

Pell's Equation

Manuel Eberl

March 17, 2025

Abstract

This article gives the basic theory of Pell's equation $x^2 = 1 + Dy^2$, where $D \in \mathbb{N}$ is a parameter and x, y are integer variables.

The main result that is proven is the following: If D is not a perfect square, then there exists a *fundamental solution* (x_0, y_0) that is not the trivial solution $(1, 0)$ and which generates all other solutions (x, y) in the sense that there exists some $n \in \mathbb{N}$ such that $|x| + |y|\sqrt{D} = (x_0 + y_0\sqrt{D})^n$. This also implies that the set of solutions is infinite, and it gives us an explicit and executable characterisation of all the solutions.

Based on this, simple executable algorithms for computing the fundamental solution and the infinite sequence of all non-negative solutions are also provided.

Contents

1 Efficient Algorithms for the Square Root on \mathbb{N}	3
1.1 A Discrete Variant of Heron’s Algorithm	3
1.2 Square Testing	4
2 Pell’s equation	6
2.1 Preliminary facts	7
2.2 The case of a perfect square	8
2.3 Existence of a non-trivial solution	8
2.4 Definition of solutions	9
2.5 The Pell valuation function	11
2.6 Linear ordering of solutions	11
2.7 The fundamental solution	12
2.8 Group structure on solutions	12
2.9 The different regions of the valuation function	15
2.10 Generating property of the fundamental solution	16
2.11 The case of an “almost square” parameter	17
2.12 Alternative presentation of the main results	18
2.13 Executable code	18
2.13.1 Efficient computation of powers by squaring	18
2.13.2 Multiplication and powers of solutions	19
2.13.3 Finding the fundamental solution	19
2.13.4 The infinite list of all solutions	20
2.13.5 Computing the n -th solution	20
2.13.6 Tests	21

```

theory Efficient-Discrete-Sqrt
imports
  Complex-Main
  HOL-Computational-Algebra.Computational-Algebra
  HOL-Library.Discrete-Functions
  HOL-Library.Tree
  HOL-Library.IArray
begin

```

1 Efficient Algorithms for the Square Root on \mathbb{N}

1.1 A Discrete Variant of Heron's Algorithm

An algorithm for calculating the discrete square root, taken from Cohen [2]. This algorithm is essentially a discretised variant of Heron's method or Newton's method specialised to the square root function.

```

lemma sqrt-eq-floor-sqrt: floor-sqrt n = nat ⌊sqrt n⌋
⟨proof⟩

fun newton-sqrt-aux :: nat ⇒ nat ⇒ nat where
  newton-sqrt-aux x n =
    (let y = (x + n div x) div 2
     in if y < x then newton-sqrt-aux y n else x)

declare newton-sqrt-aux.simps [simp del]

lemma newton-sqrt-aux-simps:
  (x + n div x) div 2 < x ⟹ newton-sqrt-aux x n = newton-sqrt-aux ((x + n div
  x) div 2) n
  (x + n div x) div 2 ≥ x ⟹ newton-sqrt-aux x n = x
⟨proof⟩

lemma heron-step-real: ⟦t > 0; n ≥ 0⟧ ⟹ (t + n/t) / 2 ≥ sqrt n
⟨proof⟩

lemma heron-step-div-eq-floored:
  (t::nat) > 0 ⟹ (t + (n::nat) div t) div 2 = nat ⌊(t + n/t) / 2⌋
⟨proof⟩

lemma heron-step: t > 0 ⟹ (t + n div t) div 2 ≥ floor-sqrt n
⟨proof⟩

lemma newton-sqrt-aux-correct:
  assumes x ≥ floor-sqrt n
  shows newton-sqrt-aux x n = floor-sqrt n
⟨proof⟩

```

```

definition newton-sqrt :: nat  $\Rightarrow$  nat where
  newton-sqrt n = newton-sqrt-aux n n

declare floor-sqrt-code [code del]

theorem Discrete-sqrt-eq-newton-sqrt [code]: floor-sqrt n = newton-sqrt n
  ⟨proof⟩

```

1.2 Square Testing

Next, we implement an algorithm to determine whether a given natural number is a perfect square, as described by Cohen [2]. Essentially, the number first determines whether the number is a square. Essentially

$\langle proof \rangle$

lemma *sub-q63-array*: $i \in \{.. < 63\} \implies IArray.\text{sub_q63_array } i \longleftrightarrow i \in q63$
 $\langle proof \rangle$

lemma *sub-q64-array*: $i \in \{.. < 64\} \implies IArray.\text{sub_q64_array } i \longleftrightarrow i \in q64$
 $\langle proof \rangle$

lemma *sub-q65-array*: $i \in \{.. < 65\} \implies IArray.\text{sub_q65_array } i \longleftrightarrow i \in q65$
 $\langle proof \rangle$

lemma *in-q11-code*: $x \bmod 11 \in q11 \longleftrightarrow IArray.\text{sub_q11_array } (x \bmod 11)$
 $\langle proof \rangle$

lemma *in-q63-code*: $x \bmod 63 \in q63 \longleftrightarrow IArray.\text{sub_q63_array } (x \bmod 63)$
 $\langle proof \rangle$

lemma *in-q64-code*: $x \bmod 64 \in q64 \longleftrightarrow IArray.\text{sub_q64_array } (x \bmod 64)$
 $\langle proof \rangle$

lemma *in-q65-code*: $x \bmod 65 \in q65 \longleftrightarrow IArray.\text{sub_q65_array } (x \bmod 65)$
 $\langle proof \rangle$

definition *square-test* :: *nat* \Rightarrow *bool* **where**
square-test $n =$
 $(n \bmod 64 \in q64 \wedge (\text{let } r = n \bmod 45045 \text{ in}$
 $r \bmod 63 \in q63 \wedge r \bmod 65 \in q65 \wedge r \bmod 11 \in q11 \wedge n = (\text{floor-sqrt } n)^2))$

lemma *square-test-code* [*code*]:
square-test $n =$
 $(IArray.\text{sub_q64_array } (n \bmod 64) \wedge (\text{let } r = n \bmod 45045 \text{ in}$
 $IArray.\text{sub_q63_array } (r \bmod 63) \wedge$
 $IArray.\text{sub_q65_array } (r \bmod 65) \wedge$
 $IArray.\text{sub_q11_array } (r \bmod 11) \wedge n = (\text{floor-sqrt } n)^2))$
 $\langle proof \rangle$

lemma *square-mod-lower*: $m > 0 \implies (q^2 :: \text{nat}) \bmod m = a \implies \exists q' < m. q^2 \bmod m = a$
 $\langle proof \rangle$

lemma *q11-upto-def*: $q11 = (\lambda k. k^2 \bmod 11) ` \{.. < 11\}$
 $\langle proof \rangle$

lemma *q11-infinite-def*: $q11 = (\lambda k. k^2 \bmod 11) ` \{0..\}$
 $\langle proof \rangle$

lemma *q63-upto-def*: $q63 = (\lambda k. k^2 \bmod 63) ` \{.. < 63\}$

```

⟨proof⟩

lemma q63-infinite-def:  $q63 = (\lambda k. k^2 \bmod 63) ` \{0..\}$ 
⟨proof⟩

lemma q64-upto-def:  $q64 = (\lambda k. k^2 \bmod 64) ` \{.. < 64\}$ 
⟨proof⟩

lemma q64-infinite-def:  $q64 = (\lambda k. k^2 \bmod 64) ` \{0..\}$ 
⟨proof⟩

lemma q65-upto-def:  $q65 = (\lambda k. k^2 \bmod 65) ` \{.. < 65\}$ 
⟨proof⟩

lemma q65-infinite-def:  $q65 = (\lambda k. k^2 \bmod 65) ` \{0..\}$ 
⟨proof⟩

lemma square-mod-existence:
  fixes n k :: nat
  assumes  $\exists q. q^2 = n$ 
  shows  $\exists q. n \bmod k = q^2 \bmod k$ 
⟨proof⟩

theorem square-test-correct: square-test n  $\longleftrightarrow$  is-square n
⟨proof⟩

definition get-nat-sqrt :: nat  $\Rightarrow$  nat option
  where get-nat-sqrt n = (if is-square n then Some (floor-sqrt n) else None)

lemma get-nat-sqrt-code [code]:
  get-nat-sqrt n =
    (if IArray.sub q64-array (n mod 64)  $\wedge$  (let r = n mod 45045 in
      IArray.sub q63-array (r mod 63)  $\wedge$ 
      IArray.sub q65-array (r mod 65)  $\wedge$ 
      IArray.sub q11-array (r mod 11)) then
      (let x = floor-sqrt n in if  $x^2 = n$  then Some x else None) else None)
    ⟨proof⟩

  end

```

2 Pell's equation

```

theory Pell
imports
  Complex-Main
  HOL-Computational-Algebra.Computational-Algebra
begin

```

Pell's equation has the general form $x^2 = 1 + Dy^2$ where $D \in \mathbb{N}$ is a parameter and x, y are \mathbb{Z} -valued variables. As we will see, that case where D is a perfect square is trivial and therefore uninteresting; we will therefore assume that D is not a perfect square for the most part.

Furthermore, it is obvious that the solutions to Pell's equation are symmetric around the origin in the sense that (x, y) is a solution iff $(\pm x, \pm y)$ is a solution. We will therefore mostly look at solutions (x, y) where both x and y are non-negative, since the remaining solutions are a trivial consequence of these.

Information on the material treated in this formalisation can be found in many textbooks and lecture notes, e.g. [3, 1].

2.1 Preliminary facts

lemma *gcd-int-nonpos-iff* [*simp*]: $\text{gcd } x \ (y :: \text{int}) \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *minus-in-Ints-iff* [*simp*]:
 $-\mathit{x} \in \mathbb{Z} \longleftrightarrow \mathit{x} \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

A (positive) square root of a natural number is either a natural number or irrational.

lemma *nonneg-sqrt-nat-or-irrat*:
assumes $x^{\wedge} 2 = \text{real } a$ **and** $x \geq 0$
shows $x \in \mathbb{N} \vee x \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

A square root of a natural number is either an integer or irrational.

corollary *sqrt-nat-or-irrat*:
assumes $x^{\wedge} 2 = \text{real } a$
shows $x \in \mathbb{Z} \vee x \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

corollary *sqrt-nat-or-irrat'*:
 $\text{sqrt } (\text{real } a) \in \mathbb{N} \vee \text{sqrt } (\text{real } a) \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

The square root of a natural number n is again a natural number iff n is a perfect square.

corollary *sqrt-nat-iff-is-square*:
 $\text{sqrt } (\text{real } n) \in \mathbb{N} \longleftrightarrow \text{is-square } n$
 $\langle \text{proof} \rangle$

corollary *irrat-sqrt-nonsquare*: $\neg \text{is-square } n \implies \text{sqrt } (\text{real } n) \notin \mathbb{Q}$
 $\langle \text{proof} \rangle$

2.2 The case of a perfect square

As we have noted, the case where D is a perfect square is trivial: In fact, we will show that the only solutions in this case are the trivial solutions $(x, y) = (\pm 1, 0)$ if D is a non-zero perfect square, or $(\pm 1, y)$ for arbitrary $y \in \mathbb{Z}$ if $D = 0$.

```

context
  fixes D :: nat
  assumes square-D: is-square D
begin

lemma pell-square-solution-nat-aux:
  fixes x y :: nat
  assumes D > 0 and x ^ 2 = 1 + D * y ^ 2
  shows (x, y) = (1, 0)
  ⟨proof⟩

lemma pell-square-solution-int-aux:
  fixes x y :: int
  assumes D > 0 and x ^ 2 = 1 + D * y ^ 2
  shows x ∈ {-1, 1} ∧ y = 0
  ⟨proof⟩

lemma pell-square-solution-nat-iff:
  fixes x y :: nat
  shows x ^ 2 = 1 + D * y ^ 2 ↔ x = 1 ∧ (D = 0 ∨ y = 0)
  ⟨proof⟩

lemma pell-square-solution-int-iff:
  fixes x y :: int
  shows x ^ 2 = 1 + D * y ^ 2 ↔ x ∈ {-1, 1} ∧ (D = 0 ∨ y = 0)
  ⟨proof⟩

end

```

2.3 Existence of a non-trivial solution

Let us now turn to the case where D is not a perfect square.

We first show that Pell's equation always has at least one non-trivial solution (apart from the trivial solution $(1, 0)$). For this, we first need a lemma about the existence of rational approximations of real numbers.

The following lemma states that for any positive integer s and real number x , we can find a rational approximation t / u to x with an error of most $1 / (u * s)$ where the denominator u is at most s .

```

lemma pell-approximation-lemma:
  fixes s :: nat and x :: real
  assumes s: s > 0

```

```

shows  $\exists u::nat. \exists t::int. u > 0 \wedge coprime u t \wedge 1 / s \in \{|t - u * x| <.. 1 / u\}$ 
⟨proof⟩

```

As a simple corollary of this, we can show that for irrational x , there is an infinite number of rational approximations t / u to x whose error is less than $1 / u^2$.

corollary *pell-approximation-corollary*:

```

fixes  $x :: real$ 
assumes  $x \notin \mathbb{Q}$ 
shows infinite  $\{(t :: int, u :: nat). u > 0 \wedge coprime u t \wedge |t - u * x| < 1 / u\}$ 
  (is infinite ?A)
⟨proof⟩

```

```

locale pell =
fixes  $D :: nat$ 
assumes nonsquare-D:  $\neg is-square D$ 
begin

lemma D-gt-1:  $D > 1$ 
⟨proof⟩

lemma D-pos:  $D > 0$ 
⟨proof⟩

```

With the above corollary, we can show the existence of a non-trivial solution. We restrict our attention to solutions (x, y) where both x and y are non-negative.

```

theorem pell-solution-exists:  $\exists (x::nat) (y::nat). y \neq 0 \wedge x^2 = 1 + D * y^2$ 
⟨proof⟩

```

2.4 Definition of solutions

We define some abbreviations for the concepts of a solution and a non-trivial solution.

```

definition solution :: ('a × 'a :: comm-semiring-1) ⇒ bool where
  solution =  $(\lambda(a, b). a^2 = 1 + of-nat D * b^2)$ 

```

```

definition nontriv-solution :: ('a × 'a :: comm-semiring-1) ⇒ bool where
  nontriv-solution =  $(\lambda(a, b). (a, b) \neq (1, 0) \wedge a^2 = 1 + of-nat D * b^2)$ 

```

```

lemma nontriv-solution-altdef: nontriv-solution  $z \longleftrightarrow$  solution  $z \wedge z \neq (1, 0)$ 
⟨proof⟩

```

```

lemma solution-trivial-nat [simp, intro]: solution (Suc 0, 0)
⟨proof⟩

```

```

lemma solution-trivial [simp, intro]: solution (1, 0)

```

$\langle proof \rangle$

lemma *solution-uminus-left* [simp]: *solution* ($-x, y :: 'a :: \text{comm-ring-1}$) \longleftrightarrow *solution* (x, y)
 $\langle proof \rangle$

lemma *solution-uminus-right* [simp]: *solution* ($x, -y :: 'a :: \text{comm-ring-1}$) \longleftrightarrow *solution* (x, y)
 $\langle proof \rangle$

lemma *solution-0-snd-nat-iff* [simp]: *solution* ($a :: \text{nat}, 0$) \longleftrightarrow $a = 1$
 $\langle proof \rangle$

lemma *solution-0-snd-iff* [simp]: *solution* ($a :: 'a :: \text{idom}, 0$) \longleftrightarrow $a \in \{1, -1\}$
 $\langle proof \rangle$

lemma *no-solution-0-fst-nat* [simp]: $\neg \text{solution} (0, b :: \text{nat})$
 $\langle proof \rangle$

lemma *no-solution-0-fst-int* [simp]: $\neg \text{solution} (0, b :: \text{int})$
 $\langle proof \rangle$

lemma *solution-of-nat-of-nat* [simp]:
solution (*of-nat* $a, \text{of-nat } b :: 'a :: \{\text{comm-ring-1}, \text{ring-char-0}\}$) \longleftrightarrow *solution* (a, b)
 $\langle proof \rangle$

lemma *solution-of-nat-of-nat'* [simp]:
solution (*case z of* (a, b) \Rightarrow (*of-nat* $a, \text{of-nat } b :: 'a :: \{\text{comm-ring-1}, \text{ring-char-0}\}$))
 \longleftrightarrow
solution z
 $\langle proof \rangle$

lemma *solution-nat-abs-nat-abs* [simp]:
solution (*nat* $|x|, \text{nat } |y|$) \longleftrightarrow *solution* (x, y)
 $\langle proof \rangle$

lemma *nontriv-solution-of-nat-of-nat* [simp]:
nontriv-solution (*of-nat* $a, \text{of-nat } b :: 'a :: \{\text{comm-ring-1}, \text{ring-char-0}\}$) \longleftrightarrow
nontriv-solution (a, b)
 $\langle proof \rangle$

lemma *nontriv-solution-of-nat-of-nat'* [simp]:
nontriv-solution (*case z of* (a, b) \Rightarrow (*of-nat* $a, \text{of-nat } b :: 'a :: \{\text{comm-ring-1}, \text{ring-char-0}\}$)) \longleftrightarrow
nontriv-solution z
 $\langle proof \rangle$

lemma *nontriv-solution-imp-solution* [dest]: *nontriv-solution* $z \implies \text{solution } z$

$\langle proof \rangle$

2.5 The Pell valuation function

Solutions (x, y) have an interesting correspondence to the ring $\mathbb{Z}[\sqrt{D}]$ via the map $(x, y) \mapsto x + y\sqrt{D}$. We call this map the *Pell valuation function*. It is obvious that this map is injective, since \sqrt{D} is irrational.

definition *pell-valuation* :: int × int ⇒ real **where**
 $pell\text{-valuation} = (\lambda(a,b). a + b * sqrt D)$

lemma *pell-valuation-nonneg* [simp]: $fst z \geq 0 \implies snd z \geq 0 \implies pell\text{-valuation } z \geq 0$
 $\langle proof \rangle$

lemma *pell-valuation-uminus-uminus* [simp]: $pell\text{-valuation } (-x, -y) = -pell\text{-valuation } (x, y)$
 $\langle proof \rangle$

lemma *pell-valuation-eq-iff* [simp]:
 $pell\text{-valuation } z1 = pell\text{-valuation } z2 \longleftrightarrow z1 = z2$
 $\langle proof \rangle$

2.6 Linear ordering of solutions

Next, we show that solutions are linearly ordered w. r. t. the pointwise order on products. This means that for two different solutions (a, b) and (x, y) , we always either have $a < x$ and $b < y$ or $a > x$ and $b > y$.

lemma *solutions-linorder*:
fixes $a b x y :: nat$
assumes *solution* (a, b) *solution* (x, y)
shows $a \leq x \wedge b \leq y \vee a \geq x \wedge b \geq y$
 $\langle proof \rangle$

lemma *solutions-linorder-strict*:
fixes $a b x y :: nat$
assumes *solution* (a, b) *solution* (x, y)
shows $(a, b) = (x, y) \vee a < x \wedge b < y \vee a > x \wedge b > y$
 $\langle proof \rangle$

lemma *solutions-le-iff-pell-valuation-le*:
fixes $a b x y :: nat$
assumes *solution* (a, b) *solution* (x, y)
shows $a \leq x \wedge b \leq y \longleftrightarrow pell\text{-valuation } (a, b) \leq pell\text{-valuation } (x, y)$
 $\langle proof \rangle$

lemma *solutions-less-iff-pell-valuation-less*:
fixes $a b x y :: nat$
assumes *solution* (a, b) *solution* (x, y)

shows $a < x \wedge b < y \longleftrightarrow \text{pell-valuation } (a, b) < \text{pell-valuation } (x, y)$
 $\langle \text{proof} \rangle$

2.7 The fundamental solution

The *fundamental solution* is the non-trivial solution (x, y) with non-negative x and y for which the Pell valuation $x + y\sqrt{D}$ is minimal, or, equivalently, for which x and y are minimal.

definition $\text{fund-sol} :: \text{nat} \times \text{nat}$ **where**

$\text{fund-sol} = (\text{THE } z :: \text{nat} \times \text{nat}. \text{ is-arg-min } (\text{pell-valuation} :: \text{nat} \times \text{nat} \Rightarrow \text{real}) \text{ nontriv-solution } z)$

The well-definedness of this follows from the injectivity of the Pell valuation and the fact that smaller Pell valuation of a solution is smaller than that of another iff the components are both smaller.

theorem $\text{fund-sol-is-arg-min}:$

$\text{is-arg-min } (\text{pell-valuation} :: \text{nat} \times \text{nat} \Rightarrow \text{real}) \text{ nontriv-solution fund-sol}$
 $\langle \text{proof} \rangle$

corollary

$\text{fund-sol-is-nontriv-solution}: \text{nontriv-solution fund-sol}$

and $\text{fund-sol-minimal}:$

$\text{nontriv-solution } (a, b) \implies \text{pell-valuation fund-sol} \leq \text{pell-valuation } (\text{int } a, \text{int } b)$

and $\text{fund-sol-minimal}':$

$\text{nontriv-solution } (z :: \text{nat} \times \text{nat}) \implies \text{pell-valuation fund-sol} \leq \text{pell-valuation } z$

$\langle \text{proof} \rangle$

lemma $\text{fund-sol-minimal}'':$

assumes $\text{nontriv-solution } z$

shows $\text{fst fund-sol} \leq \text{fst } z \text{ snd fund-sol} \leq \text{snd } z$
 $\langle \text{proof} \rangle$

2.8 Group structure on solutions

As was mentioned already, the Pell valuation function provides an injective map from solutions of Pell's equation into the ring $\mathbb{Z}[\sqrt{D}]$. We shall see now that the solutions are actually a subgroup of the multiplicative group of $\mathbb{Z}[\sqrt{D}]$ via the valuation function as a homomorphism:

- The trivial solution $(1, 0)$ has valuation 1, which is the neutral element of $\mathbb{Z}[\sqrt{D}]^*$
- Multiplication of two solutions $a+b\sqrt{D}$ and $x+y\sqrt{D}$ leads to $\bar{x}+\bar{y}\sqrt{D}$ with $\bar{x} = xa + ybD$ and $\bar{y} = xb + ya$, which is again a solution.

- The conjugate $(x, -y)$ of a solution (x, y) is an inverse element to this multiplication operation, since $(x + y\sqrt{D})(x - y\sqrt{D}) = 1$.

definition *pell-mul* :: ('*a* :: comm-semiring-1 × '*a*) ⇒ ('*a* × '*a*) ⇒ ('*a* × '*a*)

where

$$\textit{pell-mul} = (\lambda(a,b) (x,y). (x * a + y * b * \text{of-nat } D, x * b + y * a))$$

definition *pell-cnj* :: ('*a* :: comm-ring-1 × '*a*) ⇒ '*a* × '*a* **where**

$$\textit{pell-cnj} = (\lambda(a,b). (a, -b))$$

lemma *pell-cnj-snd-0* [simp]: *snd z = 0* \implies *pell-cnj z = z*
⟨proof⟩

lemma *pell-mul-commutes*: *pell-mul z1 z2 = pell-mul z2 z1*
⟨proof⟩

lemma *pell-mul-assoc*: *pell-mul z1 (pell-mul z2 z3) = pell-mul (pell-mul z1 z2) z3*
⟨proof⟩

lemma *pell-mul-trivial-left* [simp]: *pell-mul (1, 0) z = z*
⟨proof⟩

lemma *pell-mul-trivial-right* [simp]: *pell-mul z (1, 0) = z*
⟨proof⟩

lemma *pell-mul-trivial-left-nat* [simp]: *pell-mul (Suc 0, 0) z = z*
⟨proof⟩

lemma *pell-mul-trivial-right-nat* [simp]: *pell-mul z (Suc 0, 0) = z*
⟨proof⟩

definition *pell-power* :: ('*a* :: comm-semiring-1 × '*a*) ⇒ nat ⇒ ('*a* × '*a*) **where**
 $\textit{pell-power } z \ n = ((\lambda z'. \textit{pell-mul } z' \ z) \ \wedge^n \ (1, 0))$

lemma *pell-power-0* [simp]: *pell-power z 0 = (1, 0)*
⟨proof⟩

lemma *pell-power-one* [simp]: *pell-power (1, 0) n = (1, 0)*
⟨proof⟩

lemma *pell-power-one-right* [simp]: *pell-power z 1 = z*
⟨proof⟩

lemma *pell-power-Suc*: *pell-power z (Suc n) = pell-mul z (pell-power z n)*
⟨proof⟩

lemma *pell-power-add*: *pell-power z (m + n) = pell-mul (pell-power z m) (pell-power z n)*
⟨proof⟩

```

lemma pell-valuation-mult [simp]:
  pell-valuation (pell-mul z1 z2) = pell-valuation z1 * pell-valuation z2
  <proof>

lemma pell-valuation-mult-nat [simp]:
  pell-valuation (case pell-mul z1 z2 of (a, b) => (int a, int b)) =
    pell-valuation z1 * pell-valuation z2
  <proof>

lemma pell-valuation-trivial [simp]: pell-valuation (1, 0) = 1
  <proof>

lemma pell-valuation-trivial-nat [simp]: pell-valuation (Suc 0, 0) = 1
  <proof>

lemma pell-valuation-cnj: pell-valuation (pell-cnj z) = fst z - snd z * sqrt D
  <proof>

lemma pell-valuation-snd-0 [simp]: pell-valuation (a, 0) = of-int a
  <proof>

lemma pell-valuation-0-iff [simp]: pell-valuation z = 0 <=> z = (0, 0)
  <proof>

lemma pell-valuation-solution-pos-nat:
  fixes z :: nat × nat
  assumes solution z
  shows pell-valuation z > 0
  <proof>

lemma
  assumes solution z
  shows pell-mul-cnj-right: pell-mul z (pell-cnj z) = (1, 0)
  and pell-mul-cnj-left: pell-mul (pell-cnj z) z = (1, 0)
  <proof>

lemma pell-valuation-cnj-solution:
  fixes z :: nat × nat
  assumes solution z
  shows pell-valuation (pell-cnj z) = 1 / pell-valuation z
  <proof>

lemma pell-valuation-power [simp]: pell-valuation (pell-power z n) = pell-valuation
z ^ n
  <proof>

lemma pell-valuation-power-nat [simp]:
  pell-valuation (case pell-power z n of (a, b) => (int a, int b)) = pell-valuation z ^

```

n
 $\langle proof \rangle$

lemma *pell-valuation-fund-sol-ge-2*: *pell-valuation fund-sol* ≥ 2
 $\langle proof \rangle$

lemma *solution-pell-mul* [intro]:
assumes *solution z1 solution z2*
shows *solution (pell-mul z1 z2)*
 $\langle proof \rangle$

lemma *solution-pell-cnj* [intro]:
assumes *solution z*
shows *solution (pell-cnj z)*
 $\langle proof \rangle$

lemma *solution-pell-power* [simp, intro]: *solution z* \implies *solution (pell-power z n)*
 $\langle proof \rangle$

lemma *pell-mul-eq-trivial-nat-iff*:
pell-mul z1 z2 = (Suc 0, 0) \longleftrightarrow z1 = (Suc 0, 0) \wedge z2 = (Suc 0, 0)
 $\langle proof \rangle$

lemma *nontriv-solution-pell-nat-mul1*:
solution (z1 :: nat \times nat) \implies nontriv-solution z2 \implies nontriv-solution (pell-mul z1 z2)
 $\langle proof \rangle$

lemma *nontriv-solution-pell-nat-mul2*:
nontriv-solution (z1 :: nat \times nat) \implies solution z2 \implies nontriv-solution (pell-mul z1 z2)
 $\langle proof \rangle$

lemma *nontriv-solution-power-nat* [intro]:
assumes *nontriv-solution (z :: nat \times nat) n > 0*
shows *nontriv-solution (pell-power z n)*
 $\langle proof \rangle$

2.9 The different regions of the valuation function

Next, we shall explore what happens to the valuation function for solutions (x, y) for different signs of x and y :

- If $x > 0$ and $y > 0$, we have $x + y\sqrt{D} > 1$.
- If $x > 0$ and $y < 0$, we have $0 < x + y\sqrt{D} < 1$.
- If $x < 0$ and $y > 0$, we have $-1 < x + y\sqrt{D} < 0$.

- If $x < 0$ and $y < 0$, we have $x + y\sqrt{D} < -1$.

In particular, this means that we can deduce the sign of x and y if we know in which of these four regions the valuation lies.

lemma

assumes $x > 0 y > 0$ solution (x, y)
shows pell-valuation-pos-pos: pell-valuation $(x, y) > 1$
and pell-valuation-pos-neg-aux: pell-valuation $(x, -y) \in \{0 < .. < 1\}$
 $\langle proof \rangle$

lemma pell-valuation-pos-neg:

assumes $x > 0 y < 0$ solution (x, y)
shows pell-valuation $(x, y) \in \{0 < .. < 1\}$
 $\langle proof \rangle$

lemma pell-valuation-neg-neg:

assumes $x < 0 y < 0$ solution (x, y)
shows pell-valuation $(x, y) < -1$
 $\langle proof \rangle$

lemma pell-valuation-neg-pos:

assumes $x < 0 y > 0$ solution (x, y)
shows pell-valuation $(x, y) \in \{-1 < .. < 0\}$
 $\langle proof \rangle$

lemma pell-valuation-solution-gt1D:

assumes solution z pell-valuation $z > 1$
shows fst $z > 0 \wedge$ snd $z > 0$
 $\langle proof \rangle$

2.10 Generating property of the fundamental solution

We now show that the fundamental solution generates the set of the (non-negative) solutions in the sense that each solution is a power of the fundamental solution. Combined with the symmetry property that (x, y) is a solution iff $(\pm x, \pm y)$ is a solution, this gives us a complete characterisation of all solutions of Pell's equation.

definition nth-solution :: nat \Rightarrow nat \times nat where
nth-solution $n =$ pell-power fund-sol n

lemma pell-valuation-nth-solution [simp]:
pell-valuation (nth-solution n) = pell-valuation fund-sol \wedge^n
 $\langle proof \rangle$

theorem nth-solution-inj: inj nth-solution
 $\langle proof \rangle$

```

theorem nth-solution-sound [intro]: solution (nth-solution n)
  <proof>

theorem nth-solution-sound' [intro]: n > 0  $\implies$  nontriv-solution (nth-solution n)
  <proof>

theorem nth-solution-complete:
  fixes z :: nat × nat
  assumes solution z
  shows z ∈ range nth-solution
  <proof>

corollary solution-iff-nth-solution:
  fixes z :: nat × nat
  shows solution z  $\longleftrightarrow$  z ∈ range nth-solution
  <proof>

corollary solution-iff-nth-solution':
  fixes z :: int × int
  shows solution (a, b)  $\longleftrightarrow$  (nat |a|, nat |b|) ∈ range nth-solution
  <proof>

corollary infinite-solutions: infinite {z :: nat × nat. solution z}
  <proof>

corollary infinite-solutions': infinite {z :: int × int. solution z}
  <proof>

lemma strict-mono-pell-valuation-nth-solution: strict-mono (pell-valuation ∘ nth-solution)
  <proof>

lemma strict-mono-nth-solution:
  strict-mono (fst ∘ nth-solution) strict-mono (snd ∘ nth-solution)
  <proof>

end

```

2.11 The case of an “almost square” parameter

If D is equal to $a^2 - 1$ for some $a > 1$, we have a particularly simple case where the fundamental solution is simply $(1, a)$.

```

context
  fixes a :: nat
  assumes a: a > 1
begin

lemma pell-square-minus1: pell (a2 - Suc 0)
  <proof>

```

```

interpretation pell a2 = Suc 0
  ⟨proof⟩

lemma fund-sol-square-minus1: fund-sol = (a, 1)
  ⟨proof⟩

end

```

2.12 Alternative presentation of the main results

```

theorem pell-solutions:
  fixes D :: nat
  assumes ∄ k. D = k2
  obtains x0 y0 :: nat
  where ∀(x:int) (y:int).
    x2 − D * y2 = 1 ↔
    (∃ n:nat. nat |x| + sqrt D * nat |y| = (x0 + sqrt D * y0) ^ n)
  ⟨proof⟩

```

```

corollary pell-solutions-infinite:
  fixes D :: nat
  assumes ∄ k. D = k2
  shows infinite {(x :: int, y :: int). x2 − D * y2 = 1}
  ⟨proof⟩

```

```
end
```

2.13 Executable code

```

theory Pell-Algorithm
imports
  Pell
  Efficient-Discrete-Sqrt
  HOL-Library.Discrete-Functions
  HOL-Library.While-Combinator
  HOL-Library.Stream
begin

```

2.13.1 Efficient computation of powers by squaring

The following is a tail-recursive implementation of exponentiation by squaring. It works for any binary operation f that fulfills $f x (f x z) = f (f x x) z$, i. e. some weak form of associativity.

```

context
  fixes f :: 'a ⇒ 'a ⇒ 'a
begin

function efficient-power :: 'a ⇒ 'a ⇒ nat ⇒ 'a where

```

```

efficient-power y x 0 = y
| efficient-power y x (Suc 0) = f x y
| n ≠ 0 ⇒ even n ⇒ efficient-power y x n = efficient-power y (f x x) (n div 2)
| n ≠ 1 ⇒ odd n ⇒ efficient-power y x n = efficient-power (f x y) (f x x) (n div
2)
⟨proof⟩
termination ⟨proof⟩

lemma efficient-power-code [code]:
efficient-power y x n =
(if n = 0 then y
else if n = 1 then f x y
else if even n then efficient-power y (f x x) (n div 2)
else efficient-power (f x y) (f x x) (n div 2))
⟨proof⟩

lemma efficient-power-correct:
assumes ∀x z. f x (f x z) = f (f x x) z
shows efficient-power y x n = (f x ^ n) y
⟨proof⟩

end

```

2.13.2 Multiplication and powers of solutions

We define versions of Pell solution multiplication and exponentiation specialised to natural numbers, both for efficiency reasons and to circumvent the problem of generating code for definitions made inside locales.

```

fun pell-mul-nat :: nat ⇒ nat × nat ⇒ - where
pell-mul-nat D (a, b) (x, y) = (a * x + D * b * y, a * y + b * x)

lemma (in pell) pell-mul-nat-correct [simp]: pell-mul-nat D = pell.pell-mul D
⟨proof⟩

definition efficient-pell-power :: nat ⇒ nat × nat ⇒ nat ⇒ nat × nat where
efficient-pell-power D z n = efficient-power (pell-mul-nat D) (1, 0) z n

lemma efficient-pell-power-correct [simp]:
efficient-pell-power D z n = (pell-mul-nat D z ^ n) (1, 0)
⟨proof⟩

```

2.13.3 Finding the fundamental solution

In the following, we set up a very simple algorithm for computing the fundamental solution (x, y) . We try increasing values for y until $1 + Dy^2$ is a perfect square, which we check using an efficient square-detection algorithm. This is efficient enough to work on some interesting small examples.

Much better algorithms (typically based on the continued fraction expansion of \sqrt{D}) are available, but they are also considerably more complicated.

```

lemma Discrete-sqrt-square-is-square:
  assumes is-square n
  shows floor-sqrt n ^ 2 = n
  ⟨proof⟩

definition find-fund-sol-step :: nat ⇒ nat × nat + nat × nat ⇒ - where
  find-fund-sol-step D = (λInl (y, y') ⇒
    (case get-nat-sqrt y' of
      Some x ⇒ Inr (x, y)
      None ⇒ Inl (y + 1, y' + D * (2 * y + 1)))))

definition find-fund-sol where
  find-fund-sol D =
    (if square-test D then
      (0, 0)
    else
      sum.projr (while sum.isl (find-fund-sol-step D) (Inl (1, 1 + D)))))

lemma fund-sol-code:
  assumes ¬is-square (D :: nat)
  shows pell.fund-sol D = sum.projr (while isl (find-fund-sol-step D) (Inl (Suc 0, Suc D)))
  ⟨proof⟩

lemma find-fund-sol-correct: find-fund-sol D = (if is-square D then (0, 0) else
pell.fund-sol D)
  ⟨proof⟩

```

2.13.4 The infinite list of all solutions

```

definition pell-solutions :: nat ⇒ (nat × nat) stream where
  pell-solutions D = (let z = find-fund-sol D in siterate (pell-mul-nat D z) (1, 0))

lemma (in pell) snth (pell-solutions D) n = nth-solution n
  ⟨proof⟩

```

2.13.5 Computing the n -th solution

```

definition find-nth-solution :: nat ⇒ nat ⇒ nat × nat where
  find-nth-solution D n =
    (if is-square D then (0, 0) else
      let z = sum.projr (while isl (find-fund-sol-step D) (Inl (Suc 0, Suc D)))
      in efficient-pell-power D z n)

lemma (in pell) find-nth-solution-correct: find-nth-solution D n = nth-solution n
  ⟨proof⟩

```

```
end
```

2.13.6 Tests

```
theory Pell-Algorithm-Test
imports
  Pell-Algorithm
  HOL-Library.Code-Target-Numerical
  HOL-Library.Code-Lazy
begin

  code-lazy-type stream

  value find-fund-sol 73
  value find-fund-sol 106

  value stake 100 (pell-solutions 73)
  value snth (pell-solutions 73) 600

  value find-nth-solution 73 600
  value find-nth-solution 106 10

end
```

References

- [1] Pell's equation, handout for MATHS 714. Lecture notes, University of Auckland, 2008.
- [2] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 2010.
- [3] M. Jacobson and H. Williams. *Solving the Pell Equation*. CMS Books in Mathematics. Springer New York, 2008.