ACJ**s. It is used to model intervals of isometries of H^3.

An **SL2ACJ** g is a tuple (a, b, c, d) of **ACJ**s, which represents the class $\mathbf{S(g)}$ of functions $h : \mathbf{A} \to \mathbf{PSL_2(C)}$ such that $h_{0,0} \in \mathbf{S(a)}$, $h_{0,1} \in \mathbf{S(b)}$, $h_{1,0} \in \mathbf{S(c)}$, and $h_{1,1} \in \mathbf{S(d)}$.

⟨ Definition of **SL2ACJ** 15 ⟩ ≡
```
struct SL2ACJ {
SL2ACJ( ): a(1), b(0), c(0), d(1)
  { }
SL2ACJ(const ACJ &aa, const ACJ &bb, const ACJ &cc, const ACJ &dd) :
      a(aa), b(bb), c(cc), d(dd)
  { }
  ACJ a, b, c, d;
};
```
This code is used in section 66.

16. PROPOSITION $M * J$ If x and y are **SL2ACJ**, then $\mathbf{S}(x * y) \supset \mathbf{S(x)S(y)}$.

⟨ Definition of $x * y$ for **SL2ACJ** $x,\ y$ 16 ⟩ ≡
```
return SL2ACJ(x.a * y.a + x.b * y.c, x.a * y.b + x.b * y.d, x.c * y.a + x.d * y.c, x.c * y.b + x.d * y.d);
```
This code is used in section 67.

17. PROPOSITION $inverse(M)$ If x is **SL2ACJ**, then $\mathbf{S}(inverse(x)) = \mathbf{S(x)^{-1}}$.

⟨ Definition of $inverse(x)$ for **SL2ACJ** x 17 ⟩ ≡
```
return SL2ACJ(x.d, -x.b, -x.c, x.a);
```
This code is used in section 67.

18. PROPOSITION $orthodist(M)$ If x is **SL2ACJ**, then $\mathbf{S}(orthodist(x)) \supset \mathbf{L_o(S(x))}$.

⟨ Definition of $orthodist(x)$ for **SL2ACJ** x 18 ⟩ ≡
```
ACJ t = x.a * x.d + x.b * x.c;
ACJ r = ACJ(sqrt(t * t - 1));
ACJ r1 = t + r;
if (r1.f.re * r1.f.re + r1.f.im * r1.f.im ≥ 1) return t + r;
else return t - r;
```
This code is used in section 67.

19. PROPOSITION $length(M)$ If x is **SL2ACJ**, then $\mathbf{S}(length(x)) \supset \mathbf{L(S(x))}$.

⟨ Definition of $length(x)$ for **SL2ACJ** x 19 ⟩ ≡
```
ACJ t = (x.a + x.d) * 0.5;
ACJ r = ACJ(sqrt(t * t - 1));
ACJ r1 = t + r;
if (r1.f.re * r1.f.re + r1.f.im * r1.f.im ≥ 1) return (t + r) * (t + r);
else return (t - r) * (t - r);
```
This code is used in section 67.

20. PROPOSITION $notIdentity(M)$ If x is **SL2ACJ**, then $notIdentity(x)$ returns 1 implies $\pm I \notin \mathbf{S(x)}$

⟨ Definition of $notIdentity(x)$ for **SL2ACJ** x 20 ⟩ ≡
```
return absLB(x.b) > 0 ∨ absLB(x.c) > 0 ∨ (absLB(x.a - 1) > 0 ∧ absLB(x.a + 1) > 0) ∨ (absLB(x.d - 1) >
    0 ∧ absLB(x.d + 1) > 0);
```
This code is used in section 67.

21. PROPOSITION *notFPower*(M) If x is **SL2ACJ**, and *notFPower*(x) returns 1, then for all k, $\pm f^k \notin \mathbf{S}(\mathbf{M})$. We actually test a stronger condition: $M \neq \pm$*shortGenerator*(x) for all x.

⟨ Definition of *notFPower*(x) for **SL2ACJ** x 21 ⟩ ≡
  return *absLB*$(x.b) > 0 \vee$ *absLB*$(x.c) > 0$;
This code is used in section 67.

22. PROPOSITION *shortGenerator*(J) If *z1* is **ACJ**, then $\mathbf{S}($*shortGenerator*$(z)) \supset \mathbf{s}(\mathbf{S}(\mathbf{z}))$.

⟨ Definition of *shortGenerator*(z) for **ACJ** z 22 ⟩ ≡
  ACJ $sz =$ *sqrt*(z);
  ACJ *zero*(0);
  return SL2ACJ$(sz,$ *zero*$,$ *zero*$, 1/sz)$;
This code is used in section 67.

23. PROPOSITION *closeGenerator*(J, J) If x and z are **ACJ**, then $\mathbf{S}($*closeGenerator*$(x, z)) \supset \mathbf{c}(\mathbf{S}(\mathbf{x}), \mathbf{S}(\mathbf{z}))$. ∎

⟨ Definition of *closeGenerator*(x, z) for **ACJ** $x,\ z$ 23 ⟩ ≡
  ACJ $sx =$ *sqrt*$(x),\ sz =$ *sqrt*(z);
  ACJ $sh = (sx - 1/sx) * 0.5$;
  ACJ $ch = (sx + 1/sx) * 0.5$;
  return SL2ACJ$(ch * sz, sh/sz, sh * sz, ch/sz)$;
This code is used in section 67.

24. Approximate Complex 1-Jets. This section implements intervals of Complex 1-Jets. It is used to provide rigorous bounds on the values a complex function takes on a region.

Let $\mathbf{A}$ be the set of points $(z_0, z_1, z_2) \in \mathbf{C^3}$ such that $|z_k| <= 1$ for $k = 0, 1, 2$.

An **ACJ** j is a tuple (f, f_0, f_1, f_2, e) with $f, f_0, f_1, f_2 \in \mathbf{C}$ and $e \in \mathbf{R}$, which represents the class $\mathbf{S}(j)$ of functions $g : \mathbf{A} \to \mathbf{C}$ such that such that

$$|f(z_0, z_1, z_2) - (f + f_0 z_0 + f_1 z_1 + f_2 z_2)| < e$$

for all $(z_0, z_1, z_2) \in \mathbf{A}$. For convenience, we also define $\mathbf{S}(\mathbf{x}) = \{x\}$ for $x \in \mathbf{C}$

The notation suggests that $f = h(0, 0, 0)$ and $f_k = \partial_k h(0, 0, 0)$. This will usually be approximately true.

⟨ Definition of **ACJ** 24 ⟩ ≡

```
struct ACJ {
ACJ(const XComplex &ff, const XComplex &ff0 = 0, const XComplex &ff1 = 0, const
        XComplex &ff2 = 0, double err = 0) :
    f(ff), f0(ff0), f1(ff1), f2(ff2), e(err), size((1+2*EPS)*(absUB(ff0)+(absUB(ff1)+absUB(ff2))))
  { }
  XComplex f;
  XComplex f0;
  XComplex f1;
  XComplex f2;
  double e;
  double size;
};
```

This code is used in section 63.

25. PROPOSITION $-J$ If x is **ACJ**, then $\mathbf{S}(\mathbf{-x}) = -\mathbf{S}(\mathbf{x})$.

⟨ Definition of $-x$ for **ACJ** x 25 ⟩ ≡

```
return ACJ(-x.f, -x.f0, -x.f1, -x.f2, x.e);
```

This code is used in section 64.

26. PROPOSITION $J + J$ If x and y are **ACJ**, then $\mathbf{S}(\mathbf{x+y}) \supset \mathbf{S}(\mathbf{x}) + \mathbf{S}(\mathbf{y})$.

⟨ Definition of $x + y$ for **ACJ** $x,\ y$ 26 ⟩ ≡

```
AComplex r_f = x.f + y.f;
AComplex r_f0 = x.f0 + y.f0;
AComplex r_f1 = x.f1 + y.f1;
AComplex r_f2 = x.f2 + y.f2;
double r_error = (1 + 3 * EPS) * ((x.e + y.e) + ((r_f.e + r_f0.e) + (r_f1.e + r_f2.e)));
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 64.

27. PROPOSITION $J - J$ If x and y are **ACJ**, then $\mathbf{S}(\mathbf{x-y}) \supset \mathbf{S}(\mathbf{x}) - \mathbf{S}(\mathbf{y})$.

⟨ Definition of $x - y$ for **ACJ** $x,\ y$ 27 ⟩ ≡

```
AComplex r_f = x.f - y.f;
AComplex r_f0 = x.f0 - y.f0;
AComplex r_f1 = x.f1 - y.f1;
AComplex r_f2 = x.f2 - y.f2;
double r_error = (1 + 3 * EPS) * ((x.e + y.e) + ((r_f.e + r_f0.e) + (r_f1.e + r_f2.e)));
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 64.

28. PROPOSITION $J + d$ If x is **ACJ** and y is **double**, then $\mathbf{S(x{+}y)} \supset \mathbf{S(x)} + \mathbf{S(y)}$.

⟨ Definition of $x + y$ for **ACJ** x and **double** y 28 ⟩ ≡

```
AComplex r_f = x.f + y;
return ACJ(r_f.z, x.f0, x.f1, x.f2, (1 + EPS) * (x.e + r_f.e));
```

This code is used in section 64.

29. PROPOSITION $J - d$ If x is **ACJ** and y is **double**, then $\mathbf{S(x{-}y)} \supset \mathbf{S(x)} - \mathbf{S(y)}$.

⟨ Definition of $x - y$ for **ACJ** x and **double** y 29 ⟩ ≡

```
AComplex r_f = x.f - y;
return ACJ(r_f.z, x.f0, x.f1, x.f2, (1 + EPS) * (x.e + r_f.e));
```

This code is used in section 64.

30. PROPOSITION $J * J$ If x and y are **ACJ**, then $\mathbf{S(x{*}y)} \supset \mathbf{S(x)S(y)}$.

⟨ Definition of $x * y$ for **ACJ** x, y 30 ⟩ ≡

```
double xdist = size(x);
double ydist = size(y);
double ax = absUB(x.f), ay = absUB(y.f);
AComplex r_f = x.f * y.f;
AComplex r_f0 = x.f * y.f0 + x.f0 * y.f;
AComplex r_f1 = x.f * y.f1 + x.f1 * y.f;
AComplex r_f2 = x.f * y.f2 + x.f2 * y.f;
double A = (xdist + x.e) * (ydist + y.e);
double B = ax * y.e + ay * x.e;
double C = (r_f.e + r_f0.e) + (r_f1.e + r_f2.e);
double r_error = (1 + 3 * EPS) * (A + (B + C));
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 65.

31. PROPOSITION $J * d$ If x is **ACJ** and y is **double**, then $\mathbf{S(x{*}y)} \supset \mathbf{S(x)S(y)}$.

⟨ Definition of $x * y$ for **ACJ** x and **double** y 31 ⟩ ≡

```
AComplex r_f = x.f * y;
AComplex r_f0 = x.f0 * y;
AComplex r_f1 = x.f1 * y;
AComplex r_f2 = x.f2 * y;
double r_error = (1 + 3 * EPS) * ((x.e * fabs(y)) + ((r_f.e + r_f0.e) + (r_f1.e + r_f2.e)));
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 64.

32. PROPOSITION J/J If x and y are **ACJ**, then $\mathbf{S(x/y)} \supset \mathbf{S(x)/S(y)}$.

⟨ Definition of x/y for **ACJ** $x,\ y$ 32 ⟩ ≡

```
double xdist = size(x);
double ydist = size(y);
double ax = absUB(x.f), ay = absLB(y.f);
double D = ay − (1 + EPS) ∗ (y.e + ydist);
if (¬(D > 0)) return ACJ(0, 0, 0, 0, infinity( ));
AComplex den = (y.f ∗ y.f);
AComplex r_f = x.f/y.f;
AComplex r_f0 = (x.f0 ∗ y.f − x.f ∗ y.f0 )/den;
AComplex r_f1 = (x.f1 ∗ y.f − x.f ∗ y.f1 )/den;
AComplex r_f2 = (x.f2 ∗ y.f − x.f ∗ y.f2 )/den;
double A = (ax + (xdist + x.e))/D;
double B = (ax/ay + xdist/ay) + (ydist ∗ ax)/(ay ∗ ay);
double C = (r_f.e + r_f0.e) + (r_f1.e + r_f2.e);
double r_error = (1 + 3 ∗ EPS) ∗ (((1 + 3 ∗ EPS) ∗ A − (1 − 3 ∗ EPS) ∗ B) + C);
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 65.

33. PROPOSITION d/J If x is **double** and y is **ACJ**, then $\mathbf{S(x/y)} \supset \mathbf{S(x)/S(y)}$.

⟨ Definition of x/y for **double** x and **ACJ** y 33 ⟩ ≡

```
double ydist = size(y);
double ax = fabs(x), ay = absLB(y.f);
double D = ay − (1 + EPS) ∗ (y.e + ydist);
if (¬(D > 0)) return ACJ(0, 0, 0, 0, infinity( ));
AComplex den = (y.f ∗ y.f);
AComplex r_f = x/y.f;
AComplex r_f0 = (−x ∗ y.f0 )/den;
AComplex r_f1 = (−x ∗ y.f1 )/den;
AComplex r_f2 = (−x ∗ y.f2 )/den;
double B = ax/ay + (ydist ∗ ax)/(ay ∗ ay);
double C = (r_f.e + r_f0.e) + (r_f1.e + r_f2.e);
double r_error = (1 + 3 ∗ EPS) ∗ (((1 + 2 ∗ EPS) ∗ (ax/D) − (1 − 3 ∗ EPS) ∗ B) + C);
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 65.

34. PROPOSITION J/d If x is **ACJ** and y is **double**, then $\mathbf{S}(x/y) \supset \mathbf{S(x)/S(y)}$.

⟨ Definition of x/y for **ACJ** x and **double** y 34 ⟩ ≡

```
AComplex r_f = x.f/y;
AComplex r_f0 = x.f0 /y;
AComplex r_f1 = x.f1 /y;
AComplex r_f2 = x.f2 /y;
double r_error = (1 + 3 ∗ EPS) ∗ ((x.e/fabs(y)) + ((r_f.e + r_f0.e) + (r_f1.e + r_f2.e)));
return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
```

This code is used in section 64.

35. PROPOSITION *sqrt*(*J*) If x is **ACJ**, then $\mathbf{S}(sqrt(x)) \supset \sqrt{\mathbf{S}(\mathbf{x})}$.

⟨ Definition of *sqrt*(*x*) for **ACJ** *x* 35 ⟩ ≡

```
double xdist = size(x);
double ax = absUB(x.f);
double D = ax − (1 + EPS) ∗ (xdist + x.e);
if (¬(D > 0)) {
  return ACJ(0, 0, 0, 0, (1 + 2 ∗ EPS) ∗ sqrt(ax + (xdist + x.e)));
}
else {
  AComplex r_f = sqrt(x.f);
  AComplex t = r_f + r_f;
  AComplex r_f0 = AComplex(x.f0.re, x.f0.im, 0)/t;
  AComplex r_f1 = AComplex(x.f1.re, x.f1.im, 0)/t;
  AComplex r_f2 = AComplex(x.f2.re, x.f2.im, 0)/t;
  double r_error = (1 + 3 ∗ EPS) ∗ (((1 + EPS) ∗ sqrt(ax) − (1 − 3 ∗ EPS) ∗ (xdist/(2 ∗ sqrt(ax)) +
      sqrt(D))) + ((r_f.e + r_f0.e) + (r_f1.e + r_f2.e)));
  return ACJ(r_f.z, r_f0.z, r_f1.z, r_f2.z, r_error);
}
```

This code is used in section 65.

36. PROPOSITION *absUB*(*J*) If x is **ACJ**, then $\mathbf{S}(absUB(x)) >= \mathbf{S}(\mathbf{x})(\mathbf{z})$ for all $z \in \mathbf{A}$.

⟨ Definition of *absUB*(*x*) for **ACJ** *x* 36 ⟩ ≡

```
return (1 + 2 ∗ EPS) ∗ (absUB(x.f) + (size(x) + x.e));
```

This code is used in section 64.

37. PROPOSITION *absLB*(*J*) If x is **ACJ**, then $\mathbf{S}(absLB(x)) <= \mathbf{S}(\mathbf{x})(\mathbf{z})$ for all $z \in \mathbf{A}$.

⟨ Definition of *absLB*(*x*) for **ACJ** *x* 37 ⟩ ≡

```
double v = (1 − EPS) ∗ (absLB(x.f) − (1 + EPS) ∗ (size(x) + x.e));
return (v > 0) ? v : 0;
```

This code is used in section 64.

38. PROPOSITION *size*(*J*) If $(j.f, j.f0, j.f1, j.f2,\ j$. **error** $)$ is **ACJ**, then $\mathbf{S}(size(j)) \geq |\mathbf{S}(j.f0)| + |\mathbf{S}(j.f1)| + |\mathbf{S}(j.f2)|$.

⟨ Definition of *size*(*x*) for **ACJ** *x* 38 ⟩ ≡

```
return x.size;
```

This code is used in section 64.

39. Complex numbers. This section implements complex numbers. We need two kinds: **XComplex**, for complex numbers which are represented exactly, and **AComplex**, which is an interval that contains the number we wish to represent. Most of the operations here act on **XComplex** and produce **AComplex**.

40. An **XComplex** x represents the complex number $\mathbf{S}(x) = \mathbf{S}(x.re) + \mathbf{i}\mathbf{S}(x.im)$.

```
⟨ Definition of XComplex 40 ⟩ ≡
  struct XComplex {
    double re;
    double im;
  XComplex(double r = 0, double i = 0) :  re(r), im(i)
    { }
  };
```

This code is used in section 60.

41. An **AComplex** x represents the set of complex numbers $\mathbf{S}(x) = \{\mathbf{y} : |\mathbf{y} - \mathbf{x.z}| \le \mathbf{x.e}\}$.

```
⟨ Definition of AComplex 41 ⟩ ≡
  struct AComplex {
    XComplex z;
    double e;
  AComplex(double r, double i, double err): z(r, i), e(err)
    { }
  };
```

This code is used in section 60.

42. PROPOSITION $-X$ If x is **XComplex**, then $\mathbf{S}(-x) = -\mathbf{S}(x)$.

```
⟨ Definition of −x for XComplex x 42 ⟩ ≡
  return XComplex(−x.re, −x.im);
```

This code is used in section 61.

43. PROPOSITION $X + d$ If x is **XComplex** and y is **double**, then $\mathbf{S}(x + y) \supset \mathbf{S}(x) + \mathbf{S}(y)$.

```
⟨ Definition of x + y for XComplex x and double y 43 ⟩ ≡
  double re = x.re + y;
  double e = HALFEPS ∗ fabs(re);
  return AComplex(re, x.im, e);
```

This code is used in section 61.

44. PROPOSITION $X - d$ If x is **XComplex** and y is **double**, then $\mathbf{S}(x - y) \supset \mathbf{S}(x) - \mathbf{S}(y)$.

```
⟨ Definition of x − y for XComplex x and double y 44 ⟩ ≡
  double re = x.re − y;
  double e = HALFEPS ∗ fabs(re);
  return AComplex(re, x.im, e);
```

This code is used in section 61.

45. PROPOSITION $X + X$ If x and y are **XComplex**, then $\mathbf{S}(x + y) \supset \mathbf{S}(x) + \mathbf{S}(y)$.

⟨ Definition of $x + y$ for **XComplex** $x,\ y$ 45 ⟩ ≡
```
double re = x.re + y.re;
double im = x.im + y.im;
double e = HALFEPS * ((1 + EPS) * (fabs(re) + fabs(im)));
return AComplex(re, im, e);
```
This code is used in section 61.

46. PROPOSITION $X - X$ If x and y are **XComplex**, then $\mathbf{S}(x - y) \supset \mathbf{S}(x) - \mathbf{S}(y)$.

⟨ Definition of $x - y$ for **XComplex** $x,\ y$ 46 ⟩ ≡
```
double re = x.re - y.re;
double im = x.im - y.im;
double e = HALFEPS * ((1 + EPS) * (fabs(re) + fabs(im)));
return AComplex(re, im, e);
```
This code is used in section 61.

47. PROPOSITION $A + A$ If x and y are **AComplex**, then $\mathbf{S}(x + y) \supset \mathbf{S}(x) + \mathbf{S}(y)$.

⟨ Definition of $x + y$ for **AComplex** $x,\ y$ 47 ⟩ ≡
```
double re = x.z.re + y.z.re;
double im = x.z.im + y.z.im;
double e = (1 + 2 * EPS) * (HALFEPS * (fabs(re) + fabs(im)) + (x.e + y.e));
return AComplex(re, im, e);
```
This code is used in section 61.

48. PROPOSITION $A - A$ If x and y are **AComplex**, then $\mathbf{S}(x - y) \supset \mathbf{S}(x) - \mathbf{S}(y)$.

⟨ Definition of $x - y$ for **AComplex** $x,\ y$ 48 ⟩ ≡
```
double re = x.z.re - y.z.re;
double im = x.z.im - y.z.im;
double e = (1 + 2 * EPS) * (HALFEPS * (fabs(re) + fabs(im)) + (x.e + y.e));
return AComplex(re, im, e);
```
This code is used in section 61.

49. PROPOSITION $X * d$ If x is **XComplex** and y is **double**, then $\mathbf{S}(x * y) \supset \mathbf{S}(x)\mathbf{S}(y)$.

⟨ Definition of $x * y$ for **XComplex** x and **double** y 49 ⟩ ≡
```
double re = x.re * y;
double im = x.im * y;
return AComplex(re, im, HALFEPS * ((1 + EPS) * (fabs(re) + fabs(im))));
```
This code is used in section 61.

50. PROPOSITION X/d If x is **XComplex** and y is **double**, then $\mathbf{S}(x/y) \supset \mathbf{S}(x)/\mathbf{S}(y)$.

⟨ Definition of x/y for **XComplex** x and **double** y 50 ⟩ ≡
```
double re = x.re/y;
double im = x.im/y;
return AComplex(re, im, HALFEPS * ((1 + EPS) * (fabs(re) + fabs(im))));
```
This code is used in section 61.

51. PROPOSITION $X * X$ If x and y are **XComplex**, then $\mathbf{S}(x * y) \supset \mathbf{S}(x)\mathbf{S}(y)$.

⟨ Definition of $x * y$ for **XComplex** $x,\ y$ 51 ⟩ ≡
```
double re1 = x.re * y.re, re2 = x.im * y.im;
double im1 = x.re * y.im, im2 = x.im * y.re;
double e = EPS * ((1 + 2 * EPS) * ((fabs(re1) + fabs(re2)) + (fabs(im1) + fabs(im2))));
return AComplex(re1 - re2, im1 + im2, e);
```
This code is used in section 61.

52. PROPOSITION d/X If x is **double** and y is **XComplex**, then $\mathbf{S}(x/y) \supset \mathbf{S}(x)/\mathbf{S}(y)$.

⟨ Definition of x/y for **double** x and **XComplex** y 52 ⟩ ≡
```
double nrm = y.re * y.re + y.im * y.im;
double re = (x * y.re)/nrm;
double im = -(x * y.im)/nrm;
double e = (2 * EPS) * ((1 + 2 * EPS) * (fabs(re) + fabs(im)));
return AComplex(re, im, e);
```
This code is used in section 62.

53. PROPOSITION X/X If x and y are **XComplex**, then $\mathbf{S}(x/y) \supset \mathbf{S}(x)/\mathbf{S}(y)$.

⟨ Definition of x/y for **XComplex** $x,\ y$ 53 ⟩ ≡
```
double nrm = y.re * y.re + y.im * y.im;
double xryr = x.re * y.re;
double xiyi = x.im * y.im;
double xiyr = x.im * y.re;
double xryi = x.re * y.im;
double re = (xryr + xiyi)/nrm;
double im = (xiyr - xryi)/nrm;
double A = ((fabs(xryr) + fabs(xiyi)) + (fabs(xiyr) + fabs(xryi)))/nrm;
double e = (5 * HALFEPS) * ((1 + 3 * EPS) * A);
return AComplex(re, im, e);
```
This code is used in section 62.

54. PROPOSITION A/A If x and y are **AComplex**, then $\mathbf{S}(x/y) \supset \mathbf{S}(x)/\mathbf{S}(y)$.

⟨ Definition of x/y for **AComplex** $x,\ y$ 54 ⟩ ≡
```
double nrm = y.z.re * y.z.re + y.z.im * y.z.im;
double xryr = x.z.re * y.z.re;
double xiyi = x.z.im * y.z.im;
double xiyr = x.z.im * y.z.re;
double xryi = x.z.re * y.z.im;
assert(y.e * y.e < (10000 * EPS * EPS) * nrm);
double A = (fabs(xryr) + fabs(xiyi)) + (fabs(xiyr) + fabs(xryi));
double B = x.e * (fabs(y.z.re) + fabs(y.z.im)) + y.e * (fabs(x.z.re) + fabs(x.z.im));
double e = (1 + 4 * EPS) * (((5 * HALFEPS) * A + (1 + 103 * EPS) * B)/nrm);
return AComplex((xryr + xiyi)/nrm, (xiyr - xryi)/nrm, e);
```
This code is used in section 62.

55. If x is **XComplex**, then $\mathbf{S}(sqrt(x)) \supset \sqrt{\mathbf{S}(x)}$.

⟨ Definition of $sqrt(x)$ for **XComplex** x 55 ⟩ ≡
```
double s = sqrt((fabs(x.re) + hypot(x.re, x.im)) * 0.5);
double d = (x.im/s) * 0.5;
double e = EPS * ((1 + 4 * EPS) * (1.25 * s + 1.75 * fabs(d)));
if (x.re > 0.0) return AComplex(s, d, e);
else return AComplex(d, s, e);
```
This code is used in section 62.

56. If x is **XComplex**, then $\mathbf{S}(absUB(x)) \geq |\mathbf{S}(x)|$.

⟨ Definition of $absUB(x)$ for **XComplex** x 56 ⟩ ≡
```
return (1 + 2 * EPS) * hypot(x.re, x.im);
```
This code is used in section 61.

57. If x is **XComplex**, then $\mathbf{S}(absLB(x)) \leq |\mathbf{S}(x)|$.

⟨ Definition of $absLB(x)$ for **XComplex** x 57 ⟩ ≡
```
return (1 - 2 * EPS) * hypot(x.re, x.im);
```
This code is used in section 61.

58. Technical details. This section contains the C++ "glue" which tells the computer how to use the definitions of the operators on **XComplex**, **AComplex**, **ACJ** and **SL2ACJ**.

```
⟨ roundoff.h 58 ⟩ ≡
#ifndef _roundoff_h_
#define _roundoff_h_
#include <float.h>
#include <math.h>
  ⟨ Definition of EPS and HALFEPS 8 ⟩
  ⟨ Declaration of floating-point functions 9 ⟩
#endif
```

59.
```
⟨ roundoff.c 59 ⟩ ≡
#include "roundoff.h"
  ⟨ Definition of floating-point functions 10 ⟩
```

60.
```
⟨ Complex.h 60 ⟩ ≡
#ifndef _Complex_h_
#define _Complex_h_
#include <assert.h>
#include "roundoff.h"
  ⟨ Definition of XComplex 40 ⟩
  ⟨ Definition of AComplex 41 ⟩
  inline const XComplex operator−(const XComplex &x);
  inline const AComplex operator+(const AComplex &x, const AComplex &y);
  inline const AComplex operator+(const XComplex &x, const XComplex &y);
  inline const AComplex operator+(const XComplex &x, double y);
  inline const AComplex operator−(const AComplex &x, const AComplex &y);
  inline const AComplex operator−(const XComplex &x, const XComplex &y);
  inline const AComplex operator−(const XComplex &x, double y);
  inline const AComplex operator ∗ (const XComplex &x, const XComplex &y);
  inline const AComplex operator ∗ (const XComplex &x, double y);
  inline const AComplex operator/(const XComplex &x, double y);
  inline const double absLB(const XComplex &x);
  inline const double absUB(const XComplex &x);
  AComplex operator/(const AComplex &x, const AComplex &y);
  AComplex operator/(const XComplex &x, const XComplex &y);
  AComplex operator/(double x, const XComplex &y);
  AComplex sqrt(const XComplex &x);
#include "Complex.inline"
#endif
```

61. ⟨ `Complex.inline` 61 ⟩ ≡

```
inline const XComplex operator−(const XComplex &x)
{
  ⟨ Definition of −x for XComplex x 42 ⟩
}
inline const AComplex operator+(const XComplex &x, const XComplex &y)
{
  ⟨ Definition of x + y for XComplex x, y 45 ⟩
}
inline const AComplex operator+(const XComplex &x, double y)
{
  ⟨ Definition of x + y for XComplex x and double y 43 ⟩
}
inline const AComplex operator+(const AComplex &x, const AComplex &y)
{
  ⟨ Definition of x + y for AComplex x, y 47 ⟩
}
inline const AComplex operator−(const XComplex &x, const XComplex &y)
{
  ⟨ Definition of x − y for XComplex x, y 46 ⟩
}
inline const AComplex operator−(const XComplex &x, double y)
{
  ⟨ Definition of x − y for XComplex x and double y 44 ⟩
}
inline const AComplex operator−(const AComplex &x, const AComplex &y)
{
  ⟨ Definition of x − y for AComplex x, y 48 ⟩
}
inline const AComplex operator ∗ (const XComplex &x, const XComplex &y)
{
  ⟨ Definition of x ∗ y for XComplex x, y 51 ⟩
}
inline const AComplex operator ∗ (const XComplex &x, double y)
{
  ⟨ Definition of x ∗ y for XComplex x and double y 49 ⟩
}
inline const AComplex operator/(const XComplex &x, double y)
{
  ⟨ Definition of x/y for XComplex x and double y 50 ⟩
}
inline const double absLB(const XComplex &x)
{
  ⟨ Definition of absLB(x) for XComplex x 57 ⟩
}
inline const double absUB(const XComplex &x)
{
  ⟨ Definition of absUB(x) for XComplex x 56 ⟩
}
```

62. ⟨ Complex.C 62 ⟩ ≡

```
#include "Complex.h"
  AComplex operator/(const AComplex &x, const AComplex &y)
  {
    ⟨ Definition of x/y for AComplex x, y 54 ⟩
  }
  AComplex operator/(const XComplex &x, const XComplex &y)
  {
    ⟨ Definition of x/y for XComplex x, y 53 ⟩
  }
  AComplex operator/(double x, const XComplex &y)
  {
    ⟨ Definition of x/y for double x and XComplex y 52 ⟩
  }
  AComplex sqrt(const XComplex &x)
  {
    ⟨ Definition of sqrt(x) for XComplex x 55 ⟩
  }
```

63. ⟨ ACJ.h 63 ⟩ ≡

```
#ifndef _ACJ_h_
#define _ACJ_h_
#include "Complex.h"
#include <assert.h>
#include <stdio.h>
#include "roundoff.h"
  ⟨ Definition of ACJ 24 ⟩
  inline const ACJ operator-(const ACJ &x);
  inline const ACJ operator+(const ACJ &x, const ACJ &y);
  inline const ACJ operator-(const ACJ &x, const ACJ &y);
  inline const ACJ operator+(const ACJ &x, double y);
  inline const ACJ operator-(const ACJ &x, const ACJ &y);
  inline const ACJ operator*(const ACJ &x, double y);
  inline const ACJ operator/(const ACJ &x, double y);
  inline const double absUB(const ACJ &x);
  inline const double absLB(const ACJ &x);
  inline const double size(const ACJ &x);
  const ACJ operator*(const ACJ &x, const ACJ &y);
  const ACJ operator/(const ACJ &x, const ACJ &y);
  const ACJ operator/(double x, const ACJ &y);
  const ACJ sqrt(const ACJ &x);
#include "ACJ.inline"
#endif
```

64. ⟨ ACJ.inline 64 ⟩ ≡

```
inline const ACJ operator−(const ACJ &x)
{
  ⟨ Definition of −x for ACJ x 25 ⟩
}
inline const ACJ operator+(const ACJ &x, const ACJ &y)
{
  ⟨ Definition of x + y for ACJ x, y 26 ⟩
}
inline const ACJ operator−(const ACJ &x, const ACJ &y)
{
  ⟨ Definition of x − y for ACJ x, y 27 ⟩
}
inline const ACJ operator+(const ACJ &x, double y)
{
  ⟨ Definition of x + y for ACJ x and double y 28 ⟩
}
inline const ACJ operator−(const ACJ &x, double y)
{
  ⟨ Definition of x − y for ACJ x and double y 29 ⟩
}
inline const ACJ operator ∗ (const ACJ &x, double y)
{
  ⟨ Definition of x ∗ y for ACJ x and double y 31 ⟩
}
inline const ACJ operator/(const ACJ &x, double y)
{
  ⟨ Definition of x/y for ACJ x and double y 34 ⟩
}
inline const double absUB(const ACJ &x)
{
  ⟨ Definition of absUB(x) for ACJ x 36 ⟩
}
inline const double absLB(const ACJ &x)
{
  ⟨ Definition of absLB(x) for ACJ x 37 ⟩
}
inline const double size(const ACJ &x)
{
  ⟨ Definition of size(x) for ACJ x 38 ⟩
}
```

65. ⟨ ACJ.C 65 ⟩ ≡

```
#include "ACJ.h"
  const ACJ operator ∗ (const ACJ &x, const ACJ &y)
  {
    ⟨ Definition of x ∗ y for ACJ x, y 30 ⟩
  }
  const ACJ operator/(const ACJ &x, const ACJ &y)
  {
    ⟨ Definition of x/y for ACJ x, y 32 ⟩
  }
  const ACJ operator/(double x, const ACJ &y)
  {
    ⟨ Definition of x/y for double x and ACJ y 33 ⟩
  }
  const ACJ sqrt(const ACJ &x)
  {
    ⟨ Definition of sqrt(x) for ACJ x 35 ⟩
  }
```

66. ⟨ SL2ACJ.h 66 ⟩ ≡

```
#ifndef _SL2ACJ_h_
#define _SL2ACJ_h_
#include "ACJ.h"
  ⟨ Definition of SL2ACJ 15 ⟩
  const SL2ACJ operator ∗ (const SL2ACJ &x, const SL2ACJ &y);
  const SL2ACJ inverse(const SL2ACJ &x);
  const ACJ orthodist(const SL2ACJ &x);
  const ACJ length(const SL2ACJ &x);
  const int notIdentity(const SL2ACJ &x);
  const int notFPower(const SL2ACJ &x);
  const SL2ACJ shortGenerator(const ACJ &z);
  const SL2ACJ closeGenerator(const ACJ &x, const ACJ &x);
#endif
```

67. ⟨ SL2ACJ.C 67 ⟩ ≡

```
#include "SL2ACJ.h"
  const SL2ACJ operator ∗ (const SL2ACJ &x, const SL2ACJ &y)
  {
    ⟨ Definition of x ∗ y for SL2ACJ x, y 16 ⟩
  }
  const SL2ACJ inverse(const SL2ACJ &x)
  {
    ⟨ Definition of inverse(x) for SL2ACJ x 17 ⟩
  }
  const ACJ orthodist(const SL2ACJ &x)
  {
    ⟨ Definition of orthodist(x) for SL2ACJ x 18 ⟩
  }
  const ACJ length(const SL2ACJ &x)
  {
    ⟨ Definition of length(x) for SL2ACJ x 19 ⟩
  }
  const int notIdentity(const SL2ACJ &x)
  {
    ⟨ Definition of notIdentity(x) for SL2ACJ x 20 ⟩
  }
  const int notFPower(const SL2ACJ &x)
  {
    ⟨ Definition of notFPower(x) for SL2ACJ x 21 ⟩
  }
  const SL2ACJ shortGenerator(const ACJ &z)
  {
    ⟨ Definition of shortGenerator(z) for ACJ z 22 ⟩
  }
  const SL2ACJ closeGenerator(const ACJ &x, const ACJ &z)
  {
    ⟨ Definition of closeGenerator(x, z) for ACJ x, z 23 ⟩
  }
```

68. ⟨ Codes.h 68 ⟩ ≡

```
#ifndef _Codes_h_
#define Codes_h_
#include "roundoff.h"
#include "SL2ACJ.h"
  int inequalityHolds(const char ∗code, const char ∗where, int depth);
  SL2ACJ evaluateWord(const char ∗word, const ACJ &along, const ACJ &ortho, const ACJ
      &whirle);
  int wordImpliesCommuting(const char ∗word);
#endif
```

69. ⟨ Codes.C 69 ⟩ ≡
```
#include "Codes.h"
#include "SL2ACJ.h"
#include "roundoff.h"
#include <stdio.h>
```
⟨ Definition of *inequalityHolds* 11 ⟩
⟨ Definition of *evaluateWord* 13 ⟩
⟨ Definition of *wordImpliesCommuting* 14 ⟩

70. ⟨ verify.C 70 ⟩ ≡
```
#include "Codes.h"
#include <stdio.h>
#include "roundoff.h"
```
⟨ Definition of MAXDEPTH 7 ⟩
⟨ Definition of *inequalityFor* 5 ⟩
⟨ Definition of *verify* 4 ⟩
⟨ Definition of *main* 6 ⟩

VERIFY