NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
COMPUTER SCIENCE

MSc THESIS

Exact Geometric Predicates in Python

Maria N. Sotiropoulou

SUPERVISORS: **Ioannis Emiris,** Professor, Dept. Informatics & Telecoms, Univ. of Athens
Christodoulos Fragoudakis, Research Associate, Dept. Informatics & Telecoms, Univ. of Athens

ATHENS

March 2013

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΥΠΟΛΟΓΙΣΤΙΚΗ ΕΠΙΣΤΗΜΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεωμετρικά Κατηγορήματα Ακριβείας στην Python

Μαρία Ν. Σωτηροπούλου

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Ιωάννης Εμίρης,** Καθηγητής, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ
Χριστόδουλος Φραγκουδάκης, Επιστημονικός Συνεργάτης, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2013

MSc THESIS

Exact Geometric Predicates in Python

Maria N. Sotiropoulou
A.M.: M1093

SUPERVISORS: **Ioannis Emiris,** Professor, Dept. Informatics & Telecoms, Univ. of Athens
Christodoulos Fragoudakis, Research Associate, Dept. Informatics & Telecoms, Univ. of Athens

COMMITTEE OF INQUIRY: **Ioannis Emiris,** Professor, Dept. Informatics & Telecoms, Univ. of Athens
Dimitrios Gounopoulos, Professor, Dept. Informatics & Telecoms, Univ. of Athens

March 2013

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεωμετρικά Κατηγορήματα Ακριβείας στην Python

Μαρία Ν. Σωτηροπούλου
Α.Μ.: Μ1093

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Ιωάννης Εμίρης,** Καθηγητής, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ
Χριστόδουλος Φραγκουδάκης, Επιστημονικός Συνεργάτης, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Ιωάννης Εμίρης,** Καθηγητής, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ
Δημήτριος Γουνόπουλος, Καθηγητής, Τμήμα Πληροφορικής & Τηλ/νιών, ΕΚΠΑ

Μάρτιος 2013

ABSTRACT

The aim of this project is to establish a “geometric idiom” for Python that could replace and simultaneously enrich with the ability of execution, the pseudo code of the algorithms seen within every written document for Computational Geometry, as well as to support the development of an environment that will assist and supplement the undergraduate courses of Computational Geometry. It regards a computational system of boosting geometrical algorithms (Geometric algorithm aNimatiOn SYStem, GNOSYS) in two and three dimensions. The contribution of this final thesis to the GNOSYS system will be the development of a pure Python geometrical library that will replace the use of Python bindings with CGAL. An important reason for this replacement is the fact that the Python CGAL bindings project is not developing along with CGAL and the available fixed version is quite complex to be used in current systems. Another significant reason for the bindings to be replaced is that they carry the expressional inflexibilities of C++, and as a result, a program that uses them cannot be classified as pseudo code.

To begin with, a new numeric type is developed in order to supplement the floating point arithmetic. Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. For cases which require exact decimal representation, the decimal module is used, which implements decimal arithmetic suitable for accounting applications and high-precision applications. In order to simplify the procedure and avoid the need to manage precision details, this new numeric type is developed with regard to exact floating point arithmetic which contains the redefinition of all arithmetic operations (add, sub, mul, div, neg, sqrt etc). Within these operations, the context of precision is defined so as to prevent any change to the floats that are being processed.

Although a pure Python geometrical library gives the capability of the physical geometrical expression to the code, it misses the speed of execution obtained with a C++ library such as CGAL. This price seems bearable, since the target group will use the Python library mostly for educational purposes and not industrial ones (which is the main target of CGAL). However, there must be proof that the pure Python geometrical library gives at least as accurate results as CGAL. As a consequence, the available version of bindings with CGAL, is used as a means of comparison to pure Python geometrical library as far as correctness is concerned. Using the paper of Kettner et al. [1] as point of reference to the possible reasons that could cause implementations to fail, respective test cases, apart from the provided examples, have been produced so as to certify the correctness of the results.

SUBJECT AREA: Pure python Library

KEYWORDS: Python, Floating-Point, Decimal, precision, Computational Geometry

ΠΕΡΙΛΗΨΗ

Στόχος της διπλωματικής εργασίας είναι να εγκαθιδρύσει ένα “γεωμετρικό ιδίωμα” για την Python που θα μπορεί να αντικαταστήσει και ταυτόχρονα να εμπλουτίσει με τη δυνατότητα εκτέλεσης, τον ψευδοκώδικα των αλγορίθμων σε κάθε τυπικό σύγγραμμα Υπολογιστικής Γεωμετρίας, καθώς επίσης και η συμβολή στην ανάπτυξη ενός περιβάλλοντος που θα βοηθά και θα συμπληρώνει τη διδασκαλία του προπτυχιακού μαθήματος της Υπολογιστικής Γεωμετρίας. Πρόκειται για ένα υπολογιστικό σύστημα “εμψύχωσης” γεωμετρικών αλγορίθμων (Geometric algorithm aNimatiOn SYStem, GNOSYS) στις δύο και τρεις διαστάσεις. Η συμβολή της διπλωματικής εργασίας στο σύστημα GNOSYS θα είναι ο προγραμματισμός μιας “pure Python” γεωμετρικής βιβλιοθήκης που θα αντικαταστήσει τη χρήση των «δεσμεύσεων» (bindings) της Python με τη CGAL. Ένας σημαντικός λόγος για την αντικατάσταση είναι ότι οι Python CGAL bindings /δεσμεύσεις δεν αναπτύσσονται παράλληλα με τη CGAL και η διαθέσιμη σταθερή τους έκδοση είναι δύσκολο να χρησιμοποιηθεί σε σύγχρονα συστήματα. Ένας ακόμη σημαντικός λόγος για την αντικατάσταση των δεσμεύσεων είναι ότι μεταφέρουν εκφραστικές δυσκαμψίες της C++ και σαν αποτέλεσμα ένα πρόγραμμα που τις χρησιμοποιεί δεν μπορεί να χαρακτηριστεί και σαν ψευδοκώδικας.

Για αρχή, ένας νέος αριθμητικός τύπος αναπτύσσεται προκειμένου να συμπληρωθεί η αριθμητική δεκαδικού σημείου. Δυστυχώς, τα περισσότερα δεκαδικά κλάσματα δεν μπορούν να αναπαρασταθούν με δυαδικά κλάσματα. Για περιπτώσεις που απαιτούν ακριβή δεκαδική αναπαράσταση, χρησιμοποιείται η δομή decimal η οποία αναπαριστά δεκαδική αριθμητική, κατάλληλη για λογιστικές εφαρμογές κα εφαρμογές μεγάλης ακρίβειας. Προκειμένου να απλοποιηθεί η διαδικασία και να αποκρύψουμε τις λεπτομέρειες για τον καθορισμό της ακρίβειας, αναπτύσσεται αυτός ο νέος αριθμητικός τύπος, με ακριβή δεκαδική αριθμητική, ο οποίος περιλαμβάνει τον επαναπροσδιορισμό όλων των αριθμητικών πράξεων. Στον ορισμό των πράξεων περιλαμβάνεται ο καθορισμός της ακρίβειας ώστε να αποφεύγονται οι αλλαγές στους δεκαδικούς αριθμός κατά την επεξεργασία τους.

Μια “pure Python” γεωμετρική βιβλιοθήκη, ενώ δίνει τη δυνατότητα φυσικής γεωμετρικής έκφρασης στον κώδικα, υπολείπεται στην ταχύτητα εκτέλεσης μιας C++ βιβλιοθήκης σαν τη CGAL. Το κόστος αυτό φαίνεται λογικό να πληρωθεί αφού το κοινό που θα χρησιμοποιεί την Python βιβλιοθήκη βρίσκεται στην εκπαίδευση και όχι στη βιομηχανία (κύριος στόχος της CGAL). Πρέπει όμως να αποδειχθεί ότι η “pure Python” γεωμετρική βιβλιοθήκη είναι τουλάχιστο τόσο ορθή στα αποτελέσματα που παράγει, όσο και η CGAL. Για το λόγο αυτό, χρησιμοποιείται η διαθέσιμη έκδοση των δεσμεύσεων με τη CGAL σαν μέτρο σύγκρισης της ορθότητας της “pure Python” γεωμετρικής βιβλιοθήκης. Συνεπώς, κρίνεται αναγκαίο να αναπτυχθούν μελέτες περίπτωσης, όπου και οι δύο υλοποιήσεις θα δίνουν ταυτόσημα αποτελέσματα. Χρησιμοποιώντας το άρθρο των Kettner et al. [1] ως σημείο αναφοράς σχετικά με τους λόγους που θα μπορούσαν να προκαλέσουν την αποτυχία των εφαρμογών, έχουν αναπτυχθεί αντίστοιχες μελέτες περιπτώσεων, πέραν των παραδειγμάτων που παρέχονται, έτσι ώστε να πιστοποιηθεί η εγκυρότητα των αποτελεσμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αμιγώς Python βιβλιοθήκη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Python, Δεκαδική αναπαράσταση, Decimal, ακρίβεια, Υπολογιστική Γεωμετρία

Εισαγωγή

Η γλώσσα προγραμματισμού Python έχει ως κύριο χαρακτηριστικό της την αναγνωσιμότητα του κώδικα. Το συντακτικό της Python δίνει στους προγραμματιστές τη δυνατότητα να υλοποιούν αλγορίθμους σε πολύ λιγότερες γραμμές συγκριτικά με άλλες γλώσσες προγραμματισμού. Αποτελεί μια δυναμική γλώσσα η οποία υποστηρίζει πολλούς τύπους προγραμματισμού, όπως αντικειμενοστραφή, συναρτησιακό, διαδικαστικό και επίσης διαχειρίζεται αυτόματα την μνήμη. Ο συνδυασμός όλων αυτών των στοιχείων καθιστούν την Python σημαντικό κομμάτι της εκπαιδευτικής διαδικασίας μαθημάτων όπως η Υπολογιστική Γεωμετρία. Οι μαθητές έχουν την δυνατότητα να συγκεντρώνονται στα αλγοριθμικά σημεία του μαθήματος καθώς ο προγραμματισμός σε Python παραπέμπει ιδιαίτερα σε ψευδο-κώδικα.

Προκειμένου να ενισχυθεί η ευκολία στην έκφραση ακόμα και σε εκφυλισμένες περιπτώσεις ένας νέος αριθμητικός τύπος έχει αναπτυχθεί. Ο τύπος αυτός συμπληρώνει την αριθμητική των πραγματικών. Οι πραγματικοί αριθμοί αναπαριστώνται ως δυαδικά κλάσματα. Ωστόσο, υπάρχει ένα μεγάλο ποσοστό αυτών που δεν μπορούν να αναπαρασταθούν ακριβώς από κάποιο δυαδικό κλάσμα. Κατά συνέπεια, ένας τέτοιος αριθμός αναπαρίσταται κατά προσέγγιση από τους πραγματικούς που μπορούν να αποθηκευτούν στη μηχανή. Ένα σύνηθες παράδειγμα πραγματικού που παρουσιάζει πρόβλημα αναπαράστασης είναι ο αριθμός 0.1, ο οποίος δεν μπορεί να αναπαρασταθεί από δυαδικό κλάσμα. Αν η Python εκτύπωνε τον ακριβή αριθμό που είναι καταχωρημένος στον υπολογιστή για τον αριθμό 0.1, θα εμφάνιζε την τιμή:

0.1000000000000000055511151231257827021181583404541015625”.

Παρόλα αυτά, επειδή στην παρουσίαση των αποτελεσμάτων υπάρχει στρογγυλοποίηση στα πιο σημαντικά δεκαδικά ψηφία, σπάνια γίνεται αντιληπτό. Το πρόβλημα γίνεται πιο έντονο όταν οι τιμές αυτές χρησιμοποιούνται σε αριθμητικές πράξεις. Για παράδειγμα, η πράξη (0.1 + 0.2) = 0.30000000000000004 και σε μια σύγκριση ισότητας με την τιμή 0.3 δίνει ψευδή απάντηση.

Εκτός από τα ζητήματα στην αναπαράσταση των πραγματικών, υπάρχει και το σφάλμα της στρογγυλοποίησης λόγω των περιορισμών ως προς την ακρίβεια. Οι παραδοχές της IEEE 754 για τους αριθμούς διπλής ακρίβειας ξεκίνησαν να εφαρμόζονται το 1985 και πλέον χρησιμοποιούνται από τα περισσότερα συστήματα. Η IEEE 754 ορίζει ότι ένας αριθμός διπλής ακρίβειας διαθέτει ένα δυαδικό ψηφίο για την καταχώρηση του πρόσημου, έντεκα δυαδικά ψηφία για τον εκθέτη και 52 ψηφία για το δεκαδικό μέρος του πραγματικού. Η πραγματική τιμή που λαμβάνεται από τα 64 δυαδικά ψηφία προκύπτει από την ακόλουθη παράσταση

τιμή = $(-1)^{sign}(1.b_{-1}b_{-2}\ldots b_{-52})_2 \times 2^{(e-1023)}$.

Ο δυαδικός εκθέτης για τους πραγματικούς διπλής ακρίβειας χρησιμοποιεί μια αρχική ποσότητα για τον υπολογισμού του πραγματικού εκθέτη και η οποία ισούται με 1023. Η μορφή της τιμής του πραγματικού που αναγράφεται παραπάνω, περιλαμβάνει μια σταθερή μονάδα μπροστά από το δεκαδικό μέρος, εκτός και αν ο αποθηκευμένος εκθέτης είναι μηδέν. Συνεπώς, μαζί με τα 52 δυαδικά ψηφία όπου αποθηκεύεται το δεκαδικό μέρος, η συνολική ακρίβεια που παρέχεται είναι 53 δυαδικά ψηφία (περίπου 16 δεκαδικά ψηφία, 53 $\log_{10}(2) \approx 15.955$).

Ενδιάμεσα στις τιμές 2^{52}=4,503,599,627,370,496 και 2^{53}=9,007,199,254,740,992, οι αριθμοί που παρεμβάλλονται και μπορούν να αναπαρασταθούν με την αριθμητική διπλής ακρίβειας είναι μόνο οι ακέραιοι. Στο επόμενο εύρος 2^{53} έως 2^{54} όλοι οι αριθμοί διπλασιάζονται και στο προηγούμενο 2^{51} έως 2^{52} υποδιπλασιάζονται αντίστοιχα. Η

διαφορά ανάμεσα στους αριθμούς που αναπαρίστανται όταν αυτοί ανήκουν στο εύρος τιμών από 2^n έως 2^{n+1} είναι 2^{n-52}. Το μέγιστο σχετικό σφάλμα όταν γίνεται στρογγυλοποίηση ενός πραγματικού στην αντίστοιχη τιμή που μπορεί να αναπαρασταθεί είναι 2^{-53}.

Για τους λόγους που έχουν προαναφερθεί, όταν ο προγραμματιστής χρειάζεται ακριβείς υπολογισμούς, χωρίς εξαρτήσεις από την ακρίβεια που παρέχεται και σφάλματα στην αναπαράσταση των αριθμών, ενδείκνυται να χρησιμοποιήσουν δομές οι οποίες να υποστηρίζουν αριθμητική τυχαία μεγάλης ακρίβειας και σωστής αναπαράστασης των αριθμών.

Στην τρέχουσα διπλωματική εργασία παρουσιάζονται όλες οι σχετικές προσεγγίσεις για την διαχείριση του προβλήματος καθώς επίσης και ένας νέος αριθμητικός τύπος, ο MP_Float, ο οποίος έχει υλοποιηθεί αμιγώς με την χρήση της Python, χωρίς περεταίρω εξαρτήσεις. Ο νέος τύπος δεδομένων αντιμετωπίζει τα προβλήματα που προκύπτουν από την χρήση της αριθμητικής για πραγματικούς και παρέχει σωστούς και με ακρίβεια αριθμητικούς υπολογισμούς. Ο MP_Float βασίζεται στον αριθμητικό τύπο «Decimal» ο οποίος περιλαμβάνεται στην δομή «decimal» και παρέχεται από την Python. Εκτός από την ακριβή αναπαράσταση των πραγματικών που παρέχεται από τον τύπο Decimal, ο MP_Float αξιοποιεί την δυνατότητα του Decimal για αλλαγή της ακρίβειας των πράξεων ανάλογα με τις ανάγκες που έχει ο κάθε χρήστης. Συγκεκριμένα, επαναπροσδιορίζονται οι αριθμητικές πράξεις έτσι ώστε να περιλαμβάνουν την κατάλληλη ακρίβεια με την οποία δεν θα χρειάζεται να εφαρμοστεί στρογγυλοποίηση. Στη συνέχεια ακολουθεί η περιγραφή του νέου αυτού αριθμητικού τύπου. Επιπλέον, γίνεται μια εκτενής αναφορά στις επιπτώσεις της χρήσης της αριθμητικής πραγματικών στο κατηγόρημα κατεύθυνσης και στον αυξητικό αλγόριθμο εύρεσης του κυρτού περιβλήματος ενός συνόλου σημείων. Οι περιπτώσεις που παρουσιάζονται στο άρθρο [1] αναπαράγονται με την χρήση αμιγώς της Python ως γλώσσα προγραμματισμού. Παράλληλα, τα αντίστοιχα αποτελέσματα μετά την χρήση του MP_Float τύπου και των κατηγορημάτων της Python/ CGAL παρατίθενται επίσης.

Άλλες Προσεγγίσεις

Υπάρχουν διάφορες προσεγγίσεις οι οποίες διαβεβαιώνουν αξιόπιστους γεωμετρικούς υπολογισμούς. Αρχικά, υπάρχει η προσέγγιση όπου χρησιμοποιείται το πρότυπο γεωμετρικών υπολογισμών ακριβείας (EGC), με χαρακτηριστικά παραδείγματα την CGAL [7], [8], την LEDA [9], [10] και την βιβλιοθήκη Core [11]. Παρόλο που αυτές οι βιβλιοθήκες χρησιμοποιούν αριθμητική ακριβείας, δεν υπάρχει διαθέσιμη δομή η οποία να μπορεί να χρησιμοποιηθεί άμεσα από την Python.

Μία δεύτερη προσέγγιση είναι η εφαρμογή κάποιας διαταραχής στα δεδομένα εισόδου, προκειμένου να εξασφαλιστεί ότι η υλοποίηση που χρησιμοποιεί την αριθμητική πραγματικών θα παράγει τα σωστά αποτελέσματα με βάση τη διαταραγμένη είσοδο [12], [13], [14]. Η προσέγγιση αυτή απαιτεί διεξοδική ανάλυση του προβλήματος και ακριβή προσδιορισμό της εισόδου και της εξόδου. Όσον αφορά στα δεδομένα εισόδου, όσο πιο πολύπλοκα είναι τα αντικείμενα τόσο πιο δύσκολο είναι να καθοριστεί η διαταραχή στην είσοδο. Αντίστοιχα, ο προσδιορισμός της εξόδου είναι επίσης εξαιρετικά δύσκολος, αφού είναι απαραίτητο να ληφθούν υπόψη όλες οι πιθανές εκβάσεις ακόμα και για εκφυλισμένες περιπτώσεις. Επιπρόσθετα, όταν πρόκειται να χρησιμοποιηθεί αριθμητική πραγματικών, μόνο ορισμένοι γεωμετρικοί αλγόριθμοι είναι γνωστοί οι οποίοι συμπεριφέρονται φαινομενικά σωστά [15], [16], [17], [18].

Ο τύπος MP_Float

Στην παρούσα διπλωματική εργασία έχει αναπτυχθεί ο νέος αριθμητικός τύπος MP_Float με αποκλειστική χρήση της Python. Ο τύπος αυτός παρέχει στον χρήστη τη δυνατότητα να χρησιμοποιεί αριθμητική πραγματικών με απροσδιόριστο εύρος για την ακρίβεια, η οποία υπόκειται μόνο στους περιορισμούς χωρητικότητας του υπολογιστή στον οποίο πραγματοποιείται η εργασία και όχι σε περιορισμούς για δεδομένη τιμή ακρίβειας. Με τα τρέχοντα δεδομένα, η Python χρησιμοποιεί αριθμητική διπλή ακρίβειας όπως αυτή ορίζεται από την IEEE 754. Όπως έχει προαναφερθεί, οι αριθμοί διπλής ακρίβειας σύμφωνα με την IEEE 754 διαθέτουν 53 δυαδικά ψηφία για το δεκαδικό μέρος του αριθμού. Κατά προσέγγιση, η δεκαδική αναπαράσταση του αριθμού έχει μέχρι 16 ($53\log_{10}(2) \approx 15.955$) δεκαδικά ψηφία ακρίβεια. Τα ψηφία αυτά περιλαμβάνουν τόσο το δεκαδικό μέρος όσο και το ακέραιο μέρος του πραγματικού αριθμού.

Η Python ήδη προσφέρει στους χρήστες τη δομή «decimal» για την υποστήριξη της αριθμητικής πραγματικών. Οι πραγματικοί μπορούν να αναπαρασταθούν ακριβώς μέσα από τον αριθμητικό τύπο «Decimal» που παρέχεται και, αντίθετα από την καθιερωμένη αριθμητική πραγματικών του συστήματος, δίνει στον χρήστη τη δυνατότητα να μεταβάλει την ακρίβεια σύμφωνα με τις απαιτήσεις του εκάστοτε προβλήματος. Εντούτοις, στην προκειμένη περίπτωση, το πρόβλημα που καλείται ο χρήστης να λύσει είναι η στρογγυλοποίηση των αριθμών η οποία θα μπορούσε να προκαλέσει υπολογιστικά σφάλματα. Τέτοια σφάλματα μπορούν να προκύψουν κατά τη διάρκεια οποιουδήποτε υπολογισμού εξαιτίας της προκαθορισμένης ακρίβειας. Για να διευκολυνθεί ο σωστός υπολογισμός των αριθμητικών πράξεων, χωρίς να ασχοληθεί ο χρήστης με θέματα ακρίβειας, κρίθηκε απαραίτητος ο ορισμός του νέου αριθμητικού τύπου και των αντίστοιχων αριθμητικών πράξεων. Η επέκταση αυτή θα μπορούσε να ενισχύσει την Python έτσι ώστε η υλοποίηση να προσεγγίζει επίπεδο ψευδο- κώδικα.

Για το όνομα MP_Float (Multi-Precision Float) γίνεται επίσης αναφορά στην CGAL. Ωστόσο, ο τρέχον τύπος, πέραν του ονόματος, δεν σχετίζεται σε κάτι με κωδικοποίηση σε CGAL. Το ζητούμενο εδώ είναι να τονιστεί ότι οι πραγματικοί αριθμοί με οιαδήποτε ακρίβεια μπορούν να αναπαρασταθούν ακριβώς. Ταυτόχρονα, κάθε αριθμητική διαδικασία δίνει τα σωστά και αναμενόμενα αποτελέσματα.

Περιορισμοί του «Duck typing»

Ο αριθμητικός τύπος MP_Float έχει ένα μόνο όρισμα, τον αριθμό. Ο δεκαδικός αριθμός που δίνεται στον MP_Float κατά τη δημιουργία του, μετατρέπεται σε «Decimal» προκειμένου να αξιοποιηθούν τα πλεονεκτήματα της δομής decimal. Παρά ταύτα, η Python χρησιμοποιεί ευρέως τον προγραμματιστικό τρόπο του «Duck typing». Σύμφωνα με το γλωσσάρι της Python ο όρος «Duck typing» ορίζεται ως η προγραμματιστική τεχνοτροπία της Python η οποία καθορίζει τον τύπο ενός αντικειμένου ελέγχοντας την μέθοδο ή την υπογραφή του ορίσματος αντί για την αποκλειστική σχέση με κάποιο τύπο δεδομένων («Αν μοιάζει με πάπια και κρώζει σαν πάπια, τότε πρέπει να είναι πάπια»). Δίνοντας έμφαση στην διεπαφή αντί για τους συγκεκριμένους τύπους, ένας καλοσχεδιασμένος κώδικας βελτιώνει την ευελιξία επιτρέποντας πολυμορφική αντικατάσταση. Το «duck typing» αποφεύγει την χρήση των συναρτήσεων type() ή isinstance(). Αντιθέτως, απλά εφαρμόζει την προγραμματιστική τακτική «Ευκολότερο να ζητάς συγχώρεση παρά άδεια».

Για τον λόγο αυτό, ο τύπος της τιμής που θα δοθεί ως όρισμα σε μια συνάρτηση, δεν μπορεί να έχει προκαθοριστεί. Όσον αφορά στον MP_Float τύπο, όταν ένας πραγματικός αριθμός με μεγάλη δεκαδική αναπαράσταση περνιέται ως παράμετρος χωρίς εισαγωγικά σημεία, ο κώδικας που επεξεργάζεται την παράμετρο θα κάνει συγκεκριμένες υποθέσεις για τον τύπο της. Κατ' επέκταση, ο πραγματικός θεωρείται ότι

είναι τύπου float αυτόματα. Εφόσον, όμως, η τρέχουσα ακρίβεια είναι μικρότερη από τις απαιτήσεις του πραγματικού υπό συζήτηση, γίνεται στρογγυλοποίηση στον πλησιέστερο float αριθμό. Για να αποφύγουμε αυτό το γεγονός, οι αριθμοί που δίνονται ως όρισμα στον MP_Float πρέπει να εσωκλείονται σε εισαγωγικά σημεία. Στην περίπτωση που ο αριθμός δεν δοθεί ως συμβολοσειρά, ο τύπος το δέχεται και τυπώνει μια προειδοποίηση για πιθανή στρογγυλοποίηση.

Ανάπτυξη συναρτήσεων και πράξεων του MP_Float

Για την διευκόλυνση της λειτουργικότητας του κώδικα, έχουν αναπτυχθεί ορισμένες βοηθητικές συναρτήσεις. Τέτοιες αποτελούν το μήκος, σε αριθμό δεκαδικών ψηφίων, του αριθμού, του ακέραιου μέρους και του δεκαδικού μέρους, καθώς επίσης και η εύρεση των ίδιων τμημάτων.

Στην προσπάθεια να αποφευχθούν οποιαδήποτε σφάλματα στρογγυλοποίησης και αναπαράστασης για κάθε πράξη μεταξύ δυο αριθμών, πρέπει να προκαθοριστεί η απαραίτητη ακρίβεια. Στην Υπολογιστική Γεωμετρία, οι πιο συχνά χρησιμοποιούμενες πράξεις είναι αυτές της πρόσθεσης, της αφαίρεσης και του πολλαπλασιασμού. Για τις συγκεκριμένες πράξεις, στην ακρίβεια μπορεί να αποδοθεί μια ελάχιστη και ασφαλής τιμή η οποία θα εγγυάται την απουσία στρογγυλοποιήσεων κατά τη διάρκεια και μετά το πέρας των αριθμητικών πράξεων. Συνεπώς, το αποτέλεσμα θα είναι το αναμενόμενο.

Όσον αφορά στην πρόσθεση, ο τρόπος με τον οποίο εκτελείται η πράξη μεταξύ δυο πραγματικών μας βοηθά να καθορίσουμε κατάλληλα την ακρίβεια. Αφού τοποθετηθούν οι αριθμοί ο ένας κάτω από τον άλλον, έτσι ώστε οι υποδιαστολές να βρίσκονται σε συστοιχία, ακολουθεί η πρόσθεση των επιμέρους ψηφίων. το δεκαδικό μέρος του νέου αριθμού έχει τόσα δεκαδικά ψηφία όσα και το μεγαλύτερο δεκαδικό μέρος των υπό επεξεργασία αριθμών. Αντίστοιχα το ακέραιο μέρος του αποτελέσματος θα έχει τόσα ψηφία όσο και το μεγαλύτερο ακέραιο μέρος στην πρόσθεση και πιθανόν και ένα ακόμη ψηφίο, το κρατούμενο. Για να αποφευχθεί η αναζήτηση των απαραίτητων μεγίστων, η τιμή της ακρίβειας έχει καθοριστεί να ισούται με το άθροισμα των μηκών των δυο πραγματικών που προστίθενται. Γνωρίζοντας ότι το ελάχιστο μήκος πραγματικού που μπορεί να δοθεί είναι δυο ψηφία (για παράδειγμα: 0.0), οι απαιτήσεις σε ακρίβεια υπέρ-καλύπτονται.

Η ακρίβεια για την αφαίρεση υπολογίζεται με αντίστοιχο τρόπο και λαμβάνει την ίδια τιμή όπως και στην πρόσθεση. Εξάλλου αν η αφαίρεση αφορά σε πραγματικούς αντιθέτου πρόσημου, τότε η πράξη μετατρέπεται σε πρόσθεση.

Η διαδικασία του πολλαπλασιασμού στους δεκαδικούς αριθμούς είναι η ίδια όπως και με τους ακέραιους, με μόνη διαφορά ότι στο αποτέλεσμα πρέπει η υποδιαστολή να τοποθετηθεί στην σωστή θέση. Στον πολλαπλασιασμό, γίνεται αρχικά πολλαπλασιασμός του κάθε ψηφίου του δεύτερου αριθμού με όλα τα ψηφία του πρώτου. Για κάθε ψηφίο του δεύτερου υπάρχει μια μετακίνηση αριστερά του εκάστοτε αποτελέσματος. Τα κενά που μένουν στο τέλος γεμίζουν με μηδενικά και έπειτα πραγματοποιείται πρόσθεση των επιμέρους γινομένων. Το μεγαλύτερο σε μήκος γινόμενο θα είναι το τελευταίο στη σειρά, το οποίο λόγω των μετατοπίσεων θα έχει τόσα μηδενικά όσες οι μετακινήσεις (δηλαδή το μήκος του δεύτερου αριθμού μείον ένα, αφού δεν έγινε μετατόπιση του πρώτου γινομένου), και επιπλέον δεκαδικά ψηφία το πολύ όσα τα ψηφία του πρώτου αριθμού συν ένα. Η προσθήκη του επιπλέον ενός ψηφίου μπορεί να προκύψει λόγω κρατούμενου. Επειδή τα πιο σημαντικά ψηφία αριστερά που μπορεί να προστεθούν είναι το πολύ δύο, λόγω και των μετατοπίσεων, το κρατούμενο δεν μπορεί να είναι πάνω από ένα δεκαδικό ψηφίο.

Εκτός από αυτές τις πράξεις, έχουν οριστεί κάποιες επιπλέον μέθοδοι. Αρχικά η μέθοδος της σύγκρισης λαμβάνει ως ακρίβεια το μέγεθος του μέγιστου από τους

αριθμούς που συγκρίνονται. Για τις υπόλοιπες, όπως οι sqrt, div, pow, mod, divmod και floordiv, το αποτέλεσμα θα μπορούσε να απαιτεί άπειρα δεκαδικά ψηφία, οπότε η ακρίβεια έχει καθοριστεί τυχαία στο μέγιστο μήκος των πραγματικών που εμπλέκονται. Πέραν αυτών των αριθμητικών πράξεων έχουν οριστεί επιπλέον αντίστοιχες συναρτήσεις (div, power, mod, divmodule, floordiv) οι οποίες δίνουν την δυνατότητα στον χρήστη να καθορίσει ο ίδιος την ακρίβεια. Τέλος θεωρήθηκε χρήσιμο να οριστούν και οι αντίστοιχες «reverse» και «in-place» πράξεις.

Προβλήματα Ευρωστίας σε Γεωμετρικούς Υπολογισμούς

Κατηγόρημα προσανατολισμού στο επίπεδο

Πολλές εφαρμογές της Υπολογιστικής Γεωμετρίας χρησιμοποιούν αριθμητικούς ελέγχους οι οποίοι είναι γνωστοί ως έλεγχοι προσανατολισμού. Οι έλεγχοι αυτοί καθορίζουν πότε ένα σημείο βρίσκεται αριστερά, δεξιά ή επάνω στην ευθεία που ορίζεται από δυο άλλα σημεία. Έστω ότι τα σημεία που ορίζουν την ευθεία είναι τα $p=(p_x, p_y)$, $q=(q_x, q_y)$ και ότι το υπό συζήτηση σημείο είναι το $r=(r_x, r_y)$. Ο προσανατολισμός της τριπλέτας (p, q, r) ισοδυναμεί με το σήμα της εξής ορίζουσας:

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}\right)$$

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right).$$

Συγκεκριμένα, αριστερόστροφος προσανατολισμός υποδηλώνεται όταν το orientation είναι θετικό (+1), δεξιόστροφος όταν είναι αρνητικό (-1) και ότι τα σημεία είναι συνευθειακά όταν το orientation ισούται με μηδέν. Η ορίζουσα εκφράζεται ως προς τις συντεταγμένες των σημείων. Αν οι συντεταγμένες αυτές εκφράζονται ως πραγματικοί αριθμοί με σύστημα μονής ή διπλής ακρίβειας, τότε κάποιο σφάλμα αναπαράστασης ή στρογγυλοποίησης μπορεί να προκύψει όταν η πραγματική τιμή της ορίζουσας προσεγγίζει το μηδέν. Έστω ότι ονομάζουμε αυτό τον προσανατολισμό float_orient(p, q, r) για να το διαχωρίσουμε από το ιδανικό κατηγόρημα. Υπάρχουν τρεις πιθανοί τρόποι με τους οποίους να διαφέρουν τα δυο κατηγορήματα.

Στρογγυλοποίηση στο μηδέν: λήψη του + ή του – ως 0,

Διαταραχή στο μηδέν: λήψη του 0 ως + ή - ,

Αντιστροφή πρόσημου: λήψη του + ως – και αντίστροφα.

Γεωμετρία του δεκαδικού προσανατολισμού

Σύμφωνα με τους Kettner et al. [1], το ακόλουθο πείραμα μπορεί να καθορίσει την γεωμετρία του float_orient. Τρία σημεία p, q και r επιλέγονται και στη συνέχεια υπολογίζεται ο ακόλουθος προσανατολισμός:

$$\text{float\_orient}((p_x + xu, p_y + yu), q, r)$$

για $0 \le x, y \le 255$, όπου u είναι η διαφορά μεταξύ δυο συνεχόμενων πραγματικών αριθμών όπως αυτοί αναπαρίστανται ως float, ανάλογα με το εύρος τιμών στο οποίο ανήκουν. Από το πείραμα αυτό προκύπτει ένας πίνακας πρόσημων διαστάσεων 256x256, το οποίο οπτικοποιείται σε ένα πλέγμα με χρωματιστά εικονοστοιχεία. Ένα κίτρινο (κόκκινο, μπλε) εικονοστοιχείο αντιπροσωπεύει συνευθειακό (αρνητικό, θετικό αντίστοιχα) προσανατολισμό.

Ο τρόπος με τον οποίο οι Kettner et al. [1] υπολογίσουν το σημείο (p_x + xu, p_y + yu) μέσα στον πηγαίο κώδικα που παρέχουν, χρησιμοποιεί την C++ και βασίζεται σε λογικές πράξεις. Συγκεκριμένα, λαμβάνονται υπόψη και τα 64 δυαδικά ψηφία του δεκαδικού και ο τρόπος με τον οποίο έχουν κατανεμηθεί προκειμένου να πραγματοποιηθούν οι σωστές λογικές προσθέσεις και μετατοπίσεις. Με αυτό τον τρόπο υπολογίζουν την ελάχιστη τιμή του u η οποία πρέπει να προστεθεί στην τιμή των συντεταγμένων σημείου ώστε να αξιοποιηθεί η διαφορά μεταξύ διαδοχικών δεκαδικών τιμών.

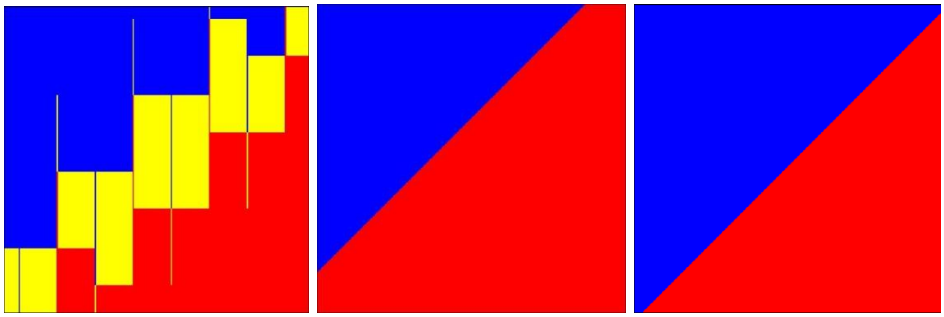
Στο παρόν κείμενο, το ίδιο πείραμα έχει διεξαχθεί χρησιμοποιώντας Python ως γλώσσα προγραμματισμού. Οι συναρτήσεις που έχουν αναπτυχθεί παράγουν την γεωμετρία του προσανατολισμού και για την τυπική αριθμητική πραγματικών και για χρήση του MP_Float. Αυτές οι συναρτήσεις βασίζονται τόσο στις οδηγίες των Kettner et al. όσο και στη διαφορά των αριθμών που προκύπτει από τις προδιαγραφές του IEEE 754. Ο κύριος αλγόριθμος πραγματοποιεί το πείραμα προσθέτοντας μια διαταραχή στις συντεταγμένες του πρώτου σημείου, και στη συνέχεια υπολογίζοντας τον ζητούμενο προσανατολισμό. Η διαταραχή u, είναι στην πραγματικότητα η διαφορά 2^{n-52}, όταν η τιμή της συντεταγμένης που επεξεργάζεται ο αλγόριθμος ανήκει στο εύρος τιμών από 2^n έως 2^{n+1}. Τα δεδομένα που απορρέουν από τις συναρτήσεις αυτές, περνούν για περεταίρω επεξεργασία και προβολή σε εικόνα από πρόγραμμα της OpenGl. Ο κώδικας αυτός συμπεριλαμβάνεται στο παράρτημα B (Appendix B).

Υπάρχουν πολλά παραδείγματα όπου η γεωμετρία του float_orient δεν είναι η αναμενόμενη. Ένα χαρακτηριστικό παράδειγμα αποτελεί η τριπλέτα σημείων:

p = (0.50000000000002531, 0.5000000000000171)

q = (17.300000000000001, 17.300000000000001)

r = (24.00000000000005, 24.0000000000000517765)



Εικόνα 1

Η πρώτη εικόνα που παρουσιάζεται είναι το αποτέλεσμα που δίνουν οι Kettner et al.και είναι η ίδια που λαμβάνουμε με χρήση Python. Η δεύτερη εικόνα παρουσιάζει το αποτέλεσμα του ίδιου πειράματος κάνοντας όμως χρήση του τύπου MP_Float στον ίδια υλοποίηση Python. Στην τελευταία εικόνα το κατηγόρημα προσανατολισμού που χρησιμοποιείται είναι το κατηγόρημα προσανατολισμού που παρέχει η CGAL. Όλες οι εικόνες αναφέρονται στις ίδιες τιμές συντεταγμένων. Τα χρώματα κίτρινο (κόκκινο, μπλε) αντιστοιχούν σε συνευθειακό (αρνητικό, θετικό αντίστοιχα) προσανατολισμό.

Οι συντεταγμένες των σημείων p και q αναπαρίστανται ακριβώς από όλες τις υλοποιήσεις. Το τρίτο σημείο (r), έχει τετμημένη με 19 ψηφία στο δεκαδικό της μέρος. Συνεπώς, ο συνήθης δεκαδικός τύπος (float) δεν μπορεί να δώσει τη σωστή αναπαράσταση της συντεταγμένης και το γεγονός αυτό φέρει αντίστοιχες επιπτώσεις στο αποτέλεσμα του κατηγορήματος προσανατολισμού. Ο MP_Float τύπος αναπαριστά

ακριβώς και τη συγκεκριμένη συντεταγμένη και η γεωμετρία που λαμβάνεται δίνει τη σωστή εικόνα. Όσον αφορά στην CGAL, εξαιτίας του φαινομένου «duck-typing», οι συντεταγμένες που δίνονται ως όρισμα στη CGAL για να δημιουργηθεί το αντίστοιχο σημείο, θεωρούνται απλοί δεκαδικοί (float) από την Python και ως CGAL::Simple_cartesian τιμές από τη CGAL. Ο δεύτερος τύπος χρησιμοποιεί αριθμητική πραγματικών όπως και η κλασσική Python. Κατά συνέπεια, η CGAL βλέπει την λάθος αναπαράσταση της r.y συντεταγμένης. Ωστόσο, κατά τη διάρκεια του υπολογισμού του κατηγορήματος, οι αριθμοί μετατρέπονται σε CGAL::Lazy_exact_nt, ο οποίος είναι ένας ειδικός τύπος για υποστήριξη υπολογισμών ακριβείας. Αυτός είναι και ο λόγος που η εικόνα που λαμβάνεται από την CGAL φαίνεται σωστός. Στην πραγματικότητα, είναι ακριβής για τα ανακριβή δεδομένα που έχει λάβει ως είσοδο.

Πρόβλημα κυρτού περιβλήματος στο επίπεδο

Ένα σύνολο καλείται κυρτό όταν για κάθε δυο σημεία p και q, ολόκληρο το ευθύγραμμο τμήμα pq περιλαμβάνεται μέσα στο σύνολο. Το κυρτό περίβλημα CH ενός συνόλου σημείων S είναι το μικρότερο (όσον αφορά στο σύνολο αυτό) κυρτό σύνολο που περικλείει το S.

Αυξητικός αλγόριθμος κυρτού περιβλήματος στο επίπεδο

Ο αυξητικός αλγόριθμος διατηρεί το τρέχον κυρτό περίβλημα CH των σημείων που έχει ελέγξει. Αρχικά το κυρτό περίβλημα απαρτίζεται από τρία μη συνευθειακά σημεία και έπειτα ελέγχει ένα-ένα τα υπόλοιπα σημεία. Όταν επεξεργάζεται ένα σημείο, πρώτα ελέγχει αν αυτό βρίσκεται εκτός του τρέχοντος CH, και εάν ισχύει, ενημερώνεται το CH αλλιώς αφαιρείται και ο αλγόριθμος συνεχίζει στο επόμενο σημείο. Το τρέχον κυρτό περίβλημα αποθηκεύεται σε μια κυκλική λίστα $L=(v_0, v_1, \ldots, v_{k-1})$ με φορά αντίθετη από αυτή των δεικτών του ρολογιού. Τα ευθύγραμμα τμήματα (v_i, v_{i+1}) αποτελούν τις ακμές του κυρτού περιβλήματος. Αν orientation$(v_i,v_{i+1},r) < 0$, τότε το r βλέπει την ακμή. Αν orientation$(v_i,v_{i+1},r) \leq 0$, τότε η ακμή (v_i, v_{i+1}) είναι ασθενώς ορατή από το r. Οι ακόλουθες ιδιότητες αποτελούν την βάση για την λειτουργία του αλγορίθμου.

Ιδιότητα Α: ένα σημείο r είναι εκτός του CH ανν το r μπορεί να δει μια πλευρά του CH.

Ιδιότητα Β: Αν το r είναι εκτός του CH, οι ακμές που είναι ασθενώς ορατές από το r σχηματίζουν μια μη κενή διαδοχική αλυσίδα, και ομοίως ισχύει για τις ακμές που δεν είναι ασθενώς ορατές.

Υπάρχουν τρεις δυνατοί τρόποι για να αναιρεθούν οι ιδιότητες Α και Β.

- Αποτυχία Α1: Ένα σημείο εκτός του τρέχοντος CH δεν βλέπει καμία ακμή του CH.
- Αποτυχία Α2: Ένα σημείο εντός του τρέχοντος CH βλέπει κάποια ακμή του CH.
- Αποτυχία Β1: Ένα σημείο εκτός του τρέχοντος CH βλέπει όλες τις ακμές του CH.
- Αποτυχία Β2: Ένα σημείο εκτός του τρέχοντος CH βλέπει ένα μη διαδοχικό σύνολο ακμών.

Ευκρινή αρνητικά αποτελέσματα των αποτυχιών

Η ανάλυση των παραπάνω αποτυχιών παρουσιάζει την επίδραση ενός λανθασμένου ελέγχου προσανατολισμού στον αυξητικό αλγόριθμο. Υπάρχουν παραδείγματα όπου το σφάλμα δεν είναι αμελητέο, αλλά αντίθετα ιδιαίτερα εμφανές. Τα αποτελέσματα που λαμβάνονται περιλαμβάνουν διάφορες εκδοχές. Ο αλγόριθμος μπορεί να υπολογίζει ένα κυρτό πολύγονο αλλά να μην συμπεριλαμβάνει κάποιο ακραίο σημείο, όπως συμβαίνει με την αποτυχία Α1. Επιπλέον, ο αλγόριθμος μπορεί να μην τερματίζει ποτέ στην περίπτωση της αποτυχίας Α2 ή να τερματίζει απότομα χωρίς να παρέχει αποτέλεσμα.

Τέλος, στην περίπτωση που συμβεί η αποτυχία B2, για παράδειγμα, το αποτέλεσμα που προκύπτει πιθανόν να είναι ένα μη κυρτό πολύγωνο.

Πρέπει να σημειωθεί ότι κατά την αναπαραγωγή των παραδειγμάτων του [1] από τον πηγαία κώδικα που παρέχεται, κρίθηκε απαραίτητη η εγκατάσταση κάποιων εξωτερικών βιβλιοθηκών πέραν της CGAL, όπως είναι η LEDA(Library of Efficient Data types and Algorithms) και η GMP (Gnu Multiple Precision). Η διαδικασία αποδείχτηκε εξαιρετικά δύσκολη. Γι αυτό το λόγο, αναπτύχθηκε μια εικονική μηχανή, την οποία και συμπεριλαμβάνουμε στην παρούσα εργασία για οποιονδήποτε επιθυμεί να πειραματιστεί.

Μελλοντικές Επεκτάσεις

Το περιεχόμενο αυτής της διπλωματικής μπορεί να χρησιμοποιηθεί για την υποστήριξη της ανάπτυξης ενός περιβάλλοντος το οποίο θα ενισχύσει τα προπτυχιακά μαθήματα της Υπολογιστικής Γεωμετρίας. Αφορά σε ένα υπολογιστικό σύστημα ενίσχυσης των γεωμετρικών αλγορίθμων (Geometric algorithm aNimatiOn SYStem, GNOSYS) σε δύο και τρεις διαστάσεις. Ο κύριος στόχος είναι η ανάπτυξη ενός εργασλείου το οποίο θα βοηθήσει τους μαθητές να παρακολουθούν οπτικά το αντικείμενο του μαθήματος καθώς αυτό εξελίσσεται. Το σύστημα παρέχει γεωμετρικές βιβλιοθήκες και βιβλιοθήκες οπτικοποίησης οι οποίες εξυπηρετούν την γρήγορη δημιουργία διαδραστικών οπτικοποιήσεων των αλγορίθμων στην Υπολογιστική Γεωμετρία. Η συμβολή της τρέχουσας διπλωματικής εργασίας στο σύστημα GNOSYS είναι η ανάπτυξη μιας «pure Python» γεωμετρικής βιβλιοθήκης η οποία θα αντικαταστήσει την χρήση των Python δεσμεύσεων με τη CGAL, το οποίο είναι ένα έργο προς ολοκλήρωση.

Επιπρόσθετα, θα ήταν ενδιαφέρον αν δινόταν η δυνατότητα από την Pyhton, τουλάχιστον όταν πρόκειται για θέματα Υπολογιστικής Γεωμετρίας, η μέθοδος του «duck-typing» να αναγνωρίζει τους πραγματικούς αριθμούς και να εφαρμόζει αριθμητική ακριβείας αντί για την συμβατική αριθμητική διπλής ακρίβειας που χρησιμοποιεί την τρέχουσα στιγμή.

Συμπεράσματα

Εν κατακλείδι, υπάρχουν διάφορες περιπτώσεις όπου η αριθμητική των πραγματικών αποτυγχάνει και ο αντίστοιχος αλγόριθμος παράγει απρόβλεπτα αποτελέσματα. Αντιθέτως, οι υλοποιήσεις που χρησιμοποιούν τον MP_Float αντί για τον κλασικό δεκαδικό τύπο, παρουσιάζουν τον σωστό και ακριβή τρόπο με τον οποίο πρέπει να εξελίσσονται οι υπολογισμοί. Η ακρίβεια που χαρακτηρίζει τν αριθμητικό τύπο του MP_Float, έχει κληρονομηθεί από τη δομή «decimal» η οποία εσωκλείεται στον τύπο του MP_Float. Επιπρόσθετα, ο νέος αυτός τύπος αξιοποιεί την δυνατότητα που παρέχεται από την «decimal» με την οποία ο χρήστης μπορεί να προκαθορίσει την ακρίβεια σύμφωνα με τις εκάστοτε απαιτήσεις, έτσι ώστε να αποφευχθούν σφάλματα στρογγυλοποίησης. Συγκεκριμένα, ο χρήστης δεν χρειάζεται να ασχοληθεί με θέματα ακρίβειας επειδή ο MP_Float επαναπροσδιορίζει τις αριθμητικές πράξεις και τους αποδίδει την κατάλληλη ακρίβεια. Μετά την αναπαραγωγή των παραδειγμάτων των Kettner et al.[1] με την χρήση της Python, τα αποτελέσματα είναι ενδεικτικά για κάθε πιθανή αποτυχία που μπορεί να προκύψει λόγω της αριθμητικής των πραγματικών, όσον αφορά στο κατηγόρημα του προσανατολισμού και τα αποτελέσματα του αυξητικού αλγορίθμου για την εύρεση κυρτού περιβλήματος ενός συνόλου σημείων. Η αριθμητική πραγματικών παρουσιάζει μη αναμενόμενα αποτελέσματα όταν τα σημεία που εξετάζονται είναι σχεδόν συνευθειακά. Ο MP_Float τύπος, αντίθετα, παράγει την αναμενόμενη έξοδο, καθώς οι υπολογισμοί είναι ακριβείς, χωρίς σφάλματα αναπαράστασης ή στρογγυλοποίησης εξαιτίας περιορισμών στην ακρίβεια. Το αντίστοιχο κατηγόρημα προσανατολισμού που παρέχεται από τη CGAL, επίσης

αποδίδει τα σωστά αποτελέσματα. Παρόλα αυτά, λόγω του «duck-typing», που αποτελεί χαρακτηριστικό της Python, οι συντεταγμένες αναγνωρίζονται αυτόματα από την Python ως δεκαδικοί και συνεπώς μπορεί να υποστούν κάποιο σφάλμα αναπαράστασης.

Ο αριθμητικός τύπος MP_Float δεν έχει περαιτέρω απαιτήσεις από το σύστημα, εκτός από την διεπαφή με την python. Συνεπώς, μπορεί εύκολα να αντικαταστήσει τον αριθμητικό τύπο float όταν πρέπει να διεξαχθούν υπολογισμοί με πεπερασμένα τυχαίο μέγεθος ακρίβειας. Ωστόσο, επειδή οι υπολογισμοί υπολείπονται σε ταχύτητα, ο συγκεκριμένος τύπος ενδείκνυται περισσότερο για χρήση κατά την εκπαιδευτική διαδικασία. Ειδικά, θα μπορούσε να υποστηρίζει τα μαθήματα της Υπολογιστικής Γεωμετρίας με την δυνατότητα της επαγωγής οποιουδήποτε αλγορίθμου της Υπολογιστικής Γεωμετρίας σε ένα εκτελέσιμο πρόγραμμα, ακόμη και για εκφυλισμένες περιπτώσεις. Η υλοποίηση του MP_Float είναι απλή αλλά απαιτεί τετραγωνική πολυπλοκότητα για τον πολλαπλασιασμό. Το γεγονός αυτό μπορεί να αποτελεί πρόβλημα για μεγάλους πολλαπλασιαστές. Για ταχύτερες υλοποιήσεις αλλά με την ίδια όμως λειτουργικότητα στις μεγάλες τιμές δεκαδικών, ο χρήστης ίσως πρέπει να επανεξετάσει τη χρήση κάποιας από τις βιβλιοθήκες GMP ή LEDA αντίστοιχα.

INDEX

FIGURE INDEX

PROLOGUE

This project is conducted in the terms of MSc thesis of the postgraduate program of the department of Informatics and Telecommunications of the National and Kapodistrian University of Athens. The main reason for signing this project was the need for a tool that would help the students visualize the teaching subject during the courses, since it could make possible the reduction of every computational geometry algorithm to an executable multimedia program with graphical input and output. This need was even more obvious after attending lectures of Computational Geometry during the postgraduate studies. Such previous knowledge in using CGAL has shown that programming in C++ could often discourage students from working on the development of algorithms. This concerns especially students with theoretical background, as well as for instructors that have time restrictions, which do not allow them to develop the algorithms that they teach. Thus, there is the need for a system that will provide the formulation of algorithms in ways of an enhanced pseudo code.

For the completion of this MSc thesis, I would like to thank my supervisors, Mr. I. Emiris and Mr. Ch. Fragoudakis who gave prominence to the necessity of this project and shared with me valuable knowledge and support.

1 INTRODUCTION

Experience in teaching Computational Geometry courses at the University of Athens [2] has proved that C++ programming using CGAL indicates a steep learning curve which most of the times frustrates learners. Using Python, a lot of C++'s cryptic syntax is avoided and the final program gains a lot in terms of readability and self expressiveness. Regarded from a pedagogical point of view, a computational geometry course should focus on the geometrical algorithmic aspects and somehow abstract the low level details. Python is the chosen programming language that fulfills exactly that kind of functionality; it has a remarkable expressiveness that permits learners to write programs that look like pseudo code, while advanced low level details are available but easily made transparent. The Python version that has been used in the current thesis is 2.6.6.

In order to ensure the simplicity of the code even for extreme cases where input and output data require arbitrary level of precision, a new arithmetic type is developed. This type handles, with “pure” Python, the limitations and issues occurring from the use of the floating-point arithmetic.

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. As a consequence, in general, the decimal floating-point numbers entered are only approximated by the binary floating-point numbers actually stored in the machine. This is in the very nature of the binary floating-point and is an issue arising not only in Python but also in all languages that support hardware’s floating-point arithmetic (although some languages may not display the difference by default or in all output modes).

A common example for such a representation error is the decimal value 0.1 which cannot be represented exactly as a base 2 fraction. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

“0.1000000000000000055511151231257827021181583404541015625”.

This problem becomes apparent when these values are used for arithmetic computations. For instance the sum (0.1 + 0.2) = 0.30000000000000004 could lead to unexpected results when it is further used. It is indicative that in a comparison between this result and number 0.3, the answer obtained would be negative. This sum also signifies the importance of dealing with the problems arising from the use of the standard floating point arithmetic, since they can affect even the sum of two such common decimals as (0.1+0.2).

Apart from the representation issues that arise in floating-point arithmetic, the rounding error due to precision limitations should also be mentioned. To get a clearer view of the problem, the IEEE 754 standard[1] should be presented. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). Many hardware floating point units use the IEEE 754 standard. In particular, the IEEE 754 standard specifies a double precision number as having one sign bit, an exponent width of 11 bits and a mantissa precision of 53 bits (52 explicitly stored).This gives from 15 - 17 digits precision in mantissa. If a decimal string with a maximum of 15 mantissa digits is converted to IEEE 754 double precision format and then converted back to the same number, then the final string should match the original; and if an IEEE 754 double

[1] ISO/IEC/IEEE 60559:2011

precision formatted decimal is converted to a decimal string with at least 17 mantissa digits and then converted back to double, then the final number must match the original.

The format is written with the mantissa having an implicit integer bit of value 1, unless the written exponent is all zeros. With the 52 bits of the fraction mantissa appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, 53 $\log_{10}(2) \approx 15.955$).

The real value assumed by a given 64-bit double-precision data with a given biased exponent e and a 52-bit fraction is

$$(-1)^{sign}(1.b_{-1}b_{-2}\ldots b_{-52})_2 \times 2^{(e-1023)}$$

or more precisely:

$$value = (-1)^{sign}\left(1+\sum_{i=1}^{52} b_{-i}\,2^{-i}\right)\times 2^{(e-1023)}$$

The double-precision binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 1023; also known as exponent bias in the IEEE 754 standard. The entire double-precision number is described by: $(-1)^{sign}\times 2^{exponent-exponent\_bias}\times 1.mantissa$

Between 2^{52}=4,503,599,627,370,496 and 2^{53}=9,007,199,254,740,992 the representable numbers are exactly the integers. For the next range, from 2^{53} to 2^{54}, everything is multiplied by 2, so the representable numbers are the even ones. Conversely, for the previous range from 2^{51} to 2^{52}, the spacing is 0.5.

The spacing as a fraction of the numbers in the range from 2^n to 2^{n+1} is 2^{n-52}. The maximum relative rounding error when rounding a number to the nearest representable one (the machine epsilon) is therefore 2^{-53}.

For the previously mentioned reasons, when the user needs exact computations, without precision dependencies and representation issues, they should use modules that support arbitrary precision arithmetic and exact representation of the numbers.

In the current diploma thesis, the related work on handling these problems is presented as well as a new arithmetic type, the MP_Float, which is coded in pure Python, with no further dependencies. This new type deals with the issues that arise from the use of the floating-point arithmetic and provides accurate and exact computations. Mp_Float is a wrapper for the “Decimal” type included in the decimal module of python. Apart from the exactness in representation that the “Decimal” type guarantees, MP_Float utilizes the ability of “Decimal” to change the precision according to the needs of the user. Specifically, it redefines the arithmetic operations so as to include the proper precision adjustment, which is needed in order to prevent any rounding error. In the next pages a presentation of the MP_Float follows. Moreover, an extensive analysis of the effects of the floating-point arithmetic on the orientation predicate and the incremental computation of convex hull is developed. In particular, the cases illustrated in [1] are reproduced with the use of Python as programming language. Concurrently, the respective results obtained by using Mp_Float, and Python/CGAL predicates instead of standard float are also described.

2 RELATED WORK

There are several approaches to assure reliable geometric computations. Firstly, there is the approach using the multi- institution European Community project CGAL [7], [8], Max-Planck Institute of Computer Science project LEDA [9], [10] and Core Library [11], which have adopted the exact geometric computation (EGC) paradigm. Although these libraries support exact arithmetic, currently there is no binding available that can be used directly in Python.

A second approach is to perturb the input so that the floating-point implementation is guaranteed to produce the correct result on the perturbed input [12], [13], [14], [15]. This approach requires a thorough problem analysis and a precise specification of input and output. When treating input data, the more complicated the objects are the more difficult is to specify the perturbed input. Output specification is not a trivial matter either, since the user must take into consideration every possibility especially for degenerate cases. In addition, if floating-point arithmetic is to be used, only few geometric algorithms are known which behave provably correct [16], [17], [18], [19].

There is also the decimal module provided by Python, which gives accurately represented decimal numbers and provides the user with capability to alter the precision according to their needs. This module is used in the current project in order to develop a more robust arithmetic type.

3 THE MP_Float()

Through this project a new arithmetic type (MP_Float) is developed with the use of pure Python. This type provides the user with the capability of using floating point arithmetic with arbitrary range of precision, which undertakes only the capacity limitations of the machine, and suffers no rounding due to fixed precision limitations. At the current state, floating-point in python uses double precision to store values. IEEE 754 double precision numbers have 53 bits of significant precision (52 explicitly used). This gives approximately 16 ($53\log_{10}(2) \approx 15.955$) mantissa digits precision, which is used to present both the integer part of the decimal and the mantissa.

Python already provides its users with the module “decimal” to support decimal floating point arithmetic. Decimal numbers can be represented exactly through the “Decimal” type provided and unlike hardware based floating-point, this module gives the user the ability to alter the precision according to the needs of the problem. However, the problem that the user is called to solve is the rounding of the numbers that could cause computational errors. Such errors can occur through any computation due to the specified precision. In order to help the user compute an arithmetic function correctly, without worrying about precision issues, it was considered important to define this new arithmetic type and its arithmetic operators. This could boost python in a way that could make pseudo coding more feasible.

The name MP_Float (Multi-Precision Float) is also referred in CGAL, but the current type is not related to CGAL coding. The point here is to highlight that the floating point numbers with arbitrary precision are represented exactly, without undergoing any representation or rounding error. In parallel, any numeric process should give the expected and exact results.

3.1 The “Duck typing” restrictions

The arithmetic type of the MP_Float has one attribute, the number. The decimal that is passed as an argument, when the MP_Float is created, is converted to a “Decimal” so as to exploit the advantages of the decimal module. However, Duck typing is heavily used in Python. The Python's Glossary[2] defines duck typing as follows:

“Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). Instead, it typically employs the EAFP (Easier to Ask Forgiveness than Permission) style of programming.”

Therefore the type of the arguments passed in the definition of a function cannot be specified in advance. As far as MP_Float is concerned, when a decimal with a long mantissa (bignum) is passed as a parameter without quotation marks, the code that processes the parameter will make certain assumptions about it. As a result, the bignum is considered as a float. Since the current precision is smaller than that of the bignum, it is rounded to the nearest float. In order to prevent this from happening, the numbers passed as parameters in the MP_Float should be included in quotation marks. In case the bignum is not passed as a string, it will cause a warning to pop up, indicating some probable rounding on the number.

[2] http://docs.python.org/2/glossary.html#term-duck-typing

3.2 Development of functions and operators

To facilitate the further function of the code, some auxiliary methods have been defined. Such are the computation of the length of digits of the bignum, the length of the integer part and that of the mantissa, as well as the integer part and the mantissa themselves.

In an effort to avoid any kind of rounding and representation error as far as possible for every operation between two numbers, the precision that is needed, must also be predefined. In Computational Geometry the most commonly used operations are addition, subtraction and multiplication. For these operations, the precision can be predefined to a specific minimum and safe length which will guarantee that there will be no rounding during and after the completion of the operation. This means that the result will be the expected one.

As far as addition is concerned, analyzing the process of adding two decimal numbers helps in acquiring a proper precision. The steps followed are writing the numbers one under the other with the decimal points lined up, padding in zeros so the numbers have the same length and then adding normally. The minimum precision needed for an addition to be correctly computed, is the sum of the maximum length of the integer part and the maximum length of the mantissa of the two numbers concerned, plus one. When adding the two mantissas the number of digits that occur is the same as the maximum length of the two mantissas. When adding the integer parts, the number of digits may be increased with one digit farther from the maximum number of digits of the integer part. However, in order to reduce the complexity of defining the precision, the sum of the length of the two numbers is being used. This sum contains both the maximum length of the integer parts and the maximum length of the mantissa, as well as the minimum ones. Since the numbers should have at least one integer digit and one digit in mantissa, the excessive digits of the sum of the lengths are at least two and at most the same as the sum of the maximum integer part and the maximum mantissa. Despite wasting some digits in precision, there is a gain in computational time.

The precision for subtraction is respectively computed. Since subtraction equals with the addition of the first number with the negation of the second, the maximum length of the resulting number is the sum of the maximum integer part and mantissa plus one. This increment of one more digit happens only when the numbers subtracted are of a different sign. In the case where the two numbers considered have the same sign, the length in digits of their difference belongs in the range from 2 digits (e.g 0.0) to (maximum integer part + maximum mantissa). Thus, the precision needed is covered profoundly with the sum of the lengths of the two numbers.

The process of multiplication with decimals is the same as with integers, apart from the fact that the decimal point should be correctly placed within the resulting number. In the multiplication of each digit of the second decimal with the first one, there is a left shift to the corresponding product. The number of shifts equals the number of the digits of the second decimal minus one, since the first product is not shifted. When multiplying two integer numbers with only one digit each, the resulting product has at most two digits. As a result, each product will have at most so many digits as the first decimal of the multiplication plus one. Finally, there is the addition of the products just as they are lined, after the shifted products have been padded with zeros. The longest number (as far as digits are concerned) is the last one, that has so many zeros (from the padding) as the number of shifts and at most so many digits as the first decimal plus one. The total number of digits is given by the sum of the length of the two decimals which are multiplied. When adding the products, reaching the most significant digit on the left, there will be at most two digits to be added together. This addition could give at most a two digit result. Therefore, the maximum length of the result of the multiplication is the

sum of the length of the first decimal and the length of the second decimal plus one. This is also the precision that has been predefined in the code for the operation of the multiplication. Aiming to further clarify the case in question, the next figure shows a simple example of decimal multiplication.

```
      999.9 (a) ——> len(a) = 4
  x     9.99 (b) ——> len(b) = 3
 ----------
     89991
    899910
+  8999100        #shifts = len(b) - 1
 ----------
   9989.001   ——> len(result) = len(a) + 1 + (#shifts)
                              = len(a) + len(b)
```

Figure 1:
Typical execution of the multiplication (999.9 * 9.99), giving the maximum length of the result that can be obtained from a four-digit decimal multiplied by another three-digit decimal.

Except for these operations, some additional methods for this new type have been defined. At first, there is the method of comparison, with the precision being equal to the maximum of the decimals concerned. For the rest, such as sqrt, div, pow, mod, divmod and floordiv, the resulting decimals could require an infinite number of digits, and thus the precision has arbitrarily been defined as the value of the maximum number of digits between the decimals involved and the previous precision. Apart from these operators, corresponding functions (div, power, mod, divmodule, floordiv), which take the precision as an argument, have also been defined. It has also been considered useful to define the corresponding reverse and in-place operators in order to complete the arithmetic type of MP_Float. The whole implementation is provided in the APPENDIX A.

4 ROBUSTNESS PROBLEMS IN GEOMETRIC COMPUTATIONS

4.1 Planar Orientation Predicate

Many computational geometry applications use numerical tests known as orientation tests. These tests determine whether a point lies to the left of, to the right of or on the line defined by two other points. Let the points that form the line be $p=(p_x, p_y)$, $q=(q_x, q_y)$ and the point of question be $r=(r_x, r_y)$. The orientation of the triple (p, q, r) is tantamount to the sign of the determinant:

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}\right) .$$

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x) \right) .$$

In particular, a left turn is implied when orientation is positive (+1), a right turn when negative (-1) and colinearity when orientation equals to zero. The determinant is expressed in terms of the coordinates of the points. If these coordinates are expressed as single or double precision floating-point numbers, an error of rounding may lead to an incorrect result when the true determinant is near zero. Let's call this orientation float_orient(p, q, r) to distinguish it from the ideal predicate. There are potentially three ways in which the result of float_orient could differ from the correct orientation.

Rounding to zero: misclassify a + or – as a 0;

Perturbed zero: misclassify 0 as a + or -;

Sign inversion: misclassify a + as – or vice versa.

4.1.1 Geometry of Float-Orientation

According to Kettner et al. [1], the following experiment can define the geometry of float_orient. Three points p, q and r are chosen and then the following float_orient is computed: float_orient((p_x+xu, p_y+yu), q, r) for $0 \le x, y \le 255$, where u is the increment between adjacent floating-point numbers in the considered range; for example, $u = 2^{-53}$ if $p_x = 1/2$ and $u = 4 \cdot 2^{-53}$ if $p_x = 2 = 4 \cdot 1/2$. The resulting 256x256 array of signs is visualized as a 256x256 grid of colored pixels. A yellow (red, blue) pixel represents collinear (negative, positive, respectively) orientation.

The way Kettner et al. [1] computes the point (p_x+xu, p_y+yu) in their source code using C++ is depending on bitwise operations. Specifically, they are taking into consideration all 64 bits of the double and how they are distributed in order to make the proper bitwise additions and shifts. This way they compute the minimum value of u that should be added to the value of a point coordinate so as to exploit the spacing between contiguous double values.

In the current project, the same experiment has been conducted in Python. The functions that have been developed here provide the geometry of orientation both for standard float and MP_Float numbers. The instructions given by the paper's writers and the spacing range, which emerges from the IEEE 754 standard, are used as a guide for the development of these functions. The main algorithm conducts the experiment of slightly adding a disturbance to the coordinates of the first of the three points, and then computes the respective orientation predicate. The disturbance u, is in fact the spacing 2^{n-52}, when the value of the coordinate processed is in the range 2^n to 2^{n+1}. The data given by these functions are processed with OpenGl code which provides the

visualization of the geometry of the orientation predicate, and is included in the Appendix B.

4.1.2 Examples

There are many examples where the geometry of float_orientation is not the expected one. Primarily, the examples shown in the paper of Kettner et al.[1] in Figure2[3] are considered, and a couple more that are provided here as well. The figures show the weird geometry of the float-orientation predicate. Specifically, the first two figures show the results of float_orient((p_x+xu, p_y+yu), q, r) for 0 ≤ x, y ≤ 255, where $u=2^{-53}$ is the increment between adjacent floating-point numbers in the considered range. The first figure is the product of the Kettner et al. [1] source code and it is the same image as the one which was obtained by the respective python code. The second figure shows the result of the same experiment while using the MP_Float type in the same python code. The third one illustrates the resulting floating point orientation geometry when it is computed with the use of CGAL. In the final case, the orientation predicate used is the one provided by CGAL. Every figure is referring to the same values of point coordinates. The result is color coded: Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation.

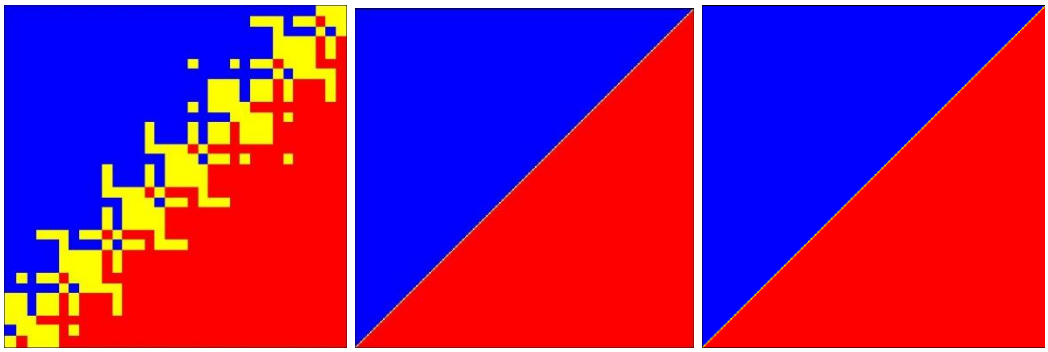


Figure 2:
p = (0.5, 0.5), q = (12, 12), r = (24, 24)

The points p, q, r from Figure 2 give an orientation(p, q, r)=0 with every implementation. However, a small disturbance to one of the points by a factor of 2^{-53} can cause many miscalculations on the orientation predicate when floating point arithmetic is used. When computing the orientation using the MP_Float or the orientation predicate provided by CGAL, the result is the correct one for every point forming the orientation geometry.

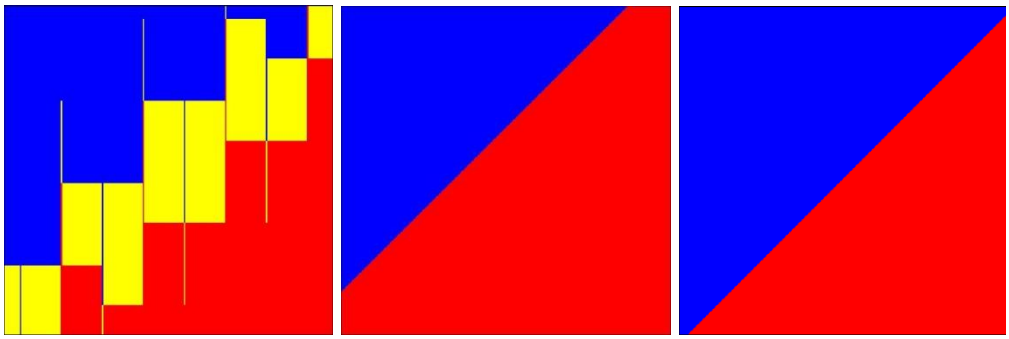


Figure 3:
p = (0.50000000000002531, 0.5000000000000171)
q = (17.300000000000001, 17.300000000000001)
r = (24.00000000000005, 24.0000000000000517765)

[3]See [1], page 5

Points read using float :	**Points read using MP_Float :**
p0=0.50000000000002531, 0.5000000000000171 p1=17.300000000000001, 17.300000000000001 p2=24.00000000000005, 24.000000000000053 for points:p0=p, p1=q, p2=r A=q.x - p.x = 16.799999999999976 B=r.y - p.y = 23.500000000000036 C=q.y - p.y= 16.799999999999983 D=r.x - p.x = 23.500000000000025 A*B=394.80000000000001 C*D=394.80000000000001 float_orient(p, q, r) = 0 for points:p1=p, p2=q , p0=r A=q.x - p.x = 6.700000000000049 B=r.y - p.y = -16.799999999999983 C=q.y - p.y= 6.7000000000000526 D=r.x - p.x = -16.799999999999976 A*B=-112.56000000000071 C*D=-112.56000000000073 float_orient(p, q, r) > 0 for points:p2=p p0=q , p1=r A=q.x-p.x = -23.500000000000025 B=r.y - p.y = -6.7000000000000526 C=q.y - p.y= -23.500000000000036 D=r.x - p.x = -6.700000000000049 A*B=157.45000000000141 C*D=157.45000000000138 float_orient(p, q, r) > 0	p0=0.50000000000002531, 0.5000000000000171 p1=17.300000000000001, 17.300000000000001, p2=24.00000000000005, 24.0000000000000517765 for points:p0=p, p1=q, p2=r A=q.x - p.x = 16.79999999999997569 B=r.y - p.y = 23.5000000000000346765 C=q.y - p.y= 16.7999999999999839 D=r.x - p.x = 23.50000000000002469 A*B=394.80000000000001128019999999157014285 C*D=394.80000000000003644199999999602491 exact_orient(p, q, r) < 0 for points:p1=p, p2=q, p0=r A=q.x - p.x = 6.700000000000049 B=r.y - p.y = -16.7999999999999839 C=q.y - p.y= 6.7000000000000507765 D=r.x - p.x = -16.79999999999997569 A*B=-112.560000000000715329999999992111 C*D=-112.560000000000690168199999998765623285 exact_orient(p, q, r) < 0 for points:p2=p, p0=q, p1=r A=q.x - p.x = -23.50000000000002469 B= r.y - p.y = -6.7000000000000507765 C=q.y - p.y= -23.5000000000000346765 D=r.x - p.x = -6.700000000000049 A*B=157.450000000001358670750000001253671785 C*D=157.450000000001383832550000016991485 exact_orient(p, q, r) < 0

Points read using CGAL:

p0=0.50000000000002531 , 0.5000000000000171

p1=17.300000000000001 , 17.300000000000001

p2=24.00000000000005 , 24.000000000000053

CGAL.orientation(p0, p1, p2) = LARGER

CGAL.orientation(p1, p2, p0) = LARGER

CGAL.orientation(p2, p0, p1) = LARGER

In the example of Figure 3, the coordinates of the points p and q are exactly represented in all three implementations. The third point (r), has an abscissa with 19 digits of mantissa. Therefore, the regular float cannot represent the coordinate exactly and this affects the result of the orientation predicate with or without the addition of a disturbance to the coordinates. The MP_Float type represents exactly the certain coordinate and the orientation presents the correct geometry. As far as CGAL is concerned, due to duck – typing, the coordinates that are passed to CGAL in order to create the respective point, are considered as simple floats by python and as CGAL::Simple_cartesian values by CGAL. The latter type uses double-precision arithmetic, just like the "double" arithmetic type. This led CGAL to see the wrong representation of the r.y coordinate. However, during the process of computing the orientation predicate, the numbers are converted to CGAL::Lazy_exact_nt, which is a type that assures exact computations. This is the reason why the orientation geometry provided by CGAL seems correct. In fact, it is accurate for the inexact values of point coordinates that it is processing.

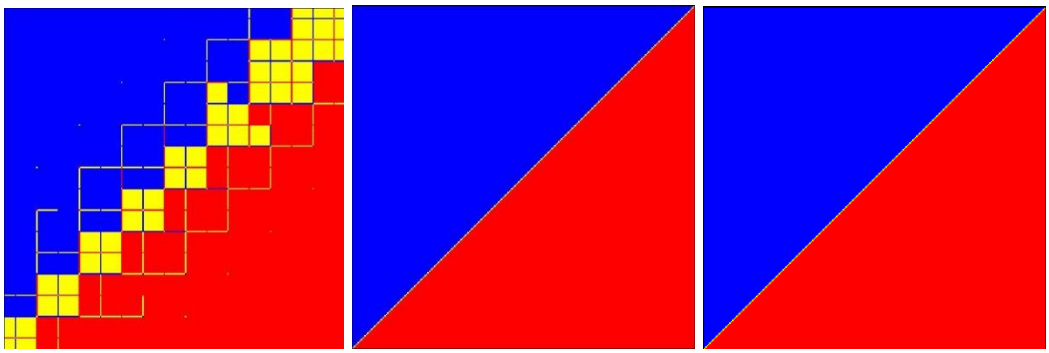


Figure 4:
p = (0.5, 0.5),
q = (8.8000000000000007, 8.8000000000000007)
r = (12.1, 12.1)

The points presented in Figure 4 are also correctly represented. Every implementation gives orientation(p, q, r) = 0 for these points. Once again, the geometry obtained by floating point arithmetic significantly deviates from the expected one. Instead, the orientation that is using the MP_Float, as well as the CGAL orientation predicate, produces the expected geometry.
A few interesting examples also provided by Kettner et al. [1] give the following results:

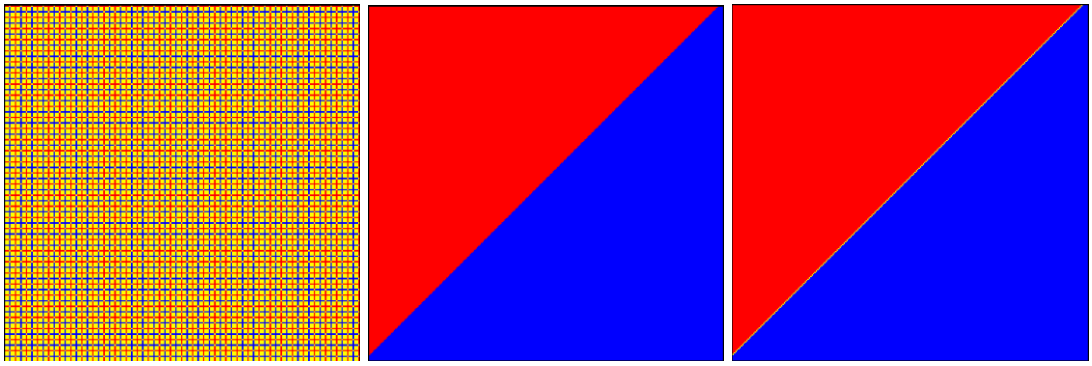


Figure 5:
p = (7.3, 7.3),
q = (24.000000000000068, 24.000000000000071)
r = (24.00000000000005, 24.000000000000053)

Points read using float :	**Points read using MP_Float:**
p0=7.2999999999999998,7.2999999999999998	p0=7.3, 7.3
p1=24.000000000000068,24.000000000000071	p1=24.000000000000068, 24.000000000000071,
p2=24.00000000000005,24.000000000000053	p2=24.00000000000005, 24.000000000000053
for points: p0=p, p1=q, p2=r	for points: p0=p, p1=q, p2=r
A=q.x - p.x = 16.700000000000067	A=q.x - p.x = 16.700000000000068
B=r.y - p.y = 16.700000000000053	B=r.y - p.y = 16.700000000000053
C=q.y - p.y= 16.70000000000007	C=q.y - p.y= 16.700000000000071
D=r.x - p.x = 16.700000000000049	D=r.x - p.x = 16.70000000000005
A*B=278.89000000000198	A*B=278.8900000000020207000000000003604
C*D=278.89000000000198	C*D=278.8900000000020207000000000000355
float_orient(p, q, r) = 0	exact_orient(p, q, r) >0
for points: p1=p, p2=q, p0=r	for points: p1=p, p2=q, p0=r
A=q.x - p.x = -1.7763568394002505e-14	A=q.x - p.x = -1.8E-14
B=r.y - p.y = -16.70000000000007	B=r.y - p.y = -16.700000000000071
C=q.y - p.y= -1.7763568394002505e-14	C=q.y - p.y= -1.8E-14
D=r.x - p.x = -16.700000000000067	D=r.x - p.x = -16.700000000000068
A*B=2.9665159217984309e-13	A*B=3.00600000000001278E-13
C*D=2.9665159217984304e-13	C*D=3.00600000000001224E-13
float_orient(p, q, r) >0	exact_orient(p, q, r) >0
for points: p2=p, p0=q, p1=r	for points: p1=p, p2=q, p0=r
A=q.x - p.x = -16.700000000000049	A=q.x - p.x = -16.70000000000005
B=r.y - p.y = 1.7763568394002505e-14	B=r.y - p.y = 1.8E-14
C=q.y - p.y= -16.700000000000053	C=q.y - p.y= -16.700000000000053
D=r.x - p.x = 1.7763568394002505e-14	D=r.x - p.x = 1.8E-14
A*B=-2.9665159217984269e-13	A*B=-3.0060000000000090E-13
C*D=-2.9665159217984274e-13	C*D=-3.00600000000000954E-13
float_orient(p, q, r) >0	exact_orient(p, q, r) >0

Points read using CGAL:

p0=7.2999999999999998 , 7.2999999999999998

p1=24.000000000000068 , 24.000000000000071

p2=24.00000000000005 , 24.000000000000053

CGAL.orientation(p0, p1, p2) = LARGER

CGAL.orientation(p1, p2, p0) = LARGER

CGAL.orientation(p2, p0, p1) = LARGER

The example in Figure 5 is showing a representation error in the values of the point coordinates. As a result, the orientation geometry obtained by floating point orientation and CGAL orientation predicate is bound to be inaccurate.

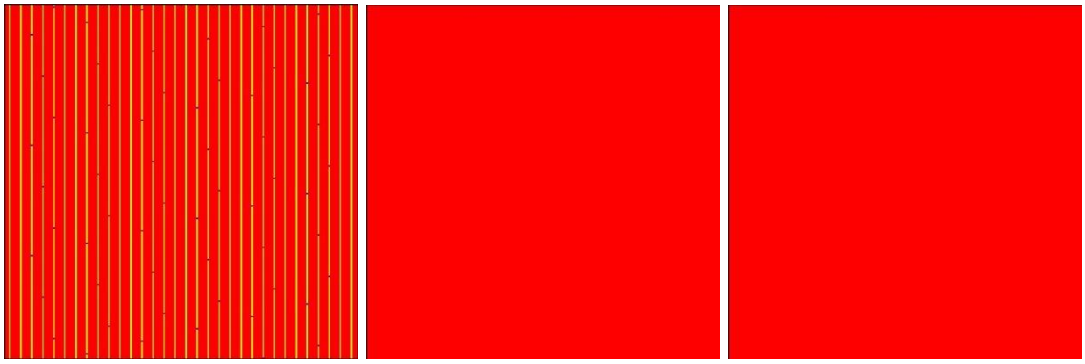


Figure 6:
p = (7.8, 10.49),
q = (24.000000000000068, 24.000000000000071)
r = (24.00000000000005, 24.000000000000053)

Points read using float :	**Points read using MP_Float :**
p0=7.7999999999999998,10.49	p0= 7.8, 10.49
p1=24.000000000000068,24.000000000000071	p1=24.000000000000068, 24.000000000000071
p2=24.00000000000005,24.000000000000053	p2= 24.00000000000005, 24.000000000000053
Points read using CGAL:	
p0=7.7999999999999998,10.49	
p1=24.000000000000068,24.000000000000071	
p2=24.00000000000005,24.000000000000053	

The point coordinates from Figure 6 are not exactly represented by floating point arithmetic, causing a completely unexpected geometry for float orientation. Even though the points are clearly non-collinear, float_orient presents a geometry where many points are thought to be collinear, when a few others seem to give the opposite sign after the computation of the orientation. Although, the representation of the points is inexact, the CGAL orientation provides the correct geometry.

In order to make more test cases, there has been an attempt to create a generator of floating point coordinates. These points indicate the problem while computing the predicate of orientation with the use of floating point arithmetic. The resulting figures show the abnormalities that occur with the orientation predicate, some at a narrower range of values and other at a wider one. The geometry obtained when the values are processed with the MP_Float type, is the expected one, for every three points. A couple more examples are presented below.

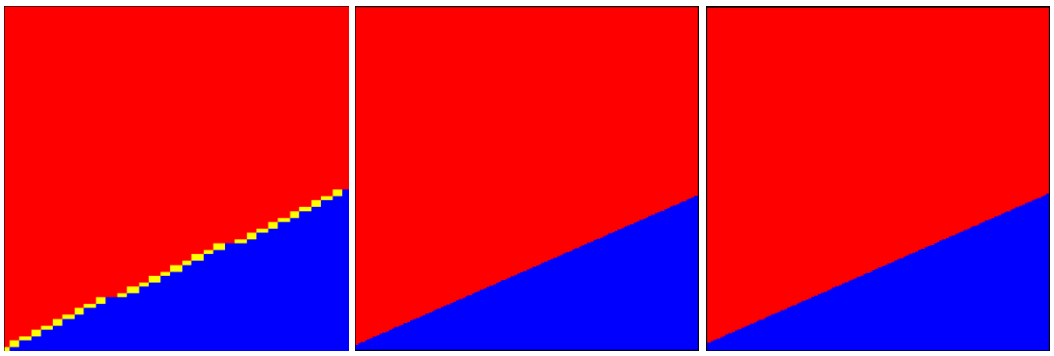


Figure 7:
0.9776094757002022, 13.843266329901409
563.15368022071505, 3949.0757615450054
6.1752503355833230, 50.226752349083261

Points read using float :	**Points read using MP_Float :**
p0=0.97760947570020218, 13.843266329901409	p0= 0.9776094757002022, 13.843266329901409
p1=563.15368022071505, 3949.0757615450052	p1=563.15368022071505, 3949.0757615450054
p2=6.175250335583323, 50.226752349083263	p2= 6.1752503355833230, 50.226752349083261
Points read using CGAL:	
p0=0.97760947570020218, 13.843266329901409	
p1=563.15368022071505, 3949.0757615450052	
p2=6.175250335583323, 50.226752349083263	

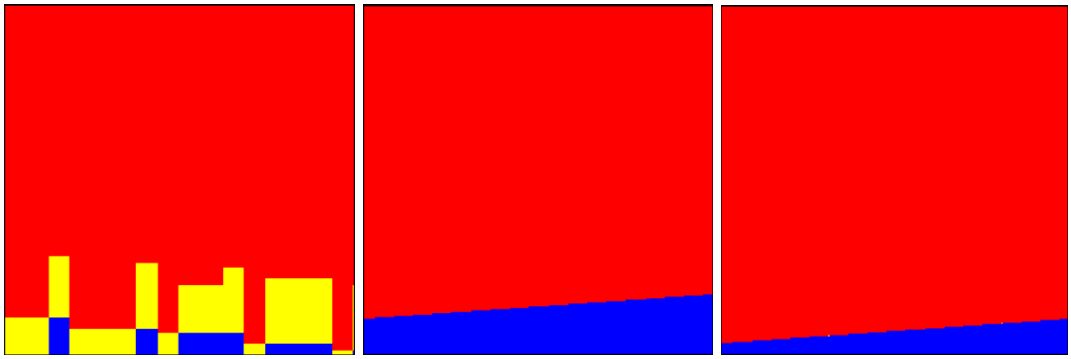


Figure 8:
0.1927460515658708, 25.734714464092815
8.8733976801358919, 103.86057912122302
8.1260616400516564, 97.134554760464907

Points read using float :	**Points read using MP_Float :**
p0=0.1927460515658708, 25.734714464092814	p0= 0.1927460515658708, 25.734714464092815
p1=8.8733976801358914, 103.86057912122303	p1= 8.8733976801358919, 103.86057912122302
p2=8.1260616400516561, 97.134554760464908	p2= 8.1260616400516564, 97.134554760464907
Points read using CGAL:	
p0=0.1927460515658708, 25.734714464092814	
p1=8.8733976801358914, 103.86057912122303	
p2=8.1260616400516561, 97.134554760464908	

All diagrams exhibit block structure. According to Kettner et al. [1], this fact can be explained when focusing on one dimension. At first, y is kept fixed, and then, $\text{float\_orient}((p_x + xu, p_y + yu), q, r)$ is evaluated, for $0 \le x \le 255$, where u is the increment between adjacent floating-point numbers in the considered range. Since the orientation predicate is:

$$\text{orientation}(p, q, r) = \text{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)),$$

round-off errors can occur in the additions/subtractions and also in the multiplications. The differences that are varying in this predicate, are (q_x - p_x) and (r_x - p_x). Figure 2 can be used as an example when considering each of these differences. In this example q_x=12 and $p_x \approx 0.5$ (due to its increment with xu). Since $2^{-1} \le 0.5 < 2^0$, the increment u equals 2^{-53}, which means that the representable numbers greater than 0.5 differ by a factor 2^{-53}. Additionally, $2^3 \le 12 < 2^4$. Since the spacing between the representable numbers within the range [2^3, 2^4) is equal to 2^{3-52}=2^{-49}, addition or subtraction with any number between 2^{-53} and 2^{-49} has no effect on the numbers in this range. As a result, the last four bits of x are lost in the subtraction. In other words, the result of the subtraction (qx−px) is constant for 2^4 consecutive values of x. Because of rounding to nearest, the intervals of constant value are [8, 23], [24, 39], [40, 55],… .Similarly, the floating-point result of rx− px is constant for 2^5 consecutive values of x. Because of rounding to nearest, the intervals of constant value are [16, 47], [48, 69],… .Overlaying the two progressions gives intervals [16, 23], [24, 39], [40, 47], [48, 55],… and this explains the structure illustrated in the rows of Figure 2. The image shows short blocks of length 8, 16, 24,… . In figures 3 and 4, the situation is more complicated. It is again true that there are intervals for x, where the results of the subtractions are constant. However, since q and r have more complex coordinates, the relative shifts of these intervals are different and hence the features are more narrow and broad.

The results which are obtained by the source code given by Kettner et al. [1] and the python code presented here, are the same for the respective test cases of float orientation. This reassures that the values that are used for the orientation geometry are the same. For these values, the resulting geometry which is acquired from the “MP_Float” orientation predicate indicates that the computation of the orientation is exact and the expected one. The values for the later orientation predicate are using explicitly the MP_Float arithmetic. This way the orientation geometry can be used as a verification of the exactness of the computations when using the MP_Float instead of the standard floating point arithmetic.

4.2 Planar Convex Hull Problem

A set is called convex if, for any two points p and q in the set, the entire line segment pq is contained in the set. The convex hull CH of a set S of points is the smallest (with respect to the set inclusion) convex set containing S. A point $p \in S$ is called an extreme point of S if there is closed halfspace containing S such that p is the only point in S that lies in the boundary of the halfspace.

4.2.1 Incremental Convex Hull Algorithm

The incremental algorithm maintains the current convex hull CH of the points seen so far. Initially, CH is formed by choosing three non-collinear points in S. It then considers the remaining points one by one. When considering a point r, it first determines whether r is outside the current convex hull polygon. If not, r is discarded. Otherwise, the hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew’s (Andrew, 1979) and other variants of

Graham's scan (Graham, 1972) and also in the randomized incremental algorithm (Clarkson and Shor, 1989).

The algorithm maintains the current hull as a circular list $L=(v_0,v_1,\ldots,v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i,v_{i+1}), $0 \le i < k-1$ (indices are modulo $k \ge 3$) are the edges of the current hull. If orientation$(v_i,v_{i+1},r) < 0$, r sees the edge (v_i,v_{i+1}) and the edge (v_i,v_{i+1}) is visible from r. If orientation$(v_i,v_{i+1},r) \le 0$, the edge (v_i,v_{i+1}) is weakly visible from r. The following properties are the basis of the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i,v_{i+1}),\ldots,(v_{j-1},v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1},\ldots,v_{j-1})$ by r. The subsequence $(v_i,\ldots,v_j)$ is taken in the circular sense. For instance, if i > j then the subsequence is $(v_i,\ldots,v_{k-1},v_0,\ldots,v_j)$. Depending on these properties, the following algorithm has been derived:

INCREMENTAL CONVEX HULL ALGORITHM

Initialize L to the counter-clockwise triangle (a,b,c). Remove a,b,c from S.
for all r $\in$ S do
    iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, discard r. Otherwise, take any one of the visible edges as the starting edge for the next item.
    if there is an edge e visible from r then
        - compute the sequence $(v_i,\ldots,v_j)$ of edges that are weakly visible from r.
        Starting from a visible edge e, move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, move clockwise from e until a non-weakly-visible edge is encountered.
        - Replace the subsequence $(v_{i+1},\ldots,v_{j-1})$ by r:
        Delete the vertices in $(v_{i+1},\ldots,v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy").
    end if
end for

The detailed implementation is presented in Appendix C and has been used in every experimental process described in this thesis.

There are four logical ways to negate properties A and B.

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_2: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 contradict to Property B and are feasible if failure A_1 is also taken into account.

4.2.2 Instance presentation and failure analysis

In this section, some instances are presented, which indicate the violation of the correctness properties in the algorithm. Specifically, these instances are represented by

sequences of points p_1, p_2, p_3, … such that the first three points form a counter-clockwise triangle (and float_orient correctly discovers this) and such that the insertion of some later point leads to a violation of a correctness property (in the computations with float_orient). All the examples contain nearly or truly collinear points; in the view of a standard rounding-error analysis sufficiently non-collinear points would not cause any problems. This fact may seem unrealistic. However, many point sets contain nearly collinear points or truly collinear points, which become nearly collinear by conversion to floating-point representation.

Failure A_1: A point outside the current hull sees no edge of the current hull:

In order to comprehend the way that such a failure can affect the result of the algorithm, let us consider the following set of points.

p_1= (7.3000000000000194, 7.3000000000000167),

p_2 = (24.000000000000068, 24.000000000000071),

p_3 = (24.00000000000005, 24.000000000000053),

p_4 = (0.50000000000001621, 0.50000000000001243),

p_5 = (8, 4), p_6 = (4, 9), p_7 = (15,27),

p_8 = (26,25), p_9 = (19,11)

The four first points are those that are affected the most by the floating point representation issues and thus those that cause miscalculations within the algorithm. These points lie almost on the line y = x, and float_orient gives the following results:

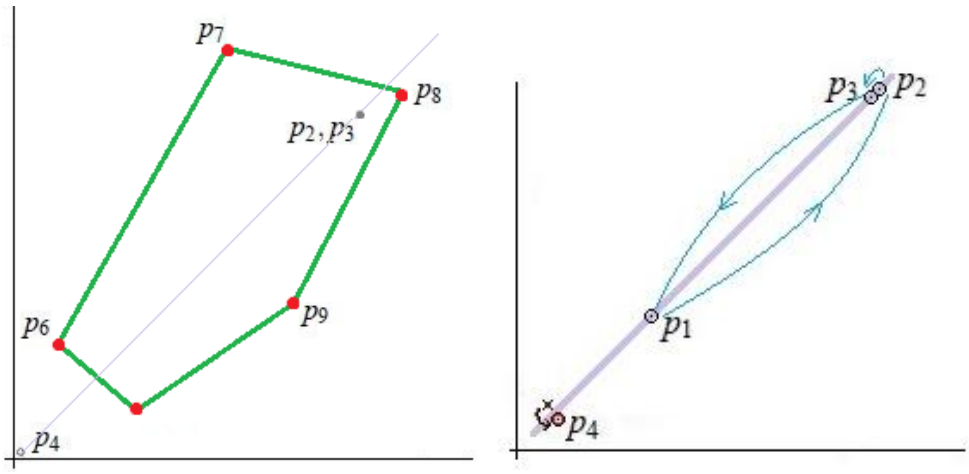
exact_orient = float orient(p_1, p_2, p_3) > 0

exact_orient = float orient(p_1, p_2, p_4) > 0

exact_orient = float orient(p_2, p_3, p_4) > 0

exact_orient ≠ float orient(p_3, p_1, p_4) > 0

hull = <p5, p9, p8, p7, p6>



(a) (b)

Figure 9:
(a) The convex hull illustrating Failure A_1: The point p_4 in the lower left corner is left out of the hull.
(b) Schematic view indicating the position where float_orient thinks of p_4 to be: it lies to the left of all sides of the triangle (p_1, p_2,p_3) (indicated by a circle ○). In fact, it lies on the right of the line defined by the edge (p_3, p_1) (as indicated by the x symbol in the figure).

Figure 9(a) shows the computed convex hull, where a point that is clearly extreme was left out of the hull. The first three points are correctly considered to form a counter-clockwise triangle. Nevertheless, when the algorithm processes the point p_4 by

computing float_orient, the last evaluation gives wrong result. Geometrically, these four evaluations of float_orient, in order to update the current hull, say that p_4 sees no edge of the triangle (p_1, p_2, p_3), and leaves it out of the hull. The points from p_5 to p_9 are then correctly identified as extreme points and are added to the hull. However, the algorithm never recovers from the error made when considering p_4 and the result of the computation differs drastically from the correct hull.

So as to understand how floating point arithmetic affects the orientation predicate and as a result, the outcome of the algorithm, the contentious evaluation should be considered step by step. Specifically, let's see how numbers are processed with the use of floating point arithmetic.

float_orient(p_3, p_1, p_4):

p = p_3 = (24.00000000000005, 24.000000000000053)

q = p_1 = (7.3000000000000194, 7.3000000000000167)

r = p_4 = (0.50000000000001621, 0.50000000000001243)

A = q.x - p.x = -16.700000000000031

B = r.y - p.y = -23.500000000000043

C = q.y - p.y = -16.700000000000038

D = r.x - p.x = -23.500000000000032

A*B = 392.45000000000147,

C*D = 392.45000000000141

float_orient (p,q,r) = sign((A*B) – (C*D)) > 0

The points of interest here are p, q, r shown above. As it can be noticed, the representation of the coordinates of the points is not affected in the first place. However, the fact that these numbers belong to a different range of numbers ($2^n \leq k < 2^{n+1}$)[4], while requiring every bit available for the presentation of the mantissa, leads to miscalculation when they are combined in a computation. This is also obvious from the subtractions A, B, C and D which do not give the expected results. The problem progresses even further after the multiplications among them.

The same orientation predicate has been computed using the MP_Float type:

exact_orient(p_3, p_1, p_4):

p = p3 = (24.00000000000005, 24.000000000000053)

q = p1 = (7.3000000000000194, 7.3000000000000167)

r = p4 = (0.50000000000001621, 0.50000000000001243)

A = q.x - p.x = -16.7000000000000306

B = r.y - p.y = -23.50000000000004057

C = q.y - p.y= -16.7000000000000363

D = r.x - p.x = -23.50000000000003379

A*B = 392.45000000000139661900000001241442

C*D= 392.45000000000141734300000001226577

exact_orient(p,q,r) = sign((A*B) – (C*D)) < 0

[4] Refered to in section 3.1.1, page 11

The respective results obtained are correct and exact. The correctness can also be checked within a few minutes on the paper.

Such examples, where the point is miscalculated but it obviously belongs to the hull, are defined by certain characteristics. They start with a triangle with two almost parallel sides (<p_1,p_2> and <p_3,p_1>) and with a query point (p_4) near the wedge defined by the two nearly parallel edges. This query point lies almost on the extension of the two parallel sides. There can also be another set of points, where the initial triangle does not need to be almost parallel. In this case, after getting three random points that form a counter-clockwise triangle, the fourth point is almost collinear with two of the three points and near to the edge of the triangle formed by those two points. The notion of these examples is shown in Figure 10 below.

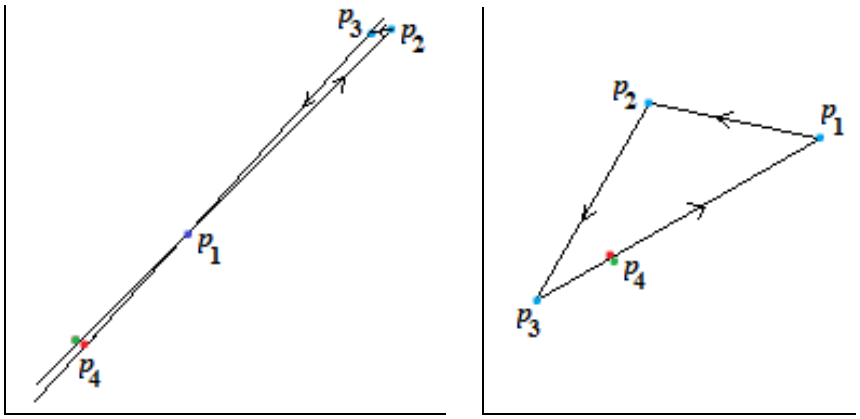


Figure 10:
(a) A zooming of the example in figure 9. The green spot shows where p_4 really lies and the red one where orientation concerning p_3, p_1 thinks of it. **(b)** A random example where p_4 is outside of the triangle (green spot) but orientation concerning the edge <p_3, p_1> moves r_4 to the red spot.

Failure A_2: A point inside the current hull sees an edge of the current hull:

Finding such an example is easy. Starting with any counter-clockwise triangle, the fourth point should be chosen inside the triangle but close to one of the edges. The distance of the fourth point to the edge should be such that a small disturbance due to computations would falsely make the algorithm believe that it lies outside the current hull.

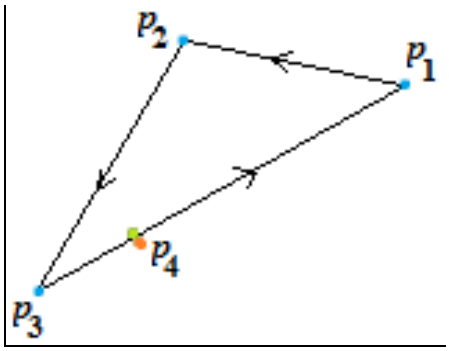


Figure 11:
A random counter-clockwise triangle p_1, p_2, p_3 and a point p_4 next to the edge <p_3, p_1>. The point p_4 lies inside the triangle (green spot) but orientation sees it outside (red spot).

A concrete example follows:

p_1 = (27.643564356435643, −21.881188118811881)
p_2 = (83.366336633663366, 15.544554455445542)
p_3 = (4.0, 4.0)
p_4 = (73.415841584158414, 8.8613861386138595)

The convex hull is correctly initialized to (p_1, p_2, p_3). Then the float_orient algorithm gives the following results when considering p_4 regarding each edge of the current hull.

exact_float = float orient(p_1, p_2, p_3) > 0

exact_float ≠ float orient(p_1, p_2, p_4) < 0

exact_float = float orient(p_2, p_3, p_4) > 0

exact_float = float orient(p_3, p_1, p_4) > 0

The point p_4 is inside the current convex hull, but the algorithm incorrectly believes that p_4 can see the edge (p_1, p_2) and hence changes the hull to (p_1, p_4, p_2, p_3), a slightly non-convex polygon.

By watching the outcome of the computations for both the float_orient and the exact_orient, again, it can be noticed that even though the points are represented intact at the beginning, during the process many miscalculations occur.

float_orient(p_1, p_2, p_4):	**exact_orient(p_1, p_2, p_4):**
p = (27.643564356435643, -21.881188118811881)	p = (27.643564356435643, 21.881188118811881)
q = (83.366336633663366, 15.544554455445542)	q = (83.366336633663366, 15.544554455445542)
r = (73.415841584158414, 8.8613861386138595)	r = (73.415841584158414, 8.8613861386138595)
A = q.x - p.x = 55.722772277227719	A = q.x - p.x = 55.722772277227723
B = r.y - p.y = 30.742574257425741	B = r.y - p.y = 30.7425742574257405
C = q.y - p.y= 37.425742574257427	C = q.y - p.y= 37.425742574257423
D = r.x - p.x = 45.772277227722768	D = r.x - p.x = 45.772277227722771
A*B = 1713.0614645622975	A*B = 1713.0614645622977057357219880403881 5
C*D = 1713.0614645622977	C*D = 1713.0614645622976406047446328791 33
float_orient (p,q,r) = sign((A*B) – (C*D)) < 0	exact_orient(p,q,r) = sign((A*B) – (C*D)) > 0

Failure B_1: A point outside the current hull sees all edges of the convex hull:

To construct such an example, one should start with a counter- clockwise triangle with one angle close to π and hence, three almost parallel sides. The query point should be placed near one of the sides so that it could see two of the sides and “float- see” the third. An example as such follows:

p_1 = (200.0, 49.200000000000003)
p_2 = (100.0, 49.600000000000001)
p_3 = (−233.33333333333334, 50.93333333333333)
p_4 = (166.66666666666669, 49.333333333333336)

Let us check each orientation predicate separately, both with the regular float and the MP_Float type.

float orient(p_1, p_2, p_3) > 0	exact orient(p_1, p_2, p_3) < 0
float orient(p_1, p_2, p_4) < 0	exact orient(p_1, p_2, p_4) < 0
float orient(p_2, p_3, p_4) < 0	exact orient(p_2, p_3, p_4) < 0
float orient(p_3, p_1, p_4) < 0	exact orient(p_3, p_1, p_4) > 0

The first three points form a “pseudo” counter-clockwise triangle and, apart from that, the algorithm believes that point p_4 can see all edges of the triangle, while it can see only the two of them. The result of the algorithm depends on the implementation details.

If the algorithm first searches for an invisible edge, it will search forever and never terminate. If it deletes points on-line from the set of points, it will crash or compute nonsense depending on the details of the implementation.

Failure B2: A point outside the current hull sees a non-contiguous set of edges:

Consider the following points:

p_1 = (0.50000000000001243, 0.50000000000000189)
p_2 = (0.50000000000001243, 0.50000000000000333)
p_3 = (24.00000000000005, 24.000000000000053)
p_4 = (24.000000000000068, 24.000000000000071)
p_5 = (17.300000000000001, 17.300000000000001)

Inserting the first four points results in the convex quadrilateral (p_1, p_4, p_3, p_2); this is correct, as the exact orientation predicate produces the same result.

When processing the point p_5, the orientation takes the following values:

float orient(p_1, p_4, p_5) < 0	exact orient(p_1, p_4, p_5) > 0
float orient(p_4, p_3, p_5) > 0	exact orient(p_4, p_3, p_5) > 0
float orient(p_3, p_2, p_5) < 0	exact orient(p_3, p_2, p_5) < 0
float orient(p_2, p_1, p_5) > 0	exact orient(p_2, p_1, p_5) > 0

The last point p_5 sees only the edge (p_3, p_2) and none of the other three. However, float orient makes p_5 see the edge (p_1, p_4) as well. The subsequences of visible and invisible edges are not contiguous. Since the falsely classified edge (p_1, p_4) comes first, the algorithm, used for this project, inserts p_5 at this edge, removes no other vertex, and returns a polygon that has self-intersections and is not simple.

Such an example consists of a quadrilateral with two nearly parallel sides and the two other sides being very short. These four points should provide a convex hull by the algorithm used. Then, a query point sitting above the middle of one of the long sides might be able to “float-see” the opposite side of the quadrilateral, while continue seeing the edge next to it. This point should not see the two short sides as figure 12 is showing beneath.

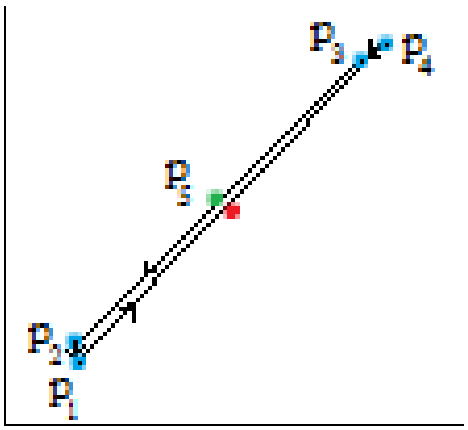


Figure 12:
Presentation of the quadrilateral (p1, p4, p3, p2), which forms the convex hull of these points, and point p5 which sees only the edge (p3, p2) (green spot). However, even though orientation is correctly computed as far as edge (p3, p2) is concerned, it is miscalculated for the edge (p1, p4), which makes p5 see this edge as well (p5 is thought to lie on the red spot).

4.2.3 Distinct Effects of Failures

All the examples seen by now, cover the negation of the correctness properties on the incremental algorithm. The analysis of failures that has proceeded shows the effect of

an incorrect orientation test for a single update step. Nonetheless, since most of the convex hulls that occur could be considered as an approximation to the correct one, one shall regard the problem as insignificant. In the means of clarifying the situation, there are yet some examples were the error is not negligible, but instead pretty obvious.

The algorithm computes a convex polygon, but misses some of the extreme points:

This case is the same as in the example in Failure A_1. This example can be modified so that the ratio of the area of the true hull and the computed hull becomes arbitrarily large. To take this as a result, the fourth point of failure A_1 is chosen towards infinity. The true convex hull has four extreme points, but the algorithm misses p_4.

p_1 = (0.10000000000000001,0.10000000000000001)

p_2 = (0.20000000000000001,0.20000000000000004)

p_3 = (0.79999999999999993,0.80000000000000004)

p_4 = (1.267650600228229*10^{30},1.2676506002282291*10^{30})

float orient(p_1,p_2,p_3) < 0	exact orient(p_1,p_2,p_3) < 0
float orient(p_1,p_3,p_4) = 0	exact orient(p_1,p_3,p_4) < 0
float orient(p_2,p_1,p_4) = 0	exact orient(p_2,p_1,p_4) > 0
float orient(p_3,p_2,p_4) > 0	exact orient(p_3,p_2,p_4) > 0
hull=<p_1, p_3, p_2>	hull=<p1, p4, p3, p2>

The algorithm crashes or does not terminate:

Failure B1 is a correspondent example.

The algorithm computes a non-convex polygon:

The example that follows illustrates a case where the hull is obviously non-convex.

p_1 = (24.00000000000005, 24.000000000000053)

p_2 = (24.0, 6.0)

p_3 = (54.85, 6.0)

p_4 = (54.850000000000357, 61.000000000000121)

p_5 = (24.000000000000068, 24.000000000000071)

p_6 = (6.0, 6.0)

Let’s check how the algorithm processes this set of points.

float orient(p_1, p_2, p_3)>0	exact orient(p_1, p_2, p_3)>0
Hull=<p_1, p_2, p_3>	Hull=<p_1, p_2, p_3>
float orient(p_1, p_2, p_4)>0	exact orient(p_1, p_2, p_4)>0
float orient(p_2, p_3, p_4)>0	exact orient(p_2, p_3, p_4)>0
float orient(p_3, p_1, p_4)<0	exact orient(p_3, p_1, p_4)<0
Hull=<p_1, p_2, p_3, p_4>	Hull=<p_1, p_2, p_3, p_4>
float orient(p_1, p_2, p_5)>0	exact orient(p_1, p_2, p_5)>0
float orient(p_2, p_3, p_5)>0	exact orient(p_2, p_3, p_5)>0
float orient(p_3, p_4, p_5)>0	exact orient(p_3, p_4, p_5)>0
float orient(p_4, p_1, p_5)<0	exact orient(p_4, p_1, p_5)>0

Hull=<p_1, p_2, p_3, p_4, p_5>	Hull= <p_1, p_2, p_3, p_4>
float orient(p_1, p_2, p_6)<0	exact orient(p_1, p_2, p_6)<0
float orient(p_2, p_3, p_6)=0	exact orient(p_2, p_3, p_6)=0
float orient(p_3, p_4, p_6)>0	exact orient(p_3, p_4, p_6)>0
float orient(p_5, p_1, p_6)>0	exact orient(p_4, p_1, p_6)<0
Hull= <p_1, p_6, p_3, p_4, p_5>	Hull= <p_3, p_4, p_6>

The development of the convex hull is illustrated with the use of graphics in Figure 13.

After the insertion of p_1 to p_4, the algorithm gets the convex hull (p_1, p_2, p_3, p_4). This is correct. Point p_5 lies inside the convex hull of the first four points; but float orient(p_4, p_1, p_5) < 0. Thus p_5 is inserted between p_4 and p_1 and the hull is updated to (p_1, p_2, p_3, p_4, p_5). However, this error is not visible yet to the eye, as it is shown in Figure 13(a).

Continuing to the next point, p6 sees the edges (p_4, p_5) and (p_1, p_2), but does not see the edge (p_5, p_1). All of this is correctly determined by float orient. Then the insertion process for point p6 is considered. Depending on where the algorithm starts the search for a visible edge, it will either find the edge (p_4, p_5) or the edge (p_1, p_2). In the former case, p_6 is inserted between p_4 and p_5 and the polygon shown in (b) occurs. It is visibly non-convex and has a self-intersection. In the latter case, p6 is inserted between p_1 and p_2 and the polygon shown in (c) occurs. It is also visibly non-convex.

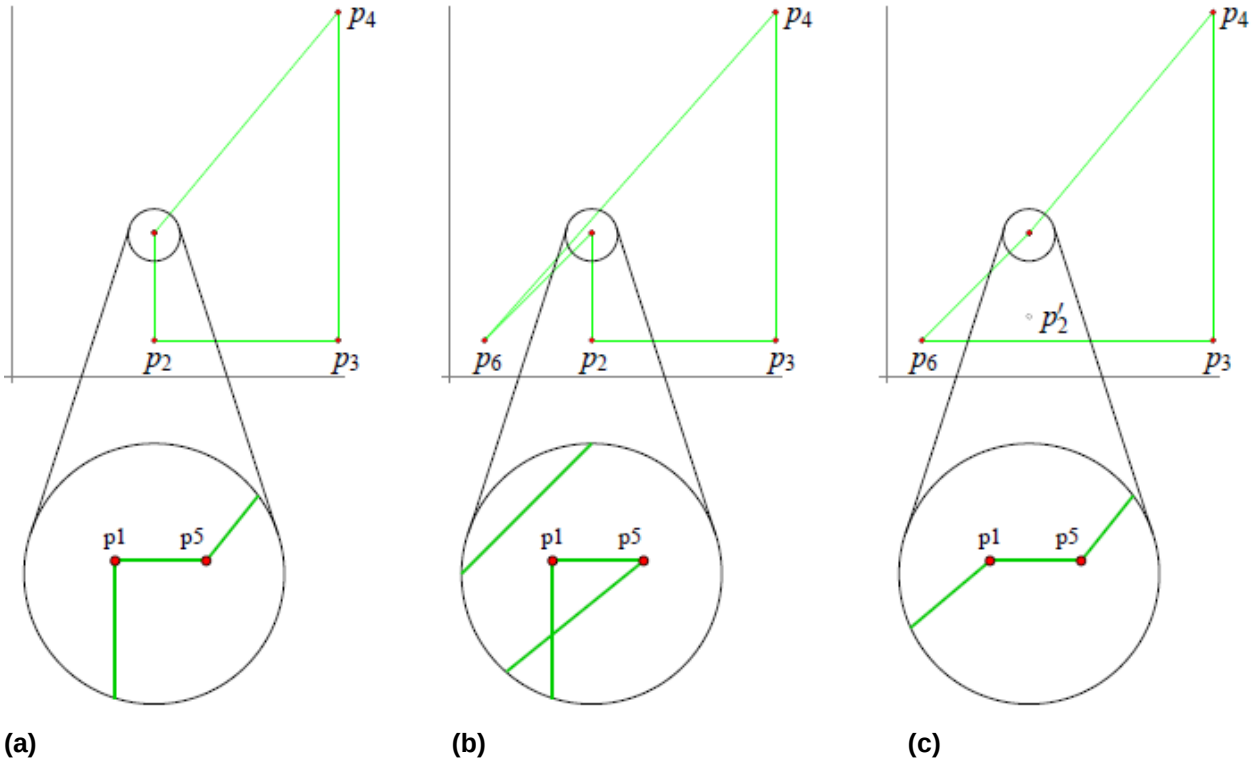


Figure 13:
The hull constructed after processing points p_1 to p_5. Points p_1 and p_5 lie close to each other and are indistinguishable in the upper figure. The magnified schematic view below shows that there is a concave corner at p_5. The point p_6 sees the edges (p_1, p_2) and (p_4, p_5), but does **not** see the edge (p_5, p_1). One of the former edges will be chosen by the algorithm as the chain of edges visible from p_6. Depending on the choice, the hulls shown in **(b)** or **(c)** are obtained. In **(b)**, (p_4, p_5) is found as the visible edge, and in **(c)**, (p_1, p_2) is found. The figures show the coordinate axes to give the reader a frame of reference.

It should be mentioned that during the reproduction of the examples which were presented previously in this thesis, the execution of the code that is provided by [1] has

been of great importance. However, the fact that there are some dependences on external libraries, apart from python and CGAL, such as the LEDA (Library of Efficient Data types and Algorithms) and the GMP (Gnu Multiple Precision), gave us some hard time to proceed to the next step. As a result, the need to develop a virtual machine arose. The certain environment contains all the dependences required by the code and is also provided here for anyone who wants to experiment.

5 FURTHER RESEARCH

The content of this project can be used to support the development of an environment that will assist and supplement the undergraduate courses of Computational Geometry. It concerns a computational system of boosting geometrical algorithms (Geometric algorithm aNimatiOn SYStem, GNOSYS) in two and three dimensions. The utilization of this environment will make possible the reduction of every computational geometry algorithm to an executable multimedia program with graphical input and output. The main target is the development of a tool that will help the students visualize the teaching subject during the courses. For instance, the instructor will have the capability to insert non trivial input data and present a complicated execution designed with accuracy on the screen (instead of a manual design on the board). The students shall watch, instead of imagining, the correct progress of the algorithm executed, even for extreme or degenerate cases. Visualization is highly effective in two and three dimensions, and these are the dimensions where computational geometry occurs in practice. GNOSYS is an ongoing effort for the creation of a system which creates e-content for computational geometry teaching. The system provides geometric and visualization libraries that facilitate the quick creation of interactive visualizations of computational geometry algorithms. Inquiry-based learning is promoted as the learners have the opportunity to observe, interact and experiment with the produced animations [3]. The contribution of this final thesis to the GNOSYS system is the development of a pure Python geometrical library that will replace the use of Python bindings with CGAL, which is yet a project to be completed.

Furthermore, it would be of great interest if this thesis was used in order to boost Python with the MP_Float or another multi precision arithmetic type in a way where the duck-typing would recognize input objects as this type instead of the standard floating point type. The certain capacity should be available for environments which are used as support to the courses of Computational Geometry, and to anyone who needs exact and accurate computations without great requirements for computational speed.

6 CONCLUSION

As a conclusion, many different cases have been presented, where the floating-point arithmetic fails and the corresponding algorithm produces some unexpected results. On the contrary, the implementations using the MP_Float instead of regular float have presented the accurate and exact way in which the computations should proceed. The exactness that characterizes the arithmetic type of MP_Float, is inherited by the module decimal since it is encapsulated in the MP_Float. Moreover, this new type, utilizes the ability that “decimal” provides the user with, which is the predefinition of the precision needed, in order to prevent any rounding errors. Specifically, the user does not need to bother themselves with whether the precision suffices or not, since the MP_Float has redefined all the arithmetic operations so as to acquire the proper precision. The examples that have been reproduced in reference to [1] using pure python are indicative of every possible failure which can arise due to floating point arithmetic, as far as the orientation predicate and the incremental convex hull algorithm are concerned. The floating point implementation presents unexpected results when the points are almost collinear. The MP_Float implementation, instead, produces the expected results since the computations are exact both because it provides representation accuracy and since no rounding due to precision limitations is required. The respective CGAL orientation predicate also performs exact computations. However, due to duck-typing, which is a basic feature of python, the representation of the coordinates may be affected right after the assignment of the values and before the execution of the computations.

The MP_Float type has no further dependencies apart from a python interface. As a result, it can be used to replace standard float when computations with arbitrary precision are desired. Nevertheless, since there is a lack of speed in the computations, this type is more suitable to be used through the teaching process. Specifically, it can supplement the courses of Computational Geometry with the capability of the reduction of every computational geometry algorithm to an executable multimedia program, even for degenerate cases. The implementation of MP_Float is simple but provides a quadratic complexity for multiplications. This can be a problem for large operands. For faster implementations of the same functionality with large integral values, the user may want to consider using GMP or LEDA instead.

ABBREVIATIONS:

CGAL	Computational Geometry Algorithms Library
GNOSYS	Geometric algorithm aNimatiOn SYStem
IEEE	Institute of Electrical and Electronics Engineers
EAFP	Easier to Ask Forgiveness than Permission
OpenGl	Open Graphics Library
LEDA	Library of Efficient Data types and Algorithms

REFERENCES:

[1] Kettner, L., Mehlhorn, K., Pion, S., Schirra, S. & Yap, C. (2006). Classroom Examples of Robustness Problems in Geometric Computations.

[2] Fragoudakis, C., & Setter, O. (2008). CGAL-Python experiences in NUA and TAU. ACS Technical Report No: ACS-TR-363608-03. ACS.

[3] Fragoudakis, C., & Karampatsis, M. (2012). Visualizing Content for Computational Geometry Courses. In “Cases on Inquiry through Instructional Technology in Math and Science”,. Lennex L.C & Nettleton K.F., editors. IGI Global.

[4] Andrew A. M.. Another efficient algorithm for convex hulls in two dimensions. Information Processing Letters, 9:216–219, 1979.

[5] Clarkson K.L. and Shor P.W.. Applications of random sampling in computational geometry, II. Journal of Discrete and Computational Geometry, 4:387–421, 1989.

[6] Graham R. L.. An efficient algorithm for determining the convex hulls of a finite point set. Information Processing Letters, 1:132–133, 1972.

[7] CGAL Homepage, 1998. Computational Geometry Algorithms Library (CGAL) Project. A 7-institution European Community effort.

[8] Overmars M.. Designing the computational geometry algorithms library CGAL. In Lin M. C. and Manocha D., editors, Applied Computational Geometry: Towards Geometric Engineering, pages 53-58, Berlin, 1996. Springer. Lecture Notes in Comp. Sci. No 1148.

[9] Mehlhorn K. and Näher S.. LEDA: a platform for combinatorial and geometric computing. CACM, 38:96-102, 1995

[10] Burnikel C., Könnemann J., Mehlhorn K., Näher S., Schirra S., and Uhrig C.. Exact geometric computation in LEDA. In Proc. 11th ACM Sypm. Comp. Geom., pages C18 – C19, 1995.

[11] Karamcheti V., Li C., Pechtchanski I., and Yap C.. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[12] Halperin D. and Shelton C.R.. A perturbation scheme for spherical arrangements with application to molecular modeling. Comp. Geom.: Theory and Applications, 10, 1998.

[13] Funke S., Klein Ch.. Mehlhorn K. and Schmitt S.. Controlled Perturbation for Delaunay triangulations. In Proc. of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1047-1056, Vancouver, Canada, 2005.

[14] Emiris I. Z., Canny J. F. and Seidel R.. Efficient Perturbations for Handling Geometric Degeneracies. Algorithmica, 19: 219-242, Springer – Verlag New York, 1997.

[15] Edelsbrunner H., Mücke E. P., Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms, ACM Transactions on Graphics, 9(1): 66-104, 1990

[16] Milenkovic V.. Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic. CMU Report CMU-CS-88-168, Carnegie Mellon, 1988.

[17] Milenkovic V.. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In Proc. 30th IEEE Symp. on Foundations of Computer Science, pages 500-505, 1989.

[18] Fortune S.. Numerical stability of algorithms for 2D Delaunay triangulations. In Proc. Eighth Annual Symp. on Comp. Geom. pages 83-92. ACM Press, 1992.

[19] Schorn P. and Fisher F.. How to detect a convex polygon. In Heckbert, P. (ed.), Graphics Gems IV. Academic Press, New York, 1994.

APPENDIX A

Implementation of the Mp_Float arithmetic type, the methods needed and the arithmetic operations.

```
from math import sqrt
from decimal import Decimal, localcontext, getcontext, DivisionByZero


class MP_Float(object):

__slots__ = 'number'


	def __init__(self, number):
		if isinstance(number, MP_Float):		# keep the number as it is
			self.number = number.number
		elif isinstance(number, Decimal):  # an MP_Float is a Decimal number
			self.number = number
		elif isinstance(number, float):
			self.number = Decimal(`number`)
						#	.from_float(number)(Python 2.7 onwards )
			print "Warning: the number", self.number, " has been passed as a
float. There could be a representation error!"
		elif isinstance(number, str):
			if '.' not in number:
				number = number + '.0'
			self.number = Decimal(number)
		else:
			if '.' not in `number`:
				number = `number` + '.0'
			self.number = Decimal(number)


	def string(self, i=None):		# 0<=i<=strlen
		tmpstr=str(self.number)
		if i != None:
			if 0<=i<=len(tmpstr):
				return tmpstr[i]
		return tmpstr[:]


	def length(self):		#string length - don't include the decimal point
		s = str(self.number)
		l=len(s)-1
		return int(l)


	def decimal_part(self):#keep the significant part at the left of the decimal point
		tmp = []
		i=self.find('.')
		for k in range(i):
			tmp.append(self.string(k))
			t=''.join(tmp)
		return t	#self.string[:i]  #add the '' to significants repr


	def decimal_partLen(self):
		for i in range(self.length()):
			if self.string(i)=='.':
				return int(i)
```

```
        def mantissa(self):  #keep the mantissa's digits at the right of the decimal point
                mant = []
                for i in range(self.length()):
                        if self.string(i)=='.':
                                j=i+1
                                strLen = self.length()
                                while j < strLen:
                                        mant.append(self.string(j))
                                        j+=1
                                        m = ''.join(mant)
                                return m

        def mantissaLen(self):
                a=self.length()
                for i in range(a):
                        if self.string(i)=='.':
                                return int(a-i)

        def __repr__(self):
                return 'MP_Float(%s)' % self.number.__str__()

        def __str__(self):
                return self.number.__str__()


        def __neg__(self):      #copy_negate uses no context and is quiet
                return MP_Float(self.number.copy_negate())

        def sqrt(self, precision=None):
                with localcontext() as ctx:
                        if precision == None:
                                ctx.prec = max(self.length(),getcontext().prec)
                        else:
                                ctx.prec = precision
                        return MP_Float(self.number.sqrt())


# other must be MP_Float or string
#->by using float the initial precision can be messed up
        def __cmp__(self, other):
                other=MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = max(self.length(), other.length())
                        return self.number.compare(other.number)


        def __add__(self, other):
                other=MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = self.length()+other.length()
#               ctx.prec = max(self.mantissaLen(), other.mantissaLen())+1
                        return MP_Float(self.number + other.number)

        def __sub__(self, other):
                other=MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = self.length()+other.length()
#               ctx.prec = max(self.mantissaLen(), other.mantissaLen())
                        return MP_Float(self.number - other.number)
```

```
	def __mul__(self, other):
		other=MP_Float(other)
		with localcontext() as ctx:
			ctx.prec = self.length() + other.length() + 1
			return MP_Float(self.number * other.number)

	def __div__(self, other):
		other=MP_Float(other)
		if other.number.is_zero():
			print "Error: Division by zero"
			DivisionByZero()
		else:
			with localcontext() as ctx:
				ctx.prec =
max(self.length(),other.length(),getcontext().prec)
				return MP_Float(self.number / other.number)


	def div(self,other, precision=None):
		other=MP_Float(other)
		if other.number.is_zero():
			print "Error: Division by zero"
			DivisionByZero()
		else:
			with localcontext() as ctx:
				if precision==None:
					ctx.prec =
max(self.length(),other.length(),getcontext().prec)
				else:
					ctx.prec = precision
				return MP_Float(self.number / other.number)


	def __pow__(self, other):
		other = MP_Float(other)
		with localcontext() as ctx:
			ctx.prec = max(self.length(),other.length(),getcontext().prec)
			temp = self.number**other.number
			return MP_Float(temp)


	def power(self, other, prec=None):
		other = MP_Float(other)
		with localcontext() as ctx:
			if prec==None:
				ctx.prec =
max(self.length(),other.length(),getcontext().prec)
			else:
				ctx.prec=prec
			temp = pow(self.number,other.number)
			return MP_Float(temp)


	def __mod__( self , other ):
		other = MP_Float(other)
		with localcontext() as ctx:
			ctx.prec = max(self.length(),other.length(),getcontext().prec)
			return MP_Float(self.number % other.number)


	def mod(self, other, precision=None):
		other = MP_Float(other)
		with localcontext() as ctx:
			if precision==None:
```

```
                            ctx.prec =
max(self.length(),other.length(),getcontext().prec)
                    else:
                            ctx.prec=precision
                    return MP_Float(self.number % other.number)


        def __divmod__( self , other ):
                other=MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = max(self.length(),other.length(),getcontext().prec)
                        try:
                                value = MP_Float(self.number / other.number),
MP_Float(self.number % other.number)
                        except ZeroDivisionError:
                                print "Error: Division by zero"
                                raise
                        else:
                                return value

        def divmodulo( self , other, precision=None ):
                other=MP_Float(other)
                with localcontext() as ctx:
                        if precision==None:
                                ctx.prec =
max(self.length(),other.length(),getcontext().prec)
                        else:
                                ctx.prec=precision
                        try:
                                value=MP_Float(self.number / other.number),
MP_Float(self.number % other.number)
                        except ZeroDivisionError:
                                print "Error: Division by zero"
                                raise
                        else:
                                return value


        def __floordiv__(self, other):
                other=MP_Float(other)
                '''
                if other.number.is_zero():
                        print "Error: Division by zero"
                        DivisionByZero()
                else:
                '''
                with localcontext() as ctx:
                        ctx.prec = max(self.length(),other.length(),getcontext().prec)
                        try:
                                value=MP_Float(self.number // other.number)
                        except ZeroDivisionError:
                                print "Error: Division by zero"
                                raise
                        else:
                                return value


        def floordiv(self, other, precision=None):
                other=MP_Float(other)
                '''
                if other.number.is_zero():
                        print "Error: Division by zero"
                        DivisionByZero()
                else:
                '''
```

```
            with localcontext() as ctx:
                    if precision==None:
                            ctx.prec =
max(self.length(),other.length(),getcontext().prec)
                    else:
                            ctx.prec=precision
                    try:
                            value=MP_Float(self.number // other.number)
                    except ZeroDivisionError:
                            print "Error: Division by zero"
                            raise
                    else:
                            return value


        __radd__ = __add__


        __rmul__ = __mul__


        def __rsub__( self, other ):
            other=MP_Float(other)
            with localcontext() as ctx:
                    ctx.prec = self.length()+other.length()
#           ctx.prec = max(self.mantissaLen(), other.mantissaLen())
                    return MP_Float(other.number - self.number)


        def __rdiv__(self, other):
            other=MP_Float(other)
            '''
            if self.number.is_zero():
                    print "Error: Division by zero"
                    DivisionByZero()
            else:
            '''
            with localcontext() as ctx:
                    ctx.prec = max(self.length(),other.length(),getcontext().prec)
                    try:
                            value=MP_Float(other.number / self.number)
                    except ZeroDivisionError:
                            print "Error: Division by zero"
                            raise
                    else:
                            return value


        def __rmod__( self , other ):
            other = MP_Float(other)
            with localcontext() as ctx:
                    ctx.prec = max(self.length(),other.length(),getcontext().prec)
                    return MP_Float(other.number % self.number)


        def __rdivmod__( self , other ):
            other=MP_Float(other)
            '''
            if self.number.is_zero():
                    print "Error: Division by zero"
                    DivisionByZero()
            else:
            '''
            with localcontext() as ctx:
                    ctx.prec = max(self.length(),other.length(),getcontext().prec)
```

```
                    try:
                            value=MP_Float(other.number / self.number),
MP_Float(other.number % self.number)
                    except ZeroDivisionError:
                            print "Error: Division by zero"
                            raise
                    else:
                            return value


        def __rfloordiv__(self, other):
                other=MP_Float(other)
                '''
                if self.number.is_zero():
                        print "Error: Division by zero"
                        DivisionByZero()
                else:
                '''
                with localcontext() as ctx:
                        ctx.prec = max(self.length(),other.length(),getcontext().prec)
                        try:
                                value=MP_Float(other.number // self.number)
                        except ZeroDivisionError:
                                print "Error: Division by zero"
                                raise
                        else:
                                return value


        def __rpow__(self, other ):
                other = MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = max(self.length(),other.length(),getcontext().prec)
                        temp = other.number**self.number
                        return MP_Float(temp)


        def rpower(self, other, modulo=None ):
                other = MP_Float(other)
                with localcontext() as ctx:
                        ctx.prec = max(self.length(),other.length(),getcontext().prec)
                        tmp = other.number**self.number
                        temp = MP_Float(tmp)
                        return MP_Float(temp.number % modulo)


        def __pos__( self ):
                with localcontext() as ctx:
                        ctx.prec = self.length()
                        return MP_Float(self.number)

        def __abs__( self ):
                with localcontext() as ctx:
                        ctx.prec = self.length()
                        return MP_Float(abs(self.number))

        def __iadd__(self,other):
                with localcontext() as ctx:
                        ctx.prec = self.length()+other.length()
                        self.number+=other.number
                        return self

        def __isub__(self,other):
                with localcontext() as ctx:
                        ctx.prec = self.length()+other.length()
```

```
                self.number-=other.number
                return self

    def __imul__(self,other):
        with localcontext() as ctx:
                ctx.prec = self.length() + other.length() + 1
                self.number*=other.number
                return self

    def __idiv__(self,other):
        with localcontext() as ctx:
                ctx.prec = max(self.length(),other.length(),getcontext().prec)
                self.number/=other.number
                return self

    def __imod__(self,other):
        with localcontext() as ctx:
                ctx.prec = max(self.length(),other.length(),getcontext().prec)
                self.number%=other.number
                return self

    def __ipow__(self,other):
        with localcontext() as ctx:
                ctx.prec = max(self.length(),other.length(),getcontext().prec)
                self.number**=other.number
                return self

    def __ifloordiv__(self,other):
        with localcontext() as ctx:
                ctx.prec = max(self.length(),other.length(),getcontext().prec)
                self.number//=other.number
                return self

    def myfloat(self):          #getfloat
        return float(self.number)
```

APPENDIX B

In this section, the code that follows shows the implementation of data processing and the graphical presentation of the Orientation Geometry with the use of OpenGl and C programming.

```
#include <stdio.h>     // - Just for some ASCII messages
#include <GL/glut.h>   // - An interface and windows
                       //   management library
#include "visuals.h"   // Header file for our OpenGL functions

/////////////// Main Program //////////////////////////

int main(int argc, char* argv[])
{
  // initialize GLUT library state
  glutInit(&argc, argv);

  // Set up the display using the GLUT functions to
  // get rid of the window setup details:
  // - Use true RGB colour mode ( and transparency )
  // - Enable double buffering for faster window update
  // - Allocate a Depth-Buffer in the system memory or
  //   in the video memory if 3D acceleration available
                       //RGBA//DEPTH BUFFER//DOUBLE BUFFER//
  glutInitDisplayMode(GLUT_RGBA|GLUT_DEPTH|GLUT_DOUBLE);


  // Define the main window size and initial position
  // ( upper left corner, boundaries included )
  glutInitWindowSize(660,660);
  glutInitWindowPosition(50,50);

  // Create and label the main window
  glutCreateWindow("Orientation Geometry");

  // Configure various properties of the OpenGL rendering context
  Setup();

  // Callbacks for the GL and GLUT events:

  // The rendering function
  glutDisplayFunc(Render);
  glutReshapeFunc(Resize);
  glutIdleFunc(Idle);

  //Enter main event handling loop
  glutMainLoop();
  return 0;
}
```

```
#include <stdio.h>     // - Just for some ASCII messages
#include <string.h>
#include <Windows.h>
```

```
#include <GL/glut.h>    // - An interface and windows management library
#include "visuals.h"    // Header file for our OpenGL functions


char *fileName="../../workfile2";

void Render()
{
  //CLEARS FRAME BUFFER ie COLOR BUFFER& DEPTH BUFFER (1.0)
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //Clean up the colour of the
window

              // and the depth buffer
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  DisplayModel();
  glutSwapBuffers();            // All drawing commands applied to the
                              // hidden buffer, so now, bring forward
                              // the hidden buffer and hide the visible one
}


void Resize(int w, int h)
{
  // define the visible area of the window ( in pixels )
  if (h==0) h=1;
  glViewport(0,0,w,h);

  // Setup viewing volume
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  glOrtho(-(float)w/4,w, -(float)h/4,h, -3, 3);
}


void Idle()
{  glutPostRedisplay();}


void Setup()  // TOUCH IT !!
{
   //Parameter handling
   glShadeModel (GL_SMOOTH);

   glEnable(GL_DEPTH_TEST);
   glDepthFunc(GL_LEQUAL);  //renders a fragment if its z value is less
//or equal of the stored value
   glClearDepth(1);

   // polygon rendering mode
   glEnable(GL_COLOR_MATERIAL);
   glColorMaterial( GL_FRONT, GL_AMBIENT_AND_DIFFUSE );

   //Set up light source
   GLfloat ambientLight[] = { 0.3, 0.3, 0.3, 1.0 };
```

```
   glLightfv( GL_LIGHT0, GL_AMBIENT, ambientLight );

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);

   // Black background
   glClearColor(0.0f,0.0f,0.0f,1.0f);
}


void DisplayModel()              // Main function for processing data and displaying the image
{
   int a=0,b=0, count=0, flag=0;
   char* buf;
   FILE *fp;

   fp = fopen(fileName, "r");
   if(fp == NULL)
   {
      printf("Error opening file: %s\n", fileName);
      exit(-1);
   }
   buf=(char*)malloc(256*sizeof(char));
   glTranslatef(-140.0, -140.0, 0.0);
   for(b=0;b<256;b++)
   {
      for(a=0;a<256;a++)
      {
         buf[a]=fgetc(fp);
         if(buf[a]==EOF)
         {        printf("reached EOF for line=%d, row=%d\n", b,a);
                  flag=1;
                  break;
         }
         if(buf[a]=='+')
         {
                  glPushMatrix();
                  glColor3f(0.0, 0.0, 1.0);
                  glTranslatef(b*3, a*3, 0.0);     //Translate to the correct point
                  glutSolidCube(3.0);    // each point is represented by a cube
                  glPopMatrix();
         }
         else if(buf[a]=='-')
         {
                  glPushMatrix();
                  glColor3f(1.0, 0.0, 0.0);
                  glTranslatef(b*3, a*3, 0.0);
                  glutSolidCube(3.0);
                  glPopMatrix();
         }
         else if(buf[a]=='=')
         {
                  glPushMatrix();
                  glColor3f(1.0, 1.0, 0.0);
                  glTranslatef(b*3, a*3, 0.0);
                  glutSolidCube(3.0);
                  glPopMatrix();
```

```
            }
            else
                        printf("buf_a=%d , a=%d\n",buf[a], a);
        }
        if(flag==1)
            break;
    }
    fclose(fp);
    free(buf);
}
```

APPENDIX C

Implementation of the orientation geometry algorithm:

```
def get_power_of2(px, fltype):
        if isinstance(px,MP_Float):
                px=px.myfloat()
        hexpx=px.hex()
        pi = hexpx.index('p')
        power=hexpx[pi+1:]
        return fltype(2)**(int(power)-52)

def orientation_geometry(p, q, r, fltype, pivotPoint=1):
        signs=['-','=','+']
        ux=get_power_of2(p.x, fltype)
        uy=get_power_of2(p.y, fltype)
#       line_coefficients(p, q, r, u)
        f=open("workfile", 'w')
        for x in range(256):
                for y in range(256):
                        m=fltype(x)*ux
                        n=fltype(y)*uy
                        j=p.x+m
                        k=p.y+n
                        if  fltype == float:
                                orient = orientation(Point2(j,k), q, r, pivotPoint)
                        elif fltype== MP_Float:
                                orient = orientation(Vector2(j,k),q,r,pivotPoint)
                        else:
                                print "Error: float or MP_Float needed!"
                        sign=signs[1+orient]            #orient ={-1,0,+1}
                        f.write(sign)
        f.close()
```

Implementation of the generator of a triplet of point coordinates which present weird orientation geometry.

```
def write_point_in_file(fl, px, py):
        fl.write(str(px))
        fl.write(' ')
        fl.write(str(py))
        fl.write('\n')
        print str(px), str(py)

def get_disturbance(px,py, x=None, y=None):
        epowerx = get_power_of2(px, MP_Float)
        epowery = get_power_of2(py, MP_Float)
        if x==None:
                x = random.choice(range(11,246))
        if y==None:
                y = random. choice(range(x-10,x+10))
        em=MP_Float(x)*epowerx
        en=MP_Float(y)*epowery
        ex=px+em   #p.x+x*u
        ey=py+en   #p.y+y*u
        tmpx=str(ex)
```

```
        tmpy=str(ey)
        return tmpx, tmpy
def genPoint( alfa=1, beta=0, point=None):
        try:
                seed=time.time()
                r1=random.Random(seed)
                while 1:
                        print "searching for proper points..."
                        if point==None:
                                a=random.choice(range(1,2000))
                                b=random.choice(range(1,20000))
                                c=random.choice(range(1,100))
                                dx=MP_Float(a)/MP_Float(b)*c
                                dy=dx*alfa+beta
                                tmpx=str(dx)
                                tmpy=str(dy)
                                tmpx=tmpx[:18]
                                tmpy=tmpy[:18]
                                dx=MP_Float(tmpx)
                                dy=MP_Float(tmpy)
                        else:
                                dx=MP_Float(point[0][0])
                                dy=MP_Float(point[0][1])
                        point=[]
                        tmpx, tmpy = get_disturbance(dx,dy)
                        point.append((tmpx,tmpy))
                        yield point
        finally:
                pass

def gen_for_orientation(inputfile, numPoints=None):
        E=MP_Float
        F=float
        if numPoints==None:
                numPoints=3  #random.choice(range(3,10))
        print numPoints, "points generated"
        alfa=random.choice(range(1,5))
        beta=random.choice(range(0,30))
        done=0
        while not done:
                point=[]
                for i in range(numPoints):
                        generator=genPoint(alfa,beta)
                        value = generator.next()
                        generator.close()
                        point.append((E(value[0][0]), E(value[0][1])))
                for i in range(20):
                        tmpx, tmpy = get_disturbance(point[0][0],point[0][1])
                        px=E(tmpx)
                        py=E(tmpy)
                        exacto=orientation(Vector2(px,py),
   Vector2(point[1][0],point[1][1]),Vector2(point[2][0],point[2][1]))
floato=orientation(Point2(E.myfloat(px),E.myfloat(py)),
   Point2(E.myfloat(point[1][0]),E.myfloat(point[1][1])),
   Point2(E.myfloat(point[2][0]),E.myfloat(point[2][1])))
                        if exacto!=floato:
                                done = 1
```

```
                                break
        fl=open(inputfile, 'w')
        for i in range(numPoints):
                write_point_in_file(fl,point[i][0], point[i][1])
        fl.close()
        return numPoints
```

APPENDIX D

Implemenation of the incremental convex hull algorithm:

```
def find_first_triangle(Set, CGALflag):
	if CGALflag==1:
		orient=CGAL.orientation
	else:
		orient=orientation
	i=0
	n=len(Set)
	hull=[]
	if i==n:
		return hull #// empty hull
	p_min = Set[i] #*first;  intex of first point
	p_max = p_min
	i+=1
	while i != n and Set[i] == p_min:		# search 2nd distinct point
		i+=1
	if i==n:				#( first == last)  // hull has one point only
		hull.append(p_min)	##hull.push_back( p_min);
		return hull
	if Set[i] < p_min:			#( *first < p_min)
		p_min = Set[i]		#*first;
	else:
		p_max = Set[i]		#*first;
	#// search 3rd non-collinear point
	i+=1
	while i != n and orient(p_min, p_max, Set[i])==0:	#( ++first != last && orientation(
p_min, p_max, *first) == 0) {
		if Set[i] < p_min:		#( *first < p_min) // update left and right extreme points
			p_min = Set[i]	#*first;  // while we are still collinear
		elif p_max < Set[i]:		#( p_max < *first)
			p_max = Set[i]	#*first;
		i+=1
	if i == n:				#( first == last) { // hull has two points only
		hull.append(p_min)
		hull.append(p_max)
		print "hull=[p0, p1]"
		return hull
	#// found proper triangle, create hull with correct orientation
	#// extreme points in counterclockwise (ccw) orientation
	hull.append(p_min)
	if orient(p_min, p_max, Set[i]) > 0:	#( orientation( p_min, p_max, *first) > 0) {
		hull.append(p_max)		#push_back( p_max)
		hull.append(Set[i])		#push_back( *first)
	else:
		hull.append(Set[i])		#push_back( *first);
		hull.append(p_max)		#push_back( p_max);
	for index in range(i+1):
		Set.pop(0)
	return hull

def incr_convex_hull():
	inputfile='inputfl'
	flag=0
	print "\n*** 0: float, 1: vector2, 2: Point_2 (CGAL) ***\n"
```

```
print "FOR PLAIN FLOAT:"
for repeat in range(3):          #3 different cases-> plain float, exact,  CGAL float
    X, num=readPoints(inputfile,repeat)
    pointSet=[]
    for tmppoint in X:
        pointSet.append(tmppoint)
    if repeat==2:
        orient=CGAL.orientation
        flag=1
        print "CGAL's convex_hull_2 results:"
        H=CGAL.convex_hull_2(X)
        print "CGAL HLast = ",H
        print "\nFOR CGAL FLOAT:"
    else:
        orient=orientation
    H=find_first_triangle(X, flag)
    while len(X)!=0:                # for each point in set X
        #print "current hull=",H
        Hlen=len(H)
        point=X.pop(0)
            # start, end intex of edge
        start=0
        end=1
        for j in range(Hlen): #for each edge in H(#edges=#verts)
            if orient(H[start], H[end], point)<0:
            #check previous edges as weakly visible
                for rpt in range(Hlen):
                    if orient(H[(start-1)%Hlen], H[start], point)<=0:
                        start=(start-1)%Hlen
                    else:
                        break
                for rpt in range(Hlen):
                    if orient(H[end], H[(end+1)%Hlen], point)<=0:
                        end=(end+1)%Hlen
                    else:
                        break
                if start < end:
                    n=end-1 #pop from end and back
                    for m in range(start+1,end): # m>start && m<end:
                        H.pop(n)
                        n=n-1
                elif start > end: # but not the last item of the list
                    n=Hlen-1
                    if start<Hlen-1:
                        for m in range(start+1, Hlen):
                            H.pop(n)
                            n=n-1 #due to pop
                    n=end-1
                    for m in range(0, end):
                        H.pop(n)
                        n=n-1
                H.insert(start+1, point)
                break
            else:
                start=(start+1)%Hlen
                end=(end+1)%Hlen
    print '\nHull=',H
```

```
        if repeat==0:
                print "FOR EXACT FLOAT:"
    return 0
```