TP1 - Prise en main

18 septembre 2020

Les TPs comportent une partie à faire sur feuille (explictement indiquée), ainsi qu'une partie implémentation.

L'objectif de ce TP est de vous familiariser avec le fonctionnement de `sage`[1]. Si vous êtes déjà familiarisé.e.s avec, vous pouvez passer les premières questions qui n'ont que peu ou pas de lien avec le cours.
`sage` peut être vu une version de Python incluant de nombreux packages mathématiques. La sémantique du language est la même. Pour plus de détails, vous pouvez vous référer au livre *Calcul mathématique avec Sage* disponible gratuitement en ligne, en version française[2] ou anglaise[3].

> Vous pouvez obtenir la documentation d'une fonction/méthode en ajoutant un '?' à la fin, ou en tapant `help(cmd)`, où `cmd` est la commande en question.
> N'oubliez pas d'abuser de l'autocompletion pour prendre connaissance des méthodes disponibles sur un objet donné.

Exercice 0 : Lancement

1. Ouvrir un terminal et lancer `sage`. Vous êtes dorénavant dans l'interpreteur.
   (a) Si vous souhaitez travailler directement à partir du terminal, vous pouvez passer à l'exercice suivant. Vous pouvez implémenter vos scripts dans un fichier .sage et le lancer à l'aide de la commande suivante, lancée depuis l'interpreteur
   `%runfile path_to_my_script.sage`
2. Si vous souhaitez travailler dans un notebook, suivre les instructions suivantes :
   (a) Installer `jupyter`.
   (b) Lancer `jupyter-notebook` dans un terminal. Un onglet devrait s'ouvrir dans votre navigateur.
   (c) Naviguer jusqu'au dossier de TP et créer une nouvelle feuille de calcul en choisissant l'environnement `sage`.
   (d) Vous pouvez alors entrer vos instructions. Pour les exécuter, cliquez sur *Run* (ou `Shift + Entrer`).
   (e) Vous pouvez créer des cellules différentes pour ne pas exécuter tout le code à chaque fois.

Exercice 1 : Manipulations basiques

> • `ZZ` (ou `Z = Integers()`), `RR` et `QQ` désignent respectivement $\mathbb{Z}$, $\mathbb{R}$ et $\mathbb{Q}$.
> • `Zmod(n)` et `Integers(n)` désignent les entiers modulo n.
> • `P.<x> = PolynomialRing(ZZ)` permet de définir `x` comme la variable dans l'anneau des polynômes définis sur $\mathbb{Z}$.

1. `https://www.sagemath.org`
2. `http://sagebook.gforge.inria.fr/`
3. `http://sagebook.gforge.inria.fr/english.html`

1. Manipuler les expressions symboliques.
   (a) Construire l'expression symbolique $pol = (x^2+5)(x^2-7)$.
   (b) Afficher le développement de pol.
   (c) Évaluer pol en 5.
   (d) Afficher les racines de pol.
2. Fonction factoriel
   (a) Écrire une procédure itérative `fac_it` permettant de calculer $n!$.
   (b) Écrire une version récursive `fac_rec`.
   (c) Comparer les temps d'exécution pour les calculs de 20! et 250! à l'aide de la fonction `timeit`. Comparer avec la fonction `factorial` de `sage`.
3. Manipulation de listes : Créer la liste des nombres premiers inférieurs à 1000
   (a) En utilisant une boucle for et la commande `is_prime`.
   (b) En utilisant la commande `filter`.
   (c) En utilisant une boucle `while` et la méthode `next_prime`.
4. Tracé de fonction
   (a) Construire la fonction $f : x \to x^2 - 3x + 2$.
   (b) Tracer le graphe de f sur $[-20, 20]$ à l'aide de la fonction `plot`.

Exercice 2 : Répartition des nombres premiers

La section 1.4 du cours traite du Théorème des nombres premiers, permettant d'estimer la quantité de nombres premiesr inférieurs à x lorsque x tend vers l'infini. Cette quantité, notée $\pi(x)$, tend vers $x/\log x$.

Ici, l'objectif est de vous rendre compte du comportement asymptotique de la fonction π ainsi que des approximations données dans le cours.

1. Implémenter la fonction `pi`, retournant le nombre de nombres premiers inférieurs ou égaux à l'entrée fournie. Pour des raisons de performance, utilisez la fonction `prime_pi` fournie par `sage` pour la suite.
2. Implémenter les fonctions `lower_approx_pi` et (resp. `upper_approx_pi`) qui prennent une entrée entière x et retournent la borne inferieure (resp. supérieure) définies par Chebyshev :
$$0.9 \cdot \frac{x}{\log x} \leq \pi(x) \leq 1.2 \cdot \frac{x}{\log x}$$
3. Implémenter la fonction `Li` qui prend une entrée entière x et retourne $Li(x) = \int_2^x dt/\log(t)$.
4. Tracer la fonction `pi` ainsi que les fonctions d'approximation sur l'intervalle $[2, 1000]$. Faites de même sur l'intervalle $[2, 10^7]$.

Exercice 3 : PGCD et Bézout

1. *Sur feuille*, en utilisant l'algorithme d'Euclide étendu, calculer le PGCD et les coefficients de Bézout pour 252 et 90.
2. *Sur feuille*, déterminer une solution particulière (x_0, y_0) de l'équation $(E) : 252x + 90y = 36$.
3. *Sur feuille*, trouver toutes les solutions de (E) dans $\mathbb{Z}^2$.
4. Implémenter l'algorithme d'Euclide étendu, et vérifiez vos résultats.

Exercice 4 : Théorème des restes Chinois

Soient m et n deux entiers tels que $pgcd(m, n) = 1$. Le théorème des restes Chinois traduit l'isomorphisme d'anneaux $\phi : \mathbb{Z}/mn\mathbb{Z} \to \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ définit par $\phi(x) = (x \bmod m, x \bmod n)$.

1. *Sur feuille*, donner une formule pour l'application réciproque ϕ^{-1} .
2. *Sur feuille*, à l'aide des résultats obtenus à la question précédente, résoudre le système

$$(S) \begin{cases} x \equiv 3 \bmod 7 \\ x \equiv 4 \bmod 11 \end{cases}$$

3. Vérifier vos résultats sur `sage`.

Exercice 5 : Indicatrice d'Euler et petit théorème de Fermat

1. *Indicatrice d'Euler*
   (a) *Sur feuille*, en vous aidant du théorème des restes Chinois, montrer que φ est une fonction arithmétique multiplicative, c'est à dire que pour a et b premiers entre eux, $\varphi(ab) = \varphi(a)\cdot\varphi(b)$.
   (b) *Sur feuille*, sans calculatrice, calculer $\varphi(n)$ pour $n = 29484000 = 2^5 \cdot 3^4 \cdot 5^3 \cdot 7 \cdot 13$.
2. Soit p un nombre premier, et soit $a \in (\mathbb{Z}/p\mathbb{Z})^*$. En utilisant le petit théorème de Fermat, proposer une méthode pour calculer l'inverse de a. Implémenter cette méthode.

Exercice 6 : RSA

RSA (du nom de ses inventeurs Rivest–Shamir–Adleman) est un cryptosystème asymétrique dont la sécurité repose sur la difficulté de factoriser des nombres dont les facteurs premiers sont très grands (de l'ordre de 2^{1024} à minima aujourd'hui). En se basant sur cette propriété, RSA permet notamment de chiffrer/déchiffrer des messages pour en garantir la **confidentialité**, et de signer/vérifier des messages pour en garantir l'**authenticité**. Intéressons-nous au chiffrement.

Chaque entité dispose d'une clé *publique* et d'une clé *privée* correspondante (d'où le caractère asymétrique du cryptosystème). La clé publique est utilisée pour chiffrer les messages, tandis que la clé privée est utilisée pour les déchiffrer

Une clé publique RSA consiste en un module n et un exposant e, où $n = pq$ est le produit de deux nombres premiers suffisamment grands. Dans cet exercice, on prendra $e = 2^{16} + 1$, qui correspond à la norme actuelle (pour des raisons de rapidité et de sécurité). La clé privée correspondante consiste en le module n et un exposant d tel que $ed = 1 \bmod \varphi(n)$.

Basiquement (en éliminant toutes les mesures de sécurité ajoutées), les opérations de chiffrement et de déchiffrement consistent en une exponentiation modulaire dont l'exposant est respectivement e et d.

Par exemple, si Alice veut transmettre le message m à Bob, elle pourra le chiffrer à l'aide de la clé publique de Bob : $c = m^{e_b} \bmod n_b$. À la réception de c, Bob pourra retrouver le message en calculant $c^{d_b} \bmod n_b$.

1. Pourquoi le déchiffrement fonctionne-t-il ?
2. Implémenter une fonction de génération de clé prennant un paramètre ℓ représentant le taille en bits du module souhaité, et retournant e, d, n.
   (a) La fonction `random_prime` permet de générer un nombre premier aléatoire dans l'intervalle donné.
   (b) Utiliser l'algorithme d'Euclide étendu pour calculer l'inverse modulaire de e. La fonction `inverse_mod` de `sage` permet également d'arriver à ce résultat.

3. Écrire une fonction de chiffrement prennant en entrée un message (chaîne de caractères) et une clé publique, retournant le message chiffré à l'aide de la clé publique. La taille du message (nombre d'octets) doit être strictement inférieure à celle du module.
4. Faire de même avec la fonction de déchiffrement. Vous pouvez tester le bon fonctionnement en partageant votre clé publique avec vos camarades et en échangeant un (ou des) message(s).

> Le RSA tel quel (dit *textbook*) souffre de nombreuses vulnérabilités. En pratique, il est important d'appliquer un padding bien définit avant de chiffrer le message.

5. Le même message a été transmis chiffré à plusieurs destinataires. La clé de chaque destinataire a été utilisée pour chiffrer le message avant envoi. L'exposant publique est $e = 3$ pour chaque clé. On a $c_i = \texttt{Enc}(m_i, 3, n_i)$.

   Supposons qu'un attaquant ait intercepté les trois message chiffrés, et ait connaissance des clés publiques utilisées. Expliquer comment l'attaquant peut retrouver le message. Vous pouvez tester vos resultats sur les valeurs fournies dans le fichier `ex6_material.sage`, disponible sur le drive.